Ensino e aprendizado de fundamentos de programação: uma abordagem baseada em teste de software

Draylson Micael de Souza

Ensino e aprendizado de fundamentos de programação: uma abordagem baseada em teste de software

Draylson Micael de Souza

Orientadora: **Profa. Dra. Ellen Francine Barbosa**

Dissertação apresentada ao Instituto de Ciências Matemáticas e de Computação - ICMC-USP, como parte dos requisitos para obtenção do título de Mestre em Ciências - Ciências de Computação e Matemática Computacional. *VERSÃO REVISADA*

USP – São Carlos
Junho de 2012

Ficha catalográfica elaborada pela Biblioteca Prof. Achille Bassi
e Seção Técnica de Informática, ICMC/USP,
com os dados fornecidos pelo(a) autor(a)

Souza, Draylson Micael de
S719 / Draylson Micael de Souza; orientadora Ellen Francine Barbosa. -- São Carlos, 2012.
102 p.

Dissertação (Mestrado - Programa de Pós-Graduação em Ciências de Computação e Matemática Computacional) -- Instituto de Ciências Matemáticas e de Computação, Universidade de São Paulo, 2012.

1. Engenharia de Software. 2. Teste de Software. 3. Fundamentos de Programação. 4. Ensino e Aprendizagem. I. Barbosa, Ellen Francine, orient. II. Título.

Agradecimentos

Inicialmente, agradeço a Deus por toda a sua bondade e misericórdia.

Agradeço à minha família pelo apoio e o incentivo que me deram desde o início do meu processo de aprendizagem.

Agradeço aos integrantes do Laboratório de Engenharia de Software e a todos os meus amigos pelo apoio que me deram nos momentos de dificuldade.

Agradeço à minha orientadora Ellen Francine Barbosa e aos demais professores do ICMC/USP e de outras instituições que contribuíram para a minha formação.

Por fim, agradeço à FAPESP pelo apoio financeiro fornecido durante o andamento do projeto e aos funcionarios do ICMC pela atenção e serviços prestados.

Resumo

O ensino de fundamentos de programação não é uma tarefa trivial – muitos estudantes têm dificuldades em compreender os conceitos abstratos de programação e possuem visões erradas sobre a atividade de programação. Uma das iniciativas que tem sido investigada a fim de amenizar os problemas associados refere-se ao ensino conjunto de conceitos básicos de programação e de teste de software. A introdução da atividade de teste pode ajudar o desenvolvimento das habilidades de compreensão e análise nos estudantes. Além disso, aprendendo teste mais cedo os alunos podem se tornar melhores testadores e desenvolvedores. Seguindo esta tendência, em trabalhos anteriores foram investigados alguns mecanismos de apoio ao ensino integrado de fundamentos de programação e teste. Dentre os mecanismos investigados destaca-se a proposição de um ambiente de apoio para submissão e avaliação automática de trabalhos práticos dos alunos, baseado em atividades de teste de software – PROGTEST. Em sua primeira versão, a PROGTEST foi integrada à ferramenta JABUTI-SERVICE, que apoia o teste estrutural de programas escritos em Java. O presente projeto de mestrado visou a dar continuidade aos trabalhos já realizados, tendo como principal objetivo a identificação e integração de diferentes ferramentas de teste ao ambiente PROGTEST, explorando tanto técnicas e critérios de teste diferenciados como linguagens de programação distintas. O ambiente PROGTEST também foi aplicado e validado em diferentes cenários de ensino, considerando diferentes linguagens e técnicas de teste. Em linhas gerais, os resultados evidenciam a viabilidade da aplicação do ambiente em cenários de ensino e aprendizagem.

Abstract

The teaching of programming foundations is not a trivial task - many students have difficulty to understand the abstract concepts of programming and have wrong views about the programming activity. Initiatives have been investigated in order to address the related issues. One of them refers to the integrated teaching of programming foundations and software testing. The introduction of testing can help students to develop programming comprehension and analysis skills. Moreover, teaching testing earlier could become the students better testers and developers. Following this perspective, previous studies have investigated mechanisms to support the integrated teaching of programming foundations and software testing. Among them, we highlight the proposition of a environment for the submission and automatic evaluation of programming assignments, based on testing activities - PROGTEST. In its first version, PROGTEST was integrated with JABUTISERVICE tool, which supports the structural testing of Java programs. This work aims at identifying and integrating different testing tools to the PROGTEST environment, exploring both different testing criteria and different programming languages. The PROGTEST environment was also applied and validated in different teaching scenarios, with different languages and testing techniques. In general, the results show the feasibility of applying the environment in the integrated teaching of programming foundations and software testing.

Sumário

Lista de Figuras

Lista de Tabelas

CAPÍTULO

1

Introdução

1.1 Contexto e Motivação

O ensino de fundamentos de programação não é uma tarefa trivial – muitos estudantes têm dificuldades em compreender os conceitos abstratos de programação (Lahtinen et al., 2005) e possuem visões erradas sobre a atividade de programação, como observados em Edwards (2004): (1) uma vez que o compilador aceita o código, todos os erros foram removidos; (2) uma vez que o código produz o resultado esperado em um valor de teste ou dois, ele irá trabalhar bem todo o tempo; (3) o código parece "correto" para os estudantes; (4) uma vez que o código fornece a resposta correta para os dados de exemplo do instrutor, o programa está pronto.

Em geral, na abordagem tradicional, fundamentos de programação são ensinados utilizando linguagens específicas tais como Pascal, C ou Java (Hickey, 2004). Na maioria das vezes, no entanto, ênfase é dada à sintaxe da linguagem, em vez da resolução de problemas por meio do desenvolvimento de algoritmos. Como consequência, os alunos aprendem como programar por meio de uma prática de tentativa e erro, sem desenvolver as habilidades de compreensão e de análise (Edwards, 2004).

Várias iniciativas têm sido investigadas a fim de amenizar os problemas associados ao ensino de fundamentos de programação. Uma dessas iniciativas é a introdução de orientação a objetos (OO) nas disciplinas introdutórias de programação dos cursos de computação (Alphonce e Ventura, 2002). Entretanto, embora a inclusão de OO traga benefícios, vários dos problemas relacionados à programação persistem já que, em geral, os mesmos ocorrem independentemente do paradigma utilizado.

Outra iniciativa investigada refere-se ao ensino conjunto de conceitos básicos de programação e de teste de software. A introdução da atividade de teste pode ajudar o desenvolvimento das habilidades de compreensão e análise nos estudantes, já que para sua condução é necessário que os alunos conheçam o comportamento dos seus programas (Edwards, 2004).

Além disso, experiências recentes têm sugerido que a atividade de teste poderia ser ensinada o mais cedo possível (Barbosa et al., 2008; Corte et al., 2007; Dvornik et al., 2011). A ideia é que alunos que aprendem teste mais cedo podem tornam-se melhores testadores e desenvolvedores uma vez que o teste força a integração e aplicação de teorias e habilidades de análise, projeto e implementação (Barbosa et al., 2003; Jones, 2001; Patterson et al., 2003). De fato, a introdução o quanto antes de conceitos de teste pode: (1) melhorar o raciocínio sobre os programas (e suas respectivas soluções), resultando na melhor qualidade dos produtos desenvolvidos; e (2) induzir e facilitar o uso de técnicas de teste no processo de desenvolvimento de software, levando a uma melhoria na qualidade do processo, em contraste com as práticas atuais.

Apesar de sua relevância, a atividade de teste tem sido pouco explorada nos cursos de graduação. Tal aspecto reforça a necessidade da sua inclusão nos currículos de computação. Edwards (2004) destaca que a aplicação de teste de software em conjunto com fundamentos de programação em disciplinas introdutórias de programação pode tornar os alunos mais cuidadosos no que diz respeito ao desenvolvimento e à compreensão dos algoritmos. Entretanto, o ensino de teste também não é uma atividade trivial. Vários problemas relacionados podem ser apontados (Edwards, 2004; Patterson et al., 2003):

- Teste de software exige que os alunos tenham experiência em programação.
- Os professores precisam avaliar tanto os programas como os casos de teste manualmente.
- Os alunos precisam de *feedback* constante e concreto sobre como melhorar seu desempenho no teste em vários pontos ao longo do desenvolvimento de uma solução ao invés de apenas uma vez no final de uma tarefa.
- Os alunos veem o teste de software como uma atividade cansativa, na qual muito tempo é gasto na realização dos testes e a escrita de planos de teste gera uma grande sobrecarga no trabalho.

Apesar dessas limitações, Barriocanal et al. (2002) mostram que o ensino de teste mais cedo pode melhorar a qualidade do código implementado e pode facilitar o processo de aprendizagem, tanto de teste como de programação.

Dentro da perspectiva apresentada, vários trabalhos vêm sendo conduzidos a fim de fornecer apoio ao ensino integrado de fundamentos de programação e teste de software. Além disso, ambientes automatizados também vêm sendo desenvolvidos em duas perspectivas diferentes. Uma delas é a construção de ambientes de programação pedagógicos que facilitem a construção

de casos de teste apoiando a realização da atividade de teste de unidade. Entre os ambientes desenvolvidos, destacam-se o DRJAVA (Allen et al., 2002) e o BLUEJ (Patterson et al., 2003). Esses ambientes apoiam o ensino de OO e facilitam a construção de casos de teste, podendo ser utilizados pelos alunos para auxiliar o ensino integrado de fundamentos de programação e de teste de software. Entretanto, é importante ressaltar que tais ambientes não estão integrados a ferramentas de teste.

Outra perspectiva refere-se à construção de ambientes que forneçam apoio à submissão e avaliação automática dos programas desenvolvidos pelos alunos, a fim de que os mesmos possam ter um julgamento imediato da sua implementação, mostrando se o programa está ou não correto. Dentre tais ambientes destacam-se MARMOSET (Spacco et al., 2006a,b) e WEB-CAT (Edwards e Perez-Quinones, 2008). Ainda nesse contexto, Corte et al. (Barbosa et al., 2008; Corte, 2006; Corte et al., 2006, 2007) foram investigados mecanismos de apoio ao ensino integrado de fundamentos de programação e teste. Dentre os mecanismos definidos destaca-se a proposição da PROGTEST– um ambiente de apoio para submissão e avaliação automática de trabalhos práticos dos alunos, baseado em atividades de teste de software.

Em linhas gerais, o ambiente PROGTEST permite que os alunos submetam trabalhos práticos de programação; utilizando ferramentas de teste integradas, o ambiente automaticamente avalia o trabalho dos alunos, fornecendo a eles um *feedback* imediato sobre os seus trabalhos. Em sua versão inicial, estavam integrados ao ambiente PROGTEST: (1) o JUNIT (Beck e Gamma, 2010), apoiando a execução automática de casos de teste; e (2) a ferramenta JABUTISERVICE (Eler et al., 2009), fornecendo apoio a aplicação de critérios de teste estrutural; ambos apoiando o teste de programas escritos em Java.

Para avaliar os trabalhos de programação dos alunos, o ambiente requer um "trabalho oráculo", o qual consiste em: (1) um programa de referência fornecida pelo professor, que implementa a solução correta para o trabalho proposto; e (2) um conjunto de teste para o programa fornecido, sendo que este deve ser 100%-adequado aos critérios de teste considerados na avaliação. Desse modo, o aluno pode avaliar suas habilidades: (1) em programação — implementando sua versão de um dado programa e testando-a a partir dos casos de teste do trabalho oráculo; e (2) em teste — construindo e melhorando seus casos de teste a fim de obter um conjunto adequado ao teste de um dado programa, também fornecido por meio do trabalho oráculo.

O presente trabalho visa a dar continuidade ao trabalho de Corte et al. (Barbosa et al., 2008; Corte, 2006; Corte et al., 2006, 2007), em especial no que se refere ao desenvolvimento, validação e uso do ambiente PROGTEST. Os objetivos do trabalho são discutidos na próxima seção.

1.2 Objetivos

Este trabalho tem como principal objetivo investigar a integração de novas ferramentas de teste ao ambiente PROGTEST. Tal integração foi explorada a fim de propiciar a utilização, por parte dos alunos, de diferentes técnicas e critérios na condução de seus testes.

Como consequência da integração de diferentes ferramentas à PROGTEST, a submissão e avaliação de programas em diferentes linguagens de programação também foi considerada e incorporada ao ambiente. A princípio, ênfase foi dada à identificação e seleção de ferramentas de apoio ao teste de programas escritos em Java. A partir dos resultados obtidos com essa experiência, ferramentas de apoio ao teste de programas escritos em outras linguagens introdutórias de programação (C, em especial) também foram consideradas.

Em especial, para programas escritos em Java, a ferramenta JUMBLE (Irvine et al., 2007) foi integrada à PROGTEST, fornecendo apoio à aplicação do critério Análise de Mutantes. Para programas escritos em C, foram integradas as ferramentas: (1) CUNIT (Kumar e St.Clair, 2005), permitindo a execução automática de casos de teste; (2) GCOV[1], apoiando a aplicação de critérios estruturais; e (3) PROTEUM (Delamaro et al., 1993), apoiando a aplicação do critério Análise de Mutantes.

Além disso, em decorrência da integração de novas ferramentas, uma base de trabalhos oráculo para a linguagem C também foi desenvolvida. Ainda, os trabalhos oráculo em Java, que já se encontravam disponíveis na PROGTEST, foram revisitados, adequando os seus conjuntos de teste ao critério Análise de Mutantes, apoiado pela ferramenta JUMBLE.

Por fim, o ambiente PROGTEST foi aplicado e validado tanto em ambientes controlados como em cenários reais de ensino. A ideia foi verificar a viabilizada de aplicação do ambiente em disciplinas de programação e de teste.

1.3 Organização

Neste capítulo foram apresentados o contexto, a motivação e os principais objetivos deste trabalho. O restante deste documento esta organizado da seguinte forma.

No Capítulo 2 são apresentados os principais tópicos de pesquisa relacionados a este trabalho: (1) teste de software, com ênfase em técnicas, critérios e ferramentas associadas; e (2) ensino integrado de programação e teste, com ênfase em ambientes e ferramentas de apoio.

No Capítulo 3 são sumarizados as principais funcionalidades do ambiente PROGTEST, sua arquitetura e aspectos gerais de desenvolvimento. Além disso, aspectos de integração de novas ferramentas de teste também são discutidos.

[1] http://gcc.gnu.org/onlinedocs/gcc/Gcov.html

No Capítulo 4, é ilustrada a aplicação da PROGTEST considerando diferentes linguagens de programação e técnicas de teste. Os resultados obtidos a partir das aplicações conduzidas também são apresentados e discutidos.

Por fim, no Capítulo 5 são discutidas as contribuições de pesquisa proporcionadas pela condução deste trabalho bem como as perspectivas a serem investigadas em trabalhos futuros.

CAPÍTULO

2

Revisão Bibliográfica

2.1 Considerações Iniciais

Neste capítulo são apresentados os principais conceitos e pesquisas relacionadas a este trabalho, identificadas através da realização de revisões bibliográficas da literatura.

Na Seção 2.2 são apresentados os principais conceitos de teste de software. Dentre os conceitos apresentados, destaque foi dado às técnicas e critérios de teste. Além disso, ferramentas de apoio ao teste de software também são apresentadas.

Na Seção 2.3 são discutidas as principais vantagens e limitações da introdução de conceitos de teste de software em disciplinas introdutórias de programação. Dentre as iniciativas investigadas a fim de amenizar os problemas associados destaca-se o desenvolvimento de ambientes e ferramentas de apoio ao ensino integrado de programação e teste de software. Em especial, são descritos os ambientes WEB-CAT e MARMOSET. Tais ambientes fornecem suporte para submissão e avaliação automática dos programas desenvolvidos pelos alunos.

2.2 Teste de Software

A engenharia de software oferece uma série de técnicas e métodos que auxiliam a realização das diversas atividades para o desenvolvimento de um produto de software (Pressman, 2006). No entanto, mesmo com a utilização de tais técnicas e métodos, erros podem ser introduzidos no software. Para garantir que o produto de software possua um determinado grau de qualidade,

atividades que proporcionem a garantia de qualidade do software devem ser realizadas durante todo o processo de desenvolvimento de software. Dentre tais atividades, destacam-se as atividades de V&V (Verificação e Validação), que têm por objetivo verificar se o software está sendo construído de maneira correta e garantir que o mesmo está de acordo com a especificação.

O teste de software é uma das atividades de verificação e validação mais utilizadas (Pressman, 2006). Segundo Myers (2004), o objetivo da atividade de teste é revelar a presença de erros ou defeitos no produto. Uma boa atividade de teste é aquela que consegue determinar casos de teste que façam com que o programa falhe.

Em geral, a atividade de teste é realizada em três fases (Pressman, 2006): (1) teste de unidade, o qual consiste no teste individual de cada módulo de programa, em que deverão ser identificados erros de lógica e programação; (2) teste de integração, o qual se concentra na comunicação entre os módulos do sistema, visando a identificação e correção de erros na interface dos módulos; e (3) teste de sistema, o qual consiste em verificar se todos os módulos combinam-se adequadamente e se a função/desempenho global do sistema é atingida na perspectiva do usuário.

Além disso, para cada fase, quatro etapas devem ser executadas (Delamaro et al., 2007a): (a) planejamento, na qual são definidas as atividades a serem realizadas e os recursos necessários; (b) projeto de casos de teste, a qual consiste em escrever casos de teste com base em técnicas e critérios de teste estabelecidos; (c) execução, a qual consiste em executar os casos de teste projetados; e (d) análise, a qual consiste em avaliar os resultados gerados pela execução dos casos de teste.

Idealmente, uma atividade de teste deveria ser realizada com um conjunto de teste constituído por todos os valores presentes no domínio de entrada do programa, caracterizando assim, o teste exaustivo. Porém, executar todas as possíveis entradas do programa, na maioria das vezes, é uma tarefa inviável ou impraticável. Dessa forma, são definidas técnicas e critérios que permitem, para cada programa, selecionar um subconjunto dos seus possíveis valores de entrada que possam representar o domínio de entrada como todo. Embora não seja possível provar por meio de testes que um programa está correto, a aplicação e técnicas e critérios pode fornecer uma medida objetiva do nível de confiança e qualidade alcançados pelos testes realizados.

É importante ressaltar, também, o aspecto complementar entre as técnicas e critérios de teste. Uma estratégia de teste deve ser elaborada de forma a explorar as vantagens proporcionadas por cada técnica e critério. Estudos teóricos e empíricos têm sido realizados a fim de estabelecer estratégias para aplicação de teste e comparar os diversos critérios presentes (Barbosa et al., 2001; Frankl e Weyuker, 1993; Jorge et al., 2001; Maldonado et al., 2000; Mresa e Bottaci, 1999; Offutt et al., 1996; Prado et al., 2008; Rapps e Weyuker, 1982, 1985; Vincenzi et al., 2001; Zhu, 1996).

Embora a realização sistemática e bem planejada da atividade de teste seja essencial para garantir que o software possua algumas características mínimas, importantes tanto para o estabelecimento de qualidade como para o seu processo de evolução, a aplicação da atividade de teste tem sido apontada entre as mais onerosas no desenvolvimento de software (Pressman, 2006), podendo, em alguns casos, consumir grandes parte dos custos de desenvolvimento. Nesse sentido, ressalta-se a importância de ferramentas de teste que automatizem a aplicação das técnicas e critérios associados. Sem a utilização de ferramentas de apoio, a atividade de teste se torna trabalhosa, propensa a erros e limitada a programas muito simples (Horgan e Mathur, 1992).

Por fim, observa-se que o teste é apenas uma das atividades de garantia de qualidade de software e, em geral, sua utilização isolada não é suficiente para alcançar um produto de boa qualidade. É fundamental que a atividade de teste seja utilizada em complemento a outras atividades da engenharia de software, tais como inspeções, revisões ou técnicas formais e rigorosas de especificação e de verificação, *walkthrough*, entre outras (Pressman, 2006).

2.2.1 Técnicas de Teste

A fim de assegurar que os dados de teste selecionados para a realização da atividade de teste sejam adequados, critérios de teste são estabelecidos de modo a definir os requisitos mínimos que um determinado conjunto de teste deve satisfazer. Basicamente, os critérios são estabelecidos a partir de três técnicas (Delamaro et al., 2007a): (1) funcional, na qual os critérios de teste são estabelecidos a partir da especificação do software; (2) estrutural, na qual os casos de teste são derivados a partir do código fonte do software; e (3) baseada em erros, em que critérios são estabelecidos a partir de erros típicos cometidos pelos desenvolvedores. Tais técnicas são detalhadas a seguir.

2.2.1.1 Técnica Funcional

A técnica funcional, também conhecida como teste caixa preta, trata 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 (Beizer, 1990; Myers, 2004). A técnica de teste funcional envolve dois passos principais: (1) identificar as funções que o software deve realizar; e (2) projetar casos de teste capazes de checar se essas funções estão sendo realizadas pelo software. As funções que o software possui são identificadas a partir da especificação de requisitos do sistema, sem se preocupar com detalhes de implementação. Uma especificação correta e de acordo com os requisitos do usuário é de fundamental importância para apoiar a aplicação dos critérios relacionados a essa técnica.

Dentre os critérios de teste funcional, pode-se destacar (Pressman, 2006):

- Particionamento em Classes de Equivalência: Utilizando como base a especificação do programa, este critério consiste em dividir o domínio de entrada do mesmo em classes de equivalência, as quais devem ser classificadas como: (1) válidas, se correspondem a entradas referentes à utilização esperada do programa; ou (2) inválidas, caso correspondam a situações adversas de utilização do programa. Tal divisão em classes de equivalência pode ser realizada observando os requisitos do programa que envolvem intervalos de valores, tamanho da entrada, valores específicos de entrada, etc. A partir da hipótese de que um único elemento de uma classe pode representar a classe como todo, a ideia é gerar o menor número de casos de teste possível de forma que pelo menos um elemento de cada classes válida seja selecionado. Em seguida, deve-se selecionar um elemento de cada classe inválida e gerar pelo menos um caso de teste distinto para cada um deles.

- Análise do Valor Limite: Este critério complementa o particionamento em classes de equivalência. Ao invés de selecionar um elemento qualquer para representar uma classe, os casos de teste devem ser gerados levando-se em consideração valores nas fronteiras das classes, uma vez que são nesses pontos que se concentra a maioria dos erros. O espaço de saída também deve ser particionado em classes e são exigidos casos de testes que produzam resultados nas fronteiras dessas classes.

- Grafo de Causa-Efeito: Este critério tem por objetivo explorar a combinação das condições de entrada. Consiste em identificar as condições de entrada do programa (causas) e os possíveis comportamentos do programa (efeitos). Em seguida, é elaborado um grafo de causa-efeito, que representa quais combinações de entrada resultam em cada efeito. Este grafo é convertido em uma tabela (tabela de decisão), cujas informações são extraídas e transformadas em casos de teste.

Uma vez que na técnica funcional os requisitos de teste são derivados a partir da especificação, a sua realização depende de uma boa especificação de requisitos. Especificações descritivas e não formais, assim como requisitos imprecisos e informais podem dificultar a realização da técnica e comprometer a qualidade da atividade (Fabbri et al., 2007). Além disso, há dificuldades em quantificar e automatizar sua aplicação (Fabbri et al., 2007). Por fim, outra limitação presente na técnica funcional é o fato de que não se pode garantir que partes críticas do software sejam executadas durante a atividade de teste, visto que detalhes da implementação não são considerados pela técnica (Fabbri et al., 2007).

Como vantagens, a técnica funcional pode ser aplicada em todas as fases de teste e sua aplicação não depende do paradigma de programação utilizado. Além disso, a mesma permite identificar determinados tipos de erros como, por exemplo, funcionalidades ausentes.

2.2.1.2 Técnica Estrutural

A técnica de teste estrutural, também conhecida como teste caixa branca (em oposição ao nome caixa preta), baseia-se na derivação de casos de teste a partir dos detalhes de implementação (Barbosa et al., 2005). A maioria dos critérios desta técnica utilizam uma representação do programa chamada Grafo de Fluxo de Controle (GFC) ou "grafo do programa" (Barbosa et al., 2007).

O GFC é um dígrafo, em que cada nó representa um bloco de comandos do programa no qual: (1) se o primeiro comando do bloco for executado, sempre os demais comandos do bloco serão executados sequencialmente; e (2) não há desvio de fluxo para um comando no meio do bloco. O nó inicial do grafo indica o ponto de início da execução do programa e os nós finais os pontos de término da execução. No mesmo sentido, os arcos do GFC representam os desvios de fluxo entre os blocos de comandos.

Os critérios de teste estrutural, geralmente, são classificados em três tipos (Barbosa et al., 2007): (1) critérios baseados em fluxo de controle; (2) critérios baseados em fluxo de dados; e (3) critérios baseados na complexidade.

Dentre os critérios baseados em fluxo de controle, encontram-se os critérios Todos-Nós e Todos-Arcos (Myers, 2004), os quais requerem, respectivamente, que todos os nós e todos os arcos do GFC do programa sejam exercitados. Outro critério baseado em fluxo de dados é o Todos-Caminhos (Myers, 2004), que requer que todos os caminhos possíveis do grafo sejam exercitados.

Os critérios baseados em fluxo de dados envolvem definições de variáveis e referências as tais definições para derivar os requisitos de teste. Uma motivação para a proposição desses critérios foi a indicação de que os critérios baseados unicamente em fluxo de controle não são eficazes para detectar erros simples e trivial, até mesmo em programas pequenos (Barbosa et al., 2007). Além disso, os mesmos foram criados de forma a estabelecer critérios mais rigorosos, uma vez que o critério Todos-Caminhos é na maioria das vezes impraticável.

Dentre os critérios baseados em fluxo de dados, destacam-se os critérios definidos por Rapps e Weyuker (1982, 1985). Tais critérios baseiam-se no Grafo Def-Uso (GDU) — uma extensão do GFC. O GDU é um GFC no qual são anotados os pontos em que uma variável recebe um valor (definições) e os pontos em que esses valores são utilizados (usos). Os usos podem ser de dois tipos: (1) usos computacionais (*c-usos*), que afetam diretamente uma computação sendo realizada; e (2) usos predicativos (*p-usos*), que afetam diretamente o fluxo de controle do programa.

O critério mais básico da família de critérios definidos por Rapps e Weyuker (1982, 1985) é o Todas-Definições, o qual requer que para cada definição de variável, pelo menos um uso seja exercitado. Dentre os critérios dessa família, o critério Todos-Usos tem sido o mais utilizado e investigado. Todos-Usos requer que para cada definição de variável todos os usos associados

sejam exercitados, por meio de pelo menos um caminho livre de definição, ou seja, um caminho onde a variável não seja redefinida.

É importante ressaltar a relação de inclusão existente entre os critérios. O critério Todos-Arcos, por exemplo, inclui o critério Todos-Nós. Isto significa que se um conjunto de teste é adequado ao critério Todos-Arcos, ele também é adequado ao critério Todos-Nós. Da mesma forma, o critério Todos-Caminhos inclui o critério Todos-Arcos e, consequentemente, o critério Todos-Nós.

Outra família de critérios de fluxo de dados são os critérios Potenciais-Usos (Maldonado, 1991). Em linhas gerais, esses critérios eliminam a necessidade de uma ocorrência explícita de um uso a uma definição de variável. Da mesma forma que os critérios definidos por Rapps e Weyuker (1982, 1985), os critérios Potenciais-Usos utilizam o Grafo Def-Uso como base para o estabelecimento de requisitos. Na verdade, basta ter o Grafo Def (Maldonado, 1991), extensão do GFC associando a cada nó as definições que nele ocorrem.

Os critérios básicos que fazem parte da família de critérios Potenciais-Usos são (Maldonado, 1991):

- Todos-Potenciais-Usos: Requer que pelo menos um caminho livre de definição de uma variável definida em um nó *i* para todo nó e todo arco possível de ser alcançado a partir de *i* seja alcançado.

- Todos-Potenciais-Usos/Du: Requer que pelo menos um potencial-du-caminho[1] a uma variável x definida em i para todo nó e para todo arco possível de ser alcançado a partir de i seja exercitado.

- Todos-Potenciais-Du-Caminhos: Requer que todos os potenciais-du-caminhos com relação a todas as variáveis x definidas e todos os nós e arcos possíveis de serem alcançados a partir dessa definição sejam exercitados.

Por fim, os critérios baseados na complexidade utilizam informações sobre a complexidade do programa para derivar requisitos de teste. Dentre eles, podemos citar o Critério de McCabe (McCabe, 1976), também conhecido como Teste do Caminho Básico. Este critério baseia-se no conceito de complexidade ciclomática, a qual define o número de caminhos independentes de um programa. Um caminho independente é qualquer caminho que introduz pelo menos uma nova aresta ainda não atravessada pelos caminhos anteriores. No Critério de McCabe, caminhos independentes devem ser identificados e casos de teste que exercitam esses caminhos devem ser projetados e executados. A complexidade ciclomática consiste, então, no limite máximo

[1]Um potencial-du-caminho em relação à variável x é um caminho livre de definição n_1, ..., n_j, n_k com relação a x do nó n_1 para o nó n_k e para o arco (n_j, n_k), onde o caminho $(n_1, ..., n_j)$ é um caminho livre de laço e no nó n_1 ocorre uma definição de x.

do número de casos de teste necessários para garantir a cobertura de todas as instruções do programa (Pressman, 2006).

Os critérios estruturais têm sido amplamente utilizados no teste de unidade. No entanto, há vários esforços, em diferentes contextos, em estender os critérios de fluxo de dados para o teste de integração (Haley e Zweben, 1984; Harrold e Soffa, 1991; Jin e Offut, 1995; Linnenkugel e Müllerburg, 1990; Vilela, 1998).

Ainda, a definição desses critérios foi feita originalmente para o teste de programas procedimentais, mas vem sendo estendida ao longo dos anos para se adequar ao teste orientado a objetos (Vincenzi, 2005). Dentre eles, Vincenzi (2005) adaptou critérios estruturais tradicionais para o teste de unidade de programas OO e componentes de software. Ao todo, oito critérios foram definidos, os quais são separados em dois conjuntos disjuntos (Barbosa et al., 2005): (1) os que podem ser executados durante a execução normal do programa, denominados independentes de exceção; e (2) os que para serem cobertos exigem obrigatoriamente que uma exceção tenha sido gerada, denominados dependentes de exceção. Desse modo, foram estabelecidos os seguintes critérios:

- $All - Nodes_{ei}$: Todos-Nós independentes de exceção.
- $All - Nodes_{ed}$: Todos-Nós dependentes de exceção.
- $All - Edges_{ei}$: Todas-Arestas independentes de exceção.
- $All - Edges_{ed}$: Todas-Arestas dependentes de exceção.
- $All - Uses_{ei}$: Todos-Usos independentes de exceção.
- $All - Uses_{ed}$: Todos-Usos dependentes de exceção.
- $All - Potencial - Uses_{ei}$: Todos-Potenciais-Usos independentes de exceção.
- $All - Potencial - Uses_{ed}$: Todos-Potenciais-Usos dependentes de exceção.

Com respeito aos problemas associados à técnica estrutural destaca-se a impossibilidade de determinar automaticamente requisitos não executáveis. Tais requisitos referem-se a elementos de um programa (nós, arestas, usos, etc.) que são impossíveis de serem exercitados, geralmente, por questões relacionadas a lógica do programa em questão

Outro problema relacionado são os requisitos ausentes. Quando uma funcionalidade não é implementada, o programa não terá os requisitos de teste correspondentes a essa funcionalidade.

Embora tais limitações, o rigor na definição dos requisitos de teste permite obter medidas objetivas sobre a adequação dos conjuntos de teste e facilitam a automatização desses critérios. Além disso, o teste estrutural pode atuar como um complemento ao teste funcional. Os casos de teste funcional podem servir como um conjunto inicial de casos de teste para o teste

estrutural. Como na maioria das vezes os casos de teste funcional não são adequados aos critérios estruturais, mais casos de teste podem ser adicionados ao conjunto, explorando o aspecto complementar entra as duas técnicas (Barbosa et al., 2005).

2.2.1.3 Técnica Baseada em Erros

Na técnica baseada em erros os requisitos de teste são derivados com base nos erros típicos cometidos pelos desenvolvedores e nas abordagens que podem ser utilizadas para detectar a sua ocorrência (Delamaro et al., 2007c). Dentre os critérios baseados em erros podemos citar o Semeadura de Erros (Budd et al., 1980) e o Análise de Mutantes (DeMillo et al., 1978).

No critério Semeadura de Erros defeitos são inseridos automaticamente no programa. Em seguida, um conjunto de casos de teste deve ser executado, verificando a partir do total de defeitos quantos são naturais e quantos são artificiais. Utilizando estimativas de probabilidade, são estimados, a partir da quantidade de defeitos artificiais não identificados, quantos defeitos naturais não foram revelados.

O critério Semeadura de Erros possui algumas limitações, dentre elas destacam-se (Barbosa et al., 2005): (1) os defeitos artificiais podem interagir com os defeitos naturais, fazendo com que os defeitos naturais sejam "mascarados" pelos defeitos semeados; (2) para obter resultados não questionáveis, é necessário obter programas que possam ter 10.000 defeitos ou mais; e (3) assume-se que os defeitos naturais estão distribuídos uniformemente pelo programa, o que, em geral, não ocorre.

No critério Análise de Mutantes duas hipóteses são exploradas (DeMillo et al., 1978). A "hipótese do programador competente" assume que programadores experientes escrevem programas corretos ou muito próximo do correto. Os erros inseridos em seus programas consistem em pequenos desvios sintáticos que alteram a lógica do programa. O "efeito de acoplamento" assume que os erros complexos de um programa estão relacionados a erros simples. Nesse sentido, estudos empíricos mostram que conjuntos de teste capazes de revelar a presença de erros simples também são capazes de revelar a presença de erros complexos (Acree et al., 1979; Budd et al., 1980).

A ideia da Análise de Mutantes é inserir no programa defeitos típicos cometidos pelos desenvolvedores, verificando se o conjunto atual de casos de teste é capaz de identificá-los. Quando o conjunto de teste não é capaz de identificar um erro, novos casos de teste devem ser inseridos, a fim de aumentar a adequação do conjunto de teste.

Os defeitos que podem ser inseridos no programa são modelados por operadores de mutação. Por exemplo, o operador de mutação $ORRN$ substitui cada operador relacional do programa por outros operadores relacionais; já o operador $CCCR$ substitui cada constante do programa por outras constantes. Aplicando-se ao programa os operadores de mutação, vários

programas modificados serão gerados com base nas alterações definidas por cada operador de mutação. Tais programas são chamados programas mutantes.

Em seguida, um conjunto de casos de teste inicial deve ser executado no programa original e nos programas mutantes. Se um mutante apresentar para pelo menos um caso de teste uma saída diferente do programa original, diz-se que ele é um "mutante morto". Caso contrário, diz-se que ele é um "mutante vivo". Deve-se, então, projetar e executar novos casos de teste para "matar" os mutantes que ainda estão "vivos".

Outro aspecto relevante associado ao critério é a equivalência de programas. Dois programas são ditos equivalentes quando não é possível projetar um caso de teste que faça com que estes programas produzam saídas diferentes. Nesse sentido, quando o programa original é equivalente a um mutante, o mutante não pode ser "morto". Neste caso, dizemos que ele é um "mutante equivalente".

O critério Análise de Mutantes fornece uma medida objetiva sobre a adequação dos casos de teste chamado escore de mutação, o qual é calculado da seguinte forma (Delamaro et al., 2007c):

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

onde:

- $DM(P,T)$: o 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 entre 0 e 1, sendo que quanto maior for o escore de mutação, maior é adequação do conjunto de casos de teste para o programa sendo testado.

Assim como os critérios de fluxo de dados, o critério Análise de Mutantes tem sido utilizado essencialmente no teste de unidade. Entretanto, existem propostas que estendem o mesmo para o teste de integração. Delamaro e Maldonado (1997), por exemplo, propuseram o critério Mutação de Interface, que visa explorar o conceito de mutação no teste de interface.

Dentre os problemas associados ao teste baseado em erros, encontra-se o seu alto custo de aplicação. No entanto, a técnica tem se mostrado a mais eficiente em detectar a presença de erros.

2.2.2 Ferramentas de Teste

Um dos aspectos relevantes referentes à atividade de teste é o desenvolvimento de ferramentas que automatizem a aplicação das técnicas e critérios associados. Sem o apoio de tais ferramentas, a atividade de teste tende a ser extremamente trabalhosa, limitando-se a programas peque-

nos (Delamaro et al., 2007b). Além disso, a aplicação totalmente manual favorece a ocorrência de erros por parte do testador.

Nesta seção, serão sintetizadas algumas das principais ferramentas de teste existentes. Visando os objetivos deste trabalho (Seção 1.2), ênfase foi dada a identificação de ferramentas de apoio à aplicação de critérios estruturais e baseados em erros, tanto para programas escritos Java como para programas escritos em C. Ainda, *frameworks* de teste de unidade automatizados também foram investigados. Tais *frameworks* são brevemente apresentados a seguir.

2.2.2.1 *Frameworks* de Teste de Unidade Automatizados

Frameworks de teste têm sido bastante utilizados como mecanismos de apoio ao desenvolvimento de testes de unidade (Meszaros, 2007). Em linhas gerais, tais *frameworks* permitem que os desenvolvedores escrevam métodos capazes de executar casos de teste em uma unidade do programa. Tais casos de teste podem, então, ser executados automaticamente quantas vezes forem necessárias, verificando quais deles passaram pelo teste e quais deles falharam.

Uma das principais vantagens em utilizar *frameworks* de teste é que eles permitem executar um grande número de casos de teste ao mesmo tempo. Além disso, sem acréscimo de esforço, o mesmo conjunto de teste pode ser re-executado após alguma alteração ser realizada no programa, a fim de verificar se foi inserido um novo erro no programa ou se algum defeito foi corrigido. No entanto, a dependência do julgamento humano para interpretar os resultados é uma de suas limitações.

Atualmente, existe uma família de *frameworks* de teste de unidade nomeada XUNIT (Meszaros, 2007). Estes *frameworks* são baseados no projeto de Beck (1994), que desenvolveu o SUNIT, um *framework* de teste para a linguagem de programação Smalltalk. Dentre estes *frameworks*, um que tem sido amplamente utilizado é o JUNIT (Beck e Gamma, 2010), o qual permite o desenvolvimento de testes de unidade para a linguagem Java. Outros *frameworks* da família XUNIT são: CUNIT (Kumar e St.Clair, 2005), para C; CPPUNIT[2], para C++; NUNIT[3], para C#; FUNIT[4], para Fortram; FPCUNIT[5], para Free Pascal; DUNIT[6], para Delphi; HTTPUNIT[7], para teste de *web sites*; entre outros.

2.2.2.2 Ferramentas de Teste Estrutural

No teste estrutural, os requisitos de teste são derivados a partir da implementação do programa. Existem várias ferramentas que auxiliam a aplicação da técnica estrutural derivando tais requi-

[2]http://cppunit.sourceforge.net/
[3]http://www.nunit.org/
[4]http://www.funit.org
[5]http://camelos.sourceforge.net/fpcUnit.html
[6]http://dunit.sourceforge.net/
[7]http://httpunit.sourceforge.net/

sitos de teste automaticamente e verificando quais deles foram cobertos pelo conjunto de teste executado. Dentre as principais ferramentas de teste estrutural identificadas, ressaltam-se as seguintes:

- **CLOVER**[8]: É uma ferramenta para teste estrutural de unidade em programas Java. Foi desenvolvida pela *Atlassian*[9], uma companhia australiana que desenvolve ferramentas de desenvolvimento de software e colaboração, com o propósito de ajudar as equipes de desenvolvimento a aumentar sua produtividade.

- **COBERTURA**[10]: É uma ferramenta de teste estrutural de unidade de programas Java. Foi construída a partir da ferramenta comercial JCOVERAGE, desenvolvida pela *jcoverage ltd.*[11]. A ideia era fornecer à comunidade Java uma alternativa mais aberta do que a JCOVERAGE e a CLOVER.

- **EMMA**[12]: É uma ferramenta de teste estrutural de unidade de programas em Java. Ela é *open source* e livre para o desenvolvimento comercial. A EMMA diferencia-se das demais buscando apoiar o desenvolvimento em larga escala de software empresarial, porém, mantendo-se ágil e interativa para desenvolvedores individuais.

- **GCOV**[13] **e LCOV**[14]: A GCOV é uma ferramenta de teste estrutural de unidade de programas C. A ferramenta faz parte do GCC[15] (*GNU Compiler Collection*), compilador padrão da maioria dos sistemas baseados no Unix. A ferramenta LCOV estende a GCOV, gerando relatórios em páginas HTML, sobre a análise de cobertura realizada pela GCOV.

- **GROBOCODECOVERAGE**[16]: É uma ferramenta de teste de unidade de programas em Java. Faz parte do GROBOUTILS, um projeto que visa expandir as possibilidades de teste em Java. O GROBOUTILS[17] possui vários sub-projetos que focam em diferentes aspectos de teste em programas Java.

- **HANSEL**[18]: É uma ferramenta que estende o JUNIT, adicionando suporte ao teste estrutural. O HANSEL foi baseado no GRETEL[19], uma ferramenta desenvolvida na *University*

[8] http://www.atlassian.com/software/clover/
[9] http://www.atlassian.com/
[10] http://cobertura.sourceforge.net/
[11] http://www.jcoverage.com/
[12] http://emma.sourceforge.net/
[13] http://gcc.gnu.org/onlinedocs/gcc/Gcov.html
[14] http://ltp.sourceforge.net/coverage/lcov.php
[15] http://gcc.gnu.org/
[16] http://groboutils.sourceforge.net/codecoverage/
[17] http://groboutils.sourceforge.net
[18] http://hansel.sourceforge.net/
[19] http://www.cs.uoregon.edu/research/perpetual/dasada/Software/Gretel/

of Oregon. Basicamente, o HANSEL possui todas as funcionalidades do GRETEL e, adicionalmente, é compatível com o JUNIT.

- **JABUTI (*Java Bytecode Understand and Testing*)** (Vincenzi et al., 2003): Foi desenvolvida pelo Grupo de Engenharia de Software do ICMC/USP em colaboração com outras instituições. Consiste em um conjunto de ferramentas (cobertura, *slicing* e coleta de métricas) que fornecem apoio ao teste estrutural de programas e componentes Java. A ferramenta de cobertura pode ser usada para avaliar a qualidade de um determinado conjunto de teste ou para gerar um conjunto de teste baseado em diferentes critérios de controle de fluxo e de fluxo de dados. A ferramenta de *slicing* pode ser usada para identificar as regiões propensas a falhas no código, sendo útil para depuração e também para a compreensão do programa. Por fim, a ferramenta de coleta de métricas pode ser usada para identificar a complexidade e o tamanho de cada classe sob análise, com base em informações estáticas.

- **JABUTISERVICE** (Eler et al., 2009): É uma ferramenta desenvolvida no ICMC/USP com base na implementação da ferramenta JABUTI original. Basicamente, foi implementada por meio da remoção da interface gráfica da JABUTI e do reúso do seu núcleo. A JABUTISERVICE é uma ferramenta de teste projetada como um *web service*. Assim, fica disponível *on-line*, permitindo que aplicações clientes conectem-se a ela e utilizem o seu núcleo.

- **POKE-TOOL (*Potencial Uses Criteria Tool for Program Testing*)** (Chaim, 1991; Maldonado et al., 1989): Foi desenvolvida na FEEC/UNICAMP em colaboração com o ICMC/USP. Inicialmente a ferramenta foi desenvolvida para o teste de unidade de programas escritos em C e, posteriormente, devido à sua característica de multilinguagem, também foram propostas configurações para o teste de programas em Cobol e Fortran.

2.2.2.3 Ferramentas de Teste Baseado em Erros

Como visto na Seção 2.2.1.3, um dos critérios mais eficazes em revelar a presença de erros em um programa é a Análise de Mutantes. No entanto, a sua aplicação é bastante custosa, uma vez que, em geral, mesmo para programas pequenos, centenas de mutantes devem ser gerados, executados e analisados. Existem diversas ferramentas que auxiliam a aplicação da Análise de Mutantes, automatizando a geração de programas mutantes, a execução dos casos de teste, a análise dos mutantes vivos e o cálculo do *score* de mutação.

Dentre as principais ferramentas de apoio ao critério Análise de Mutantes, destacam-se:

- **JESTER (JUNIT *Test Tester*)** (Moore, 2001): É uma ferramenta que apoia a aplicação do critério Análise de Mutantes para o teste de programas em Java, com de casos de teste

implementados com o JUNIT. Há também as variações PESTER[20], para programas em Python; e NESTER[21], para programas em C#.

- **JUMBLE** (Irvine et al., 2007): Foi desenvolvida pela companhia *Reel Two*[22] e atualmente está disponível como uma ferramenta *open source*. Assim como a JESTER, apoia a aplicação do critério Análise de Mutantes com casos de teste implementados pelo JUNIT. No entanto, visa obter um melhor desempenho na geração dos mutantes e na execução dos casos de teste. A JUMBLE deve ser operada por meio de uma ferramenta de linha de comando.

- **MUJAVA** (Ma et al., 2006; Offutt et al., 2006): Apoia a aplicação do critério Análise de Mutantes para o teste de programas em Java. Foi desenvolvida colaborativamente pelas universidades *Korea Advanced Institute of Science and Technology* (KAIST) na Coreia do Sul e *George Mason University* nos EUA. Tem como diferencial a utilização de operadores de mutação em nível de classes, além dos operadores tradicionais.

- **PROTEUM** (*PROgram TEsting Using Mutants*) (Delamaro et al., 1993): É uma ferramenta desenvolvida no ICMC/USP que fornece apoio à aplicação do Análise de Mutantes em programas escritos em linguagem C. A ferramenta está disponível para os sistemas operacionais SunOS, Solaris e Linux.

2.3 Ensino Integrado de Programação e Teste de Software

Como visto na Seção 1.1, o ensino de fundamentos de programação não é uma tarefa trivial — muitos estudantes têm dificuldades em compreender os conceitos abstratos de programação (Lahtinen et al., 2005) e possuem visões erradas sobre a atividade de programação Edwards (2004).

A fim de amenizar os problemas associados, uma das iniciativas investigada refere-se ao ensino conjunto de conceitos básicos de programação e de teste de software. De fato, a introdução da atividade de teste pode ajudar o desenvolvimento das habilidades de compreensão e análise nos estudantes, já que para sua condução é necessário que os alunos conheçam o comportamento dos seus programas (Edwards, 2004). Além disso, experiências têm sugerido que o teste de software poderia ser ensinado mais cedo no processo de aprendizagem (Barbosa et al., 2003; Jones, 2001; Patterson et al., 2003), criando assim uma "cultura de teste" entre os alunos.

[20] http://jester.sourceforge.net/
[21] http://nester.sourceforge.net/
[22] http://www.reeltwo.com

Várias abordagens vêm sendo utilizadas na tentativa de explorar o ensino integrado de programação e teste. Entre elas, destaca-se o Ensino Dirigido a Testes, o qual é brevemente apresentado a seguir.

2.3.1 Ensino Dirigido a Teste

O Ensino Dirigido a Testes (TDL - *Test-Driven Learning*) é uma proposta de ensino de programação onde os novos conceitos são introduzidos e explorados por meio de testes de unidade automatizados (Janzen e Saiedian, 2006). Foi proposto visando motivar o uso de teste automatizado tanto na atividade de projeto como na atividade de verificação (Janzen e Saiedian, 2008). Uma das vantagens do TDL é que ele pode ser aplicado em diferentes níveis de um currículo de ciências de computação, desde alunos iniciantes até programadores profissionais. Janzen e Saiedian (2006) destacam os principais objetivos do TDL:

- **Ensinar teste sem custos:** Uma das ideias propostas pelo TDL consiste em ensinar conceitos de programação utilizando casos de teste para demonstrar o comportamento dos programas. Neste sentido, o ensino de teste é realizado ao mesmo tempo em que conceitos de programação são introduzidos, sem a necessidade de alocar parte da carga horária do curso para o ensino de teste de software.

- **Ensinar *frameworks* de teste de unidade automatizados de maneira simples:** Ao introduzir o uso de *frameworks* de teste gradualmente nos cursos, proporciona-se uma maior familiaridade dos alunos com os mesmos. Dependendo da linguagem e do ambiente, os professores podem introduzir os *frameworks* no início do curso ou gradualmente, conforme novos conceitos de programação e teste forem ensinados.

- **Encorajar o uso do *test-driven development*:** O TDD (*Test-Driven Development*) (Beck, 2003) é uma proposta de desenvolvimento que consiste em desenvolver testes de unidade automatizados antes de escrever a unidade funcional correspondente. O TDD é uma das principais práticas realizadas no XP (*eXtreme Programming*) e está sendo integrado cada vez mais em vários processos de desenvolvimento.

- **Melhorar a compreensão e as habilidades de programação dos alunos:** Quando os alunos observam, além da interface, o comportamento do programa em um exemplo com os testes, eles tendem a compreender um conceito mais rapidamente do que vendo simplesmente a interface em um exemplo tradicional.

- **Melhorar a qualidade do software tanto em termos de projeto como na densidade de defeitos:** Uma vez que os alunos adquiram, desde o princípio do processo de aprendizagem, os hábitos que levam a um bom projeto (como o de pensar e escrever testes), maiores são as chances que eles se tornem melhores programadores e testadores.

Dentre os obstáculos à adoção do TDL, encontra-se a necessidade dos alunos terem que aprender uma sintaxe adicional para representar seus casos de teste, o que pode dificultar o aprendizado de programação. Além disso, no contexto de um ambiente de programação, uma grande quantidade de código adicional é requerido para escrever um caso de teste, transformando o problema de escrever casos de teste em um problema muito mais complexo envolvendo a criação de uma classe contendo uma biblioteca de métodos de teste.

2.3.2 Ambientes e Ferramentas de Apoio

Outra iniciativa investigada a fim de explorar o ensino integrado de fundamentos de programação e teste de software refere-se ao desenvolvimento de ambientes e ferramentas de apoio. Tais ambientes estão sendo desenvolvidos em duas perspectivas diferentes. Uma delas é a construção de ambientes de programação pedagógicos que facilitem a construção de casos de teste apoiando a realização da atividade de teste de unidade. Dentre tais ambientes, destacam-se o DRJAVA (Allen et al., 2002) e o BLUEJ (Patterson et al., 2003). Esses ambientes apoiam o ensino de OO e facilitam a construção de casos de teste, podendo ser utilizados pelos alunos para auxiliar o ensino integrado de fundamentos de programação e de teste de software.

Ressalta-se, no entanto, que tanto o DRJAVA como o BLUEJ não são integrados a ferramentas de teste e, consequentemente, não apoiam a aplicação de técnicas e critérios de teste.

Outra perspectiva refere-se à construção de ambientes que forneçam suporte para submissão e avaliação automática dos programas desenvolvidos pelos alunos, a fim de que os mesmos possam ter um julgamento imediato da sua implementação, mostrando se o programa está ou não correto. Dentre tais ambientes, pode-se destacar: BOSS/BOSS2 (Joy et al., 2005), ONLINE JUDGE (Kurnia et al., 2001), COURSEMARKER (Higgins et al., 2003), MARMOSET (Spacco et al., 2006a,b), OTO (Tremblay et al., 2008) e WEB-CAT (Edwards, 2004).

Apesar de todos esses ambientes permitirem que os alunos submetam seus programas para a avaliação automática, somente na WEB-CAT e na MARMOSET os alunos podem, adicionalmente, submeterem os seus conjuntos de teste para serem avaliados. Neste sentido, tais ambientes são brevemente detalhados a seguir.

2.3.2.1 WEB-CAT

WEB-CAT é um ambiente Web que visa incentivar o desenvolvimento dirigido por testes, apoiando a submissão e avaliação automática de trabalhos de programação (Edwards e Perez-Quinones, 2008). Os alunos devem submeter seus programas juntamente com os casos de teste e o ambiente fornece *feedback* sobre a qualidade e a completude do teste, bem como a correção do código. O ambiente foi desenvolvido na *University Virginia Tech*. A Figura 2.1 ilustra a página principal da WEB-CAT.

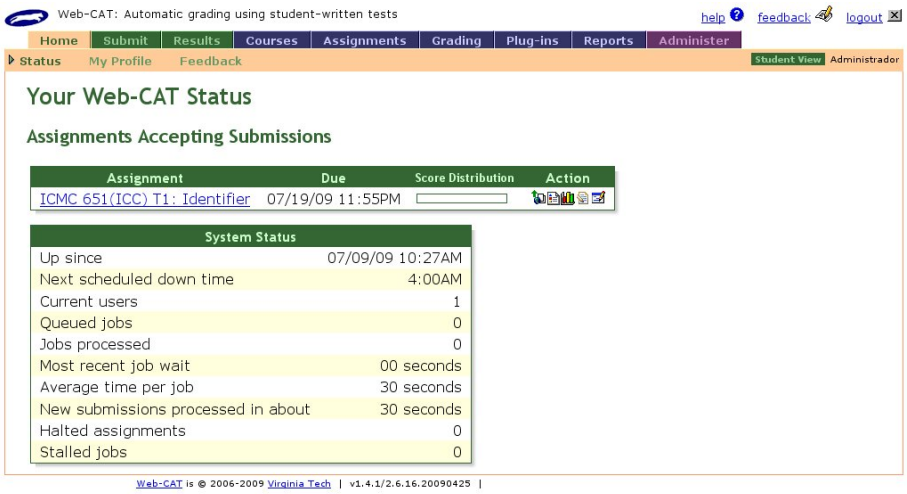


Figura 2.1: O Ambiente WEB-CAT

Basicamente, o ambiente WEB-CAT avalia as soluções do aluno de acordo com três parâmetros: (1) legibilidade/projeto, realizada manualmente por assistentes de professor; (2) estilo/codificação, utilizando ferramentas de análise estática, e (3) teste/correção, utilizando ferramentas de teste. O "peso" de cada parâmetro na avaliação é definido pelo professor. A nota final dada à solução do aluno consiste na soma dos pontos adquiridos para cada parâmetro.

As ferramentas de teste e de análise estática utilizadas nas avaliações são fornecidas por *plugins*. Vários *plugins* podem ser utilizados em um mesmo trabalho. Cada *plugin* tem um esquema de avaliação e relatórios associados para uma linguagem de programação particular. Atualmente, existem *plugins* para Java, C, C++ e Pascal. Além disso, outros *plugins* podem ser desenvolvidos com base nos existentes.

O *plugin JavaTddPlugin*, por exemplo, realiza a avaliação dos programas escritos em linguagem Java. Ele utiliza as ferramentas CHECKSTYLE[23] e *PMD*[24] para a realização da analisa estática e a ferramenta CLOVER para a realização dos testes. Para a atividade de teste, o *plugin* verifica: (1) quantos testes do aluno passaram; (2) quanto do código do aluno é exercitado pelos seus casos de teste; e (3) quanto do problema a solução do aluno cobre; cuja medida é obtida comparando a quantidade de falhas identificadas pelos casos de teste do aluno com a quantidade de falhas identificadas por casos de teste submetidos pelo professor.

A WEB-CAT também permite que o instrutor configure os prazos para submissão de trabalhos, o número de submissões que podem ser realizadas por cada aluno, o tamanho máximo de arquivo permitido, as penalidades por atraso, entre outros. Estas configurações consistem em uma regra de submissão. Regras de submissão podem ser reutilizadas, ou seja, uma mesma regra de submissão pode ser aplicada em diferentes trabalhos. A ideia de reutilização é também explorada nos trabalhos, na qual o professor pode aplicar a um curso um trabalho definido para

[23] http://checkstyle.sourceforge.net
[24] http://pmd.sourceforge.net

outro curso. Da mesma forma, o professor também pode instanciar novos cursos a partir de uma definição de curso criado anteriormente.

Informações mais detalhadas sobre o ambiente WEB-CAT e os suas principais funcionalidades podem ser encontrados em `http://web-cat.cs.vt.edu/`.

2.3.2.2 MARMOSET

MARMOSET (Spacco et al., 2006a,b) é um sistema para submissão e teste de trabalhos de programação. Segundo os autores, a ferramenta se diferencia dos demais ambientes de submissão e avaliação automática de duas maneiras: (1) utilizando uma nova técnica para premiar os alunos que começam a fazer os seus trabalhos mais cedo e que pensam criticamente sobre seus programas; e (2) capturando a situação atual do projeto toda vez que o aluno salva seus arquivos e ajudando os professores e pesquisadores a entenderem os hábitos de desenvolvimento e dificuldades dos alunos.

Os casos de teste utilizados para testar os projetos na MARMOSET são de quatro tipos: (1) testes dos alunos, os quais são fornecidos pelos próprios alunos; (2) testes públicos, os quais são fornecidos aos alunos antes de iniciarem seus trabalhos; (3) testes de liberação, escritos pelo professor e que são disponibilizados aos alunos em condições específicas ou após o prazo de entrega do trabalho; e (4) testes secretos, escritos pelo professor e que são disponibilizados somente após o prazo de entrega dos trabalhos.

Após compilar e testar os projetos, os alunos têm acesso imediato aos resultados dos seus casos de teste e dos testes públicos. Se a submissão de um aluno passar em todos os testes públicos, então o aluno tem a opção de realizar um teste de liberação naquela submissão. Se um estudante requisitar o teste de liberação de uma submissão, o sistema revelará somente o número de testes de liberação que falharam e os nomes dos dois primeiros testes que falharam, conforme ilustrado na Figura 2.2.

Há um limite de quantos testes de liberação podem ser realizados. Cada teste de liberação consome uma ficha que eventualmente se regenera. Os parâmetros podem ser configurados por projeto, mas na configuração padrão os alunos têm três fichas, que são regenerados 24 horas depois de serem usados. Esta configuração permite que o aluno possa realizar três testes de liberação do seu projeto por dia.

A MARMOSET também fornece aos professores diversas informações como, por exemplo, quantos projetos passaram por cada caso de teste, uma lista das melhores submissões de cada aluno, uma lista dos alunos que ainda não submeteram nada, dentre outras. Os professores podem ainda navegar por uma submissão específica de um aluno, visualizando os detalhes de um particular caso de teste, permitindo que o mesmo identifique os conceitos que o aluno está tendo dificuldades.

Submission #3, submitted at Thu, 10 Feb at 03:34 AM

Test Results

type	test #	outcome	points	name	short result	long result
public	0	passed	1	testPlayingWithAFullDeck	PASSED	
release	1	failed	1	testFlush		
release	2	failed	1	testFourOfAKind		
release	?	failed	?	?		
release	?	failed	?	?		
release	?	failed	?	?		

You received 1/1 points for public test cases.

You received 4/9 points for release tests.

You currently have 2 release tokens available.

Release token(s) will regenerate at:

- Fri, 23 Jun at 12:11 AM

Figura 2.2: MARMOSET: Resultado de um Teste de Liberação (Spacco et al., 2006b)

Por fim, a MARMOSET compila e testa programas escritos em linguagem Java ou em linguagens que podem ser compiladas e testadas com o MAKE (Oram e Talbott, 1991), tais como C e Ruby. No entanto, para projetos em Java, a MARMOSET implementa algumas funcionalidades específicas: (1) permite que os casos de teste sejam especificados usando o JUNIT; (2) impede, por meio de um gerenciador de segurança, que os programas dos alunos realizem tarefas que possam danificar o sistema ou que tenham acesso indevido a dados do sistema; (3) permite a identificação de defeitos no código por meio da FINDBUGS (Ayewah et al., 2007; Hovemeyer e Pugh, 2007), uma ferramenta de análise estática; e (4) permite realizar a análise de cobertura dos testes no projeto por meio da utilização da ferramenta CLOVER.

2.4 Considerações Finais

Neste capítulo foram apresentados os principais conceitos e pesquisas relacionados a este trabalho. Destaque foi dado às técnicas e critérios de teste bem como às ferramentas de apoio associadas. Observa-se que as ferramentas de teste investigadas são possíveis candidatas à integração com a PROGTEST.

Além disso, foram apresentadas iniciativas investigadas a fim de apoiar o ensino de fundamentos de programação e teste de software, dentre elas a proposição de ambientes de apoio, tais como os ambientes WEB-CAT e MARMOSET. Tais ambientes foram estudados a fim de identificar caractrerísticas e funcionalidades possíveis de serem consideradas no desenvolvimento e evolução do ambiente PROGTEST.

No próximo capítulo, detalhes sobre as principais funcionalidades do ambiente PROGTEST são apresentados. Além disso, são descritos os principais aspectos relacionados à integração de novas ferramentas de teste à PROGTEST.

CAPÍTULO

3

Desenvolvimento e Evolução do Ambiente PROGTEST

3.1 Considerações Iniciais

Um dos principais objetivos deste trabalho foi a integração de novas ferramentas de teste ao ambiente PROGTEST, considerando diferentes linguagens de programação e técnicas de teste.

Na Seção 3.2, são descritas as principais características do ambiente PROGTEST, e suas funcionalidades são detalhadas.

Na Seção 3.3, são discutidos os aspectos relacionados à integração de novas ferramentas ao ambiente. De modo geral, a PROGTEST foi evoluída a fim de permitir a integração de ferramentas como *plugins*, tornando-se extensível para a integração da maioria das ferramentas de teste investigadas. *Plugins* de ferramentas para o ambiente também foram construídos. Por fim, na Seção 3.4 são descritos os aspectos relacionados à evolução do ambiente, em especial da base de trabalhos oráculo.

3.2 O Ambiente PROGTEST

Como discutido anteriormente, uma questão crítica para o sucesso do ensino integrado de fundamentos de programação e teste de software é como fornecer um *feedback* adequado e avaliar o desempenho do aluno. Neste cenário de aprendizagem, o trabalho dos professores é dupli-

cado uma vez que tanto os casos de teste como o código devem ser avaliados. Uma alternativa visando reduzir o esforço de trabalho é o uso de ferramentas automatizadas na avaliação de trabalhos práticos. Além disso, o uso de tais ferramentas pode trazer benefícios adicionais em termos de consistência, eficácia e eficiência já que todos os programas apresentados são analisados no mesmo nível de efetividade e os resultados da avaliação são baseados nas mesmas normas. Após a avaliação, também é possível a geração de relatórios, de modo que cada aluno seja informado sobre seu desempenho em relação à média e em relação aos alunos mais produtivos.

A PROGTEST insere-se nesse contexto, tendo sido desenvolvida como uma ferramenta web para submissão e avaliação automática de trabalhos de programação baseada em atividades de teste (Barbosa et al., 2008; Corte, 2006; Corte et al., 2006, 2007). A ideia é fornecer suporte automatizado para avaliar os programas e casos de teste submetidos pelos alunos. Para isso, ferramentas de teste de cobertura podem ser integradas à PROGTEST, fornecendo apoio para aplicar os critérios de teste e avaliar a cobertura dos casos de teste, obtida a partir de execução dos programas. Tanto a qualidade do código como a qualidade dos testes podem ser analisadas com base nos critérios adotados. Atualmente uma versão da PROGTEST encontra-se disponível em `http://www.labes.icmc.usp.br:9080/ProgTest`.

A Figura 3.1 mostra o funcionamento geral da PROGTEST. Dado um programa P_{St_i} (fornecido pelo aluno) e seu respectivo conjunto de casos de teste T_{St_i} (produzido com base em um critério de teste C_K previamente estabelecido (T_{St_i} é C_K-adequado), a PROGTEST, integrada a ferramentas de teste, deve ser capaz de:

1. Executar o programa P_{St_i} com o conjunto de teste T_{Inst} (fornecido pelo professor);

2. Utilizar o conjunto de teste T_{St_i} para testar a "implementação de referência" P_{Inst} (fornecida pelo professor);

3. Comparar o comportamento do P_{Inst}, executado com o conjunto de teste T_{Inst}, com o comportamento de P_{St_i}, executado com o conjunto de teste T_{Inst}; e

4. Comparar o comportamento do P_{Inst}, executado com o conjunto de teste T_{St_i}, com o comportamento de P_{St_i}, executado com o conjunto de teste T_{St_i}.

Com base nas execuções realizadas, a PROGTEST é capaz de sugerir uma nota para o trabalho do aluno. Se o trabalho submetido não estiver correto, o aluno poderá submeter novas versões do seu trabalho até atingir nota máxima.

A seguir, as principais funcionalidades do ambiente PROGTEST são detalhadas.

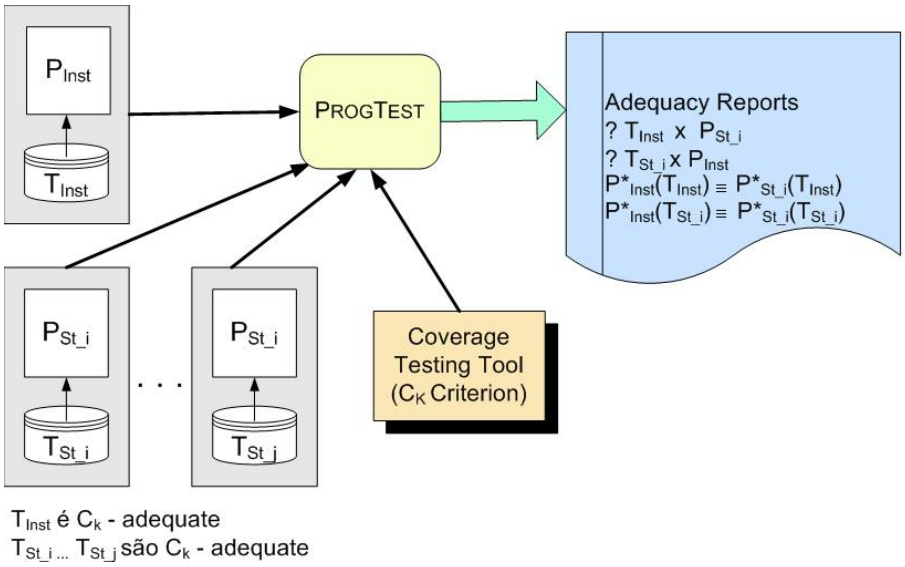


Figura 3.1: PROGTEST: Funcionamento Geral

3.2.1 PROGTEST: Principais Funcionalidades

As principais funcionalidades da PROGTEST podem ser acessadas por meio de duas visões – professor ou aluno. Em relação à visão do professor (Figura 3.2), a PROGTEST permite ao usuário criar novos cursos, matricular alunos nos cursos e definir trabalhos de programação.

Figura 3.2: PROGTEST: Visão do Professor

Para avaliar os trabalhos de programação dos alunos, o ambiente requer um "trabalho oráculo", o qual consiste em: (1) um programa de referência fornecido pelo professor, que imple-

menta a solução correta para o trabalho proposto; e (2) um conjunto de teste para o programa fornecido, sendo que este deve ser 100%-adequado aos critérios de teste considerados na avaliação.

Existem duas maneiras do professor fornecer um trabalho oráculo: (1) submetendo seu próprio programa e conjunto de teste; ou (2) selecionando um trabalho a partir da base de trabalhos oráculo disponível na PROGTEST.

A base de trabalhos é composta por um conjunto de programas e seus respectivos conjuntos de teste, selecionados a partir de livros e exercícios utilizados em cursos introdutórios de fundamentos de programação (Ziviani, 2005). A ideia é facilitar o trabalho do professor, que pode usar os trabalhos oráculo disponíveis na base da PROGTEST ao invés de definir, implementar e testar o seu próprio programa.

Para criar um trabalho prático o professor deve associar a ele um título, uma descrição, uma data inicial e uma data limite para entrega. Ressalta-se, entretanto, que a PROGTEST não impede os alunos de submeterem seus trabalhos após o prazo e nem calcula penalizações automaticamente. O objetivo, de fato, é incentivar os alunos a realizarem suas atividades. Nesse sentido, a PROGTEST apenas registra o atraso, sendo que eventuais penalizações ficam a cargo do professor.

Ao criar um trabalho prático o professor deve associar a ele um título, descrição e datas limite para início e término das submissões; bem como definir quais critérios de teste deverão ser utilizados durante a avaliação. Para cada critério selecionado, o professor também deve definir qual o peso que este terá durante a avaliação. Pesos mais altos para critérios mais rigorosos resultarão em uma avaliação mais rigorosa, enquanto pesos mais altos para critérios mais fracos resultam em uma avaliação menos rigorosa.

Definidas as propriedades do trabalho prático, a PROGTEST está pronta para realizar as atividades de teste. Inicialmente, a PROGTEST considera o trabalho oráculo, compilando o programa do professor e executando seus casos de teste. Com base nos critérios e pesos definidos pelo professor, a PROGTEST calcula uma cobertura total do trabalho oráculo ($P_{Inst} - T_{Inst}$).

Considerando a visão do aluno (Figura 3.3), este tem acesso a todos os trabalhos associados aos cursos em que ele se encontra matriculado. Para submeter uma solução para um dado trabalho, o aluno deve enviar seu programa e o conjunto de teste a ser utilizado para testá-lo. A PROGTEST compila o programa do aluno e calcula a cobertura para as seguintes combinações: (i) programa do aluno com o conjunto de teste do aluno ($P_{St_i} - T_{St_i}$); (ii) programa do professor com o conjunto de teste do aluno ($P_{Inst} - T_{St_i}$); e (iii) programa do aluno com o conjunto de teste do professor ($P_{St_i} - T_{Inst}$). Ao realizar essas execuções, a PROGTEST é capaz de sugerir uma nota para o trabalho submetido.

Para avaliar a adequação dos conjuntos de teste em relação a critérios de teste associados, ferramentas de teste estão integradas ao ambiente. Atualmente, encontram-se integradas: (1) as

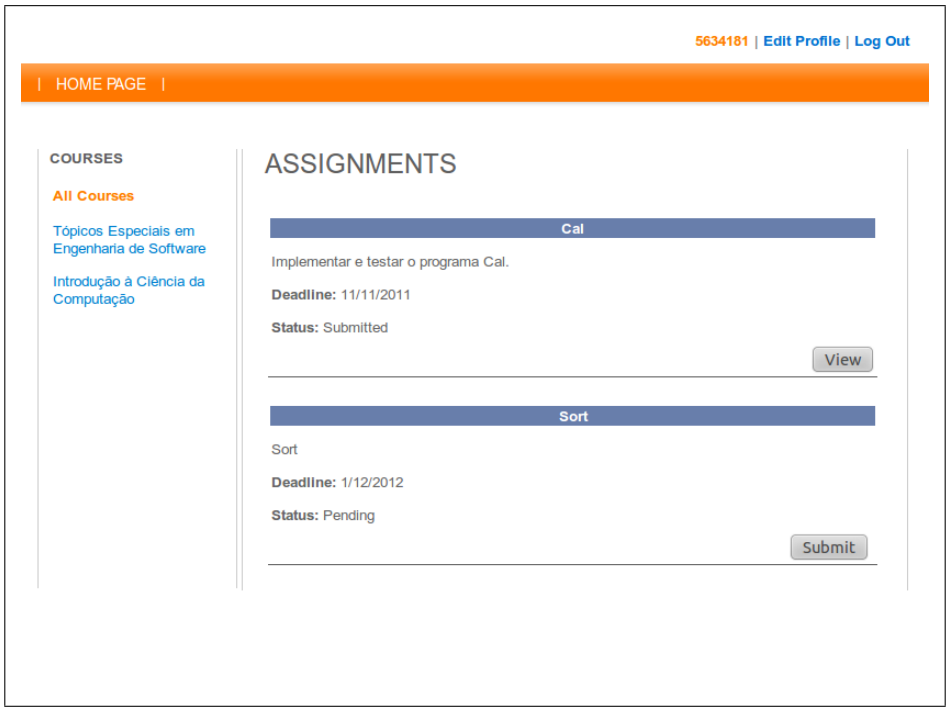


Figura 3.3: PROGTEST: Visão do Aluno

ferramentas JUNIT e JABUTISERVICE, que já se encontravam integradas à PROGTEST antes do desenvolvimento deste projeto; e (2) as ferramentas JUMBLE, CUNIT, GCOV e PROTEUM, cuja integração à PROGTEST constitui em parte do resultado deste trabalho.

Logo após ter submetido seu trabalho, o aluno pode visualizar o relatório de avaliação, conforme ilustrado na Figura 3.4, além de outros relatórios fornecidos pelas ferramentas integradas. Se o trabalho submetido não estiver correto, o aluno poderá submeter novas versões do seu trabalho até atingir nota máxima.

Além do relatório de avaliação, outros relatórios fornecidos pela PROGTEST e os relatórios fornecidos pelas ferramentas de teste são disponibilizados tanto para alunos como para professores.

3.2.2 PROGTEST: Comparação com Ambientes Similares

Vários ambientes de apoio à submissão e avaliação automática de trabalhos de programação têm sido desenvolvidos. Basicamente, em tais ambientes a correção do programa e a adequação dos conjuntos de teste são avaliados utilizando três parâmetros: (1) a taxa de sucesso dos casos de teste dos alunos contra os seus próprios programas; (2) a taxa de sucesso dos casos de teste do professor contra o programa dos alunos; e (3) a cobertura de código dos casos de teste dos alunos em seus programas. A WEB-CAT, em particular, também compara a taxa de sucesso dos casos de teste do professor com a taxa de sucesso dos casos de teste dos alunos, calculando

GENERAL COVERAGES

Criteria	Pinst-Tinst	Pst-Tst	Pinst-Tst	Pst-Tinst
JUnit/Pass Rate	100.0%	91.67%	50.0%	93.88%
JaBUTiService/All-Nodes-ei	100.0%	82.76%	95.12%	91.38%
JaBUTiService/All-Nodes-ed	100.0%	100.0%	100.0%	100.0%
JaBUTiService/All-Edges-ei	100.0%	77.94%	93.18%	92.65%
JaBUTiService/All-Edges-ed	100.0%	100.0%	100.0%	100.0%
JaBUTiService/All-Uses-ei	100.0%	71.01%	96.49%	81.16%
JaBUTiService/All-Uses-ed	100.0%	100.0%	100.0%	100.0%

TOTAL COVERAGES

Execution	Value
Instructor's Tests against Instructor's Program (Pinst-Tinst)	100.0%
Student's Tests against Student's Program (Pst-Tst)	90.14%
Student's Tests against Instructor's Program (Pinst-Tst)	73.73%
Instructor's Tests against Student's Program (Pst-Tinst)	94.04%

SUGGESTED GRADE: 8.6

Figura 3.4: PROGTEST: Resultados de uma Avaliação

a cobertura do problema. Cada falha revelada pelo conjunto de teste do professor e que não é revelada pelo conjunto de teste do aluno faz com que a cobertura do problema diminua.

Considerando, como exemplo, o problema de calcular o fatorial de um número, uma possível solução é implementada no `Programa 1`, exibido na Figura 3.5 (a). O `Conjunto de Teste 1`, exibido na Figura 3.5 (b), para o `Programa 1`, é 100%-adequado aos critérios Todos-Nós e Todas-Arestas.

```
1  public class Factorial {
2
3      public static int factorial(int x) {
4          if(x == 0 || x == 1)
5              return 1;
6          else
7              return x*factorial(x-1);
8      }
9
10 }
```

(a) `Programa 1`

```
5  public class FactorialTest {
6
7      @Test
8      public void test1() {
9          assertEquals(1, Factorial.factorial(0));
10     }
11
12     @Test
13     public void test2() {
14         assertEquals(1, Factorial.factorial(1));
15     }
16
17     @Test
18     public void test3() {
19         assertEquals(2, Factorial.factorial(2));
20     }
21
22     @Test
23     public void test4() {
24         assertEquals(120, Factorial.factorial(5));
25     }
```

(b) `Conjunto de Teste 1`

Figura 3.5: Trabalho `Fatorial 1`.

Tomando o `Conjunto de Teste 1` como o conjunto de teste do professor e tomando o `Programa 1` e o `Conjunto de Teste 1` como programa e conjunto de teste do aluno, temos um cenário em que o aluno implementa a solução correta do programa e realiza uma atividade de teste apropriada. Assim, submetendo o seu trabalho em um ambiente de avaliação automática o aluno deve receber nota máxima.

Considerando os resultados fornecidos pela WEB-CAT para a correção e adequação dos casos de teste do aluno, a taxa de sucesso dos casos de teste do aluno e a cobertura de código são avaliadas em 100%. Uma vez que os casos de teste do professor não causam falhas, a cobertura do problema também é avaliada em 100%.

Um resultado similar é obtido na MARMOSET. Como o conjunto de teste do aluno e o conjunto de teste de professor não causam falhas no programa do aluno, a taxa de sucesso dos casos de teste de ambos conjuntos de teste são 100%. Além disso, tanto a WEB-CAT como a MARMOSET usam a ferramenta Clover para analisar a cobertura de código. Por esse motivo, a cobertura de código na MARMOSET também é de 100%.

Considerando a PROGTEST, além do conjunto de teste, o professor também deve fornecer um programa de referência, que implementa uma solução correta para o problema. Então, considerando o `Programa 1` como o programa fornecido pelo professor, o aluno também obtém nota máxima quando submete seu programa na PROGTEST. Como ambos conjunto de teste e programa do professor e do aluno estão corretos, a taxa de sucesso dos conjuntos de teste são de 100% em todas as execuções. Ainda, configurando a PROGTEST para utilizar os mesmos critérios apoiados pela CLOVER (Todos-Nós e Todas-Arestas), a cobertura de código também é de 100% para ambos critérios de teste.

Um aspecto crítico relacionado ao problema de calcular o fatorial de um número é o cálculo do fatorial de zero, que deve resultar em 1. O código da Figura 3.6 (a) mostra o `Programa 2`, em que tal condição não foi implementada. Da mesma forma, o `Conjunto de Teste 2`, exibido na Figura 3.6 (b), não considera o valor zero como dado de entrada em nenhum dos casos de teste. Além disso, o `Conjunto de Teste 2` não cobre todos os nós (blocos de comandos) e arestas (desvios de fluxo) do `Programa 2`.

Submetendo o `Programa 2` e o `Conjunto de Teste 2` como trabalho do aluno, WEB-CAT identifica o problema, uma vez que no conjunto de teste do professor (`Conjunto de Teste 1`) há um caso de teste que executa o programa do aluno com o valor zero, causando uma falha. Por esse motivo, a cobertura do problema avaliada pela WEB-CAT é de 75% – um dos quatro conjuntos de teste identificou um problema que o conjunto de teste do aluno não identificou.

De forma semelhante, a MARMOSET fornece uma taxa de sucesso de 100% para o conjunto de teste do aluno. No entanto, quando o aluno solicita um teste de liberação, o ambiente exibe uma taxa de sucesso de 75% para o conjunto de teste do professor. Ressalta-se, que tanto

```java
public class Factorial {

    public static int factorial(int x) {
        if(x == 1)
            return 1;
        else
            return x*factorial(x-1);
    }

}
```

(a) `Programa 2`

```java
public class FactorialTest {

    @Test
    public void test1() {
        assertEquals(1, Factorial.factorial(1));
    }

    @Test
    public void test2() {
        assertEquals(2, Factorial.factorial(2));
    }

    @Test
    public void test3() {
        assertEquals(120, Factorial.factorial(5));
    }

}
```

(b) `Conjunto de Teste 2`

Figura 3.6: Trabalho `Fatorial 2`.

na WEB-CAT como na MARMOSET o defeito no programa do aluno só é revelado porque o conjunto de teste do professor tem um caso de teste específico que detecta este tipo de erro.

A PROGTEST também identifica o problema no programa do aluno – a taxa de sucesso do conjunto de teste do professor contra o programa do aluno é de 75%. Além disso, quando o conjunto de teste do aluno é executado contra o programa do professor, o aluno é informado que a cobertura para o critério Todas-Arestas é de 80%.

Por fim, suponha que o aluno implemente o `Programa 3`, ilustrado na Figura 3.7 (a), e projete o CONJUNTO DE TESTE 3, exibido na Figura 3.7 (b). Para números acima de 60, o programa calcula o fatorial incorretamente. O CONJUNTO DE TESTE 3, para o `Programa 1`, é 100%-adequado aos critérios Todos-Nós e Todas-Arestas. No entanto, ele não falha quando executado contra o `Programa 3` com valores acima de 60.

```java
public class Factorial {

    public static int factorial(int x) {
        if(x == 0 || x == 1)
            return 1;
        else if(x == 60)
            return 2;
        else
            return x*factorial(x-1);
    }

}
```

(a) `Programa 3`

```java
public class FactorialTest {

    @Test
    public void test1() {
        assertEquals(1, Factorial.factorial(0));
    }

    @Test
    public void test2() {
        assertEquals(1, Factorial.factorial(1));
    }

    @Test
    public void test3() {
        assertEquals(2, Factorial.factorial(60));
    }

    @Test
    public void test4() {
        assertEquals(120, Factorial.factorial(5));
    }

}
```

(b) `Conjunto de Teste 3`

Figura 3.7: Trabalho `Fatorial 3`.

Submetendo o `Programa 3` com o `Conjunto de Teste 3` na WEB-CAT, o aluno obtém nota máxima. Neste caso, o conjunto de teste do professor (`Conjunto de Teste`

`1`) não considera que o aluno possa cometer tal engano. Isso mostra que, na WEB-CAT, para que um determinado defeito no programa do aluno seja identificado, o professor deve ter implementado um caso de teste que encontre tal defeito.

O mesmo comportamento é observado na MARMOSET. Uma vez que tanto o conjunto de teste do aluno como o conjunto de teste do professor não causam falhas, a taxa de sucesso dos casos de teste é de 100%.

A PROGTEST difere da WEB-CAT e da MARMOSET por utilizar uma implementação de referência fornecida pelo professor, além do conjunto de teste fornecido por ele. A Figura 3.8 mostra o resultado da submissão do `Programa 3` com o `Conjunto de Teste 3` na PROGTEST. Quando o conjunto de teste do aluno (`Conjunto de Teste 1`) é executado contra o programa correto (`Programa 1`), o caso de teste incorreto do aluno no `Conjunto de Teste 3` causa uma falha, evidenciando seu defeito. Também, analisando a cobertura para os critérios Todos-Nós e Todas-Arestas, é possível observar que, para o programa do aluno, o conjunto de teste do professor não é 100%-adequado para os critérios Todos-Nós e Todas-Arestas. Uma vez que partes do programa do aluno não foram executadas pelo conjunto de teste do professor, tais partes podem conter defeitos.

EVALUATION RESULT

GENERAL COVERAGES

Criteria	P_Inst - T_Inst	P_St - T_St	P_Inst - T_St	P_St - T_Inst
JUnit/Pass Rate	100.0%	100.0%	80.0%	100.0%
JaBUTiService/All-Nodes-ei	100.0%	100.0%	100.0%	87.0%
JaBUTiService/All-Edges-ei	100.0%	100.0%	100.0%	85.0%

TOTAL COVERAGES

Executions	Coverage
Instructor's test set against instructor's program (P_Inst - T_Inst)	100.0%
Student's test set against student's program (P_St - T_St)	100.0%
Student's test set against instructor's program (P_Inst - T_St)	90.0%
Instructor's test set against student's program (P_St - T_Inst)	93.0%

SUGGESTED GRADE: 9.43

Figura 3.8: Resultados da Avaliação do `Trabalho 3`.

Comparando com a WEB-CAT e a MARMOSET, somente a PROGTEST realiza a avaliação utilizando uma implementação de referência fornecida pelo professor. Essa funcionalidade é importante para automaticamente assegurar que a solução submetida pelo aluno corresponde ao problema proposto. Na WEB-CAT e na MARMOSET o professor pode opcionalmente fornecer um conjunto de teste para ser confrontado com o programa do aluno, mas não pode fornecer o seu próprio programa.

Por fim, a Tabela 3.1 apresenta uma comparação entre a PROGTEST e os ambientes WEB-CAT e MARMOSET. Dentre as características e funcionalidades que diferenciam a PROGTEST, encontram-se: (1) a avaliação realizada a partir de uma implementação de referência fornecida pelo professor; (2) a utilização de critérios de teste mais "fortes", como critérios de fluxo de dados e critérios baseados em erros; e (3) a possibilidade de definir quais critérios de teste serão considerados e o peso que cada um deles terá na avaliação.

Tabela 3.1: Comparação entre PROGTEST, MARMOSET e WEB-CAT

	PROGTEST	MARMOST	WEB-CAT
Teste funcional	X	X	X
Teste estrutural (fluxo de controle)	X	X	X
Teste estrutural (fluxo de dados)	X		
Teste baseado em erros	X		
Análise estática		X	X
Permite escolha de critérios de teste	X		
Permite definição de pesos para critérios de teste	X		
Professor provê uma implementação de referência	X		
Possui uma base de trabalhos	X		

3.2.3 PROGTEST: Arquitetura

A arquitetura do ambiente PROGTEST (Figura 3.9) foi desenvolvida segundo o padrão Model-View-Controller (MVC) (Fowler, 2002), o qual consiste em: (1) uma camada de visão, responsável pela interação com o usuário; (2) uma camada modelo, contendo as regras de negócio; e (3) uma camada de controle, responsável por intermediar os dados entre as outras duas camadas.

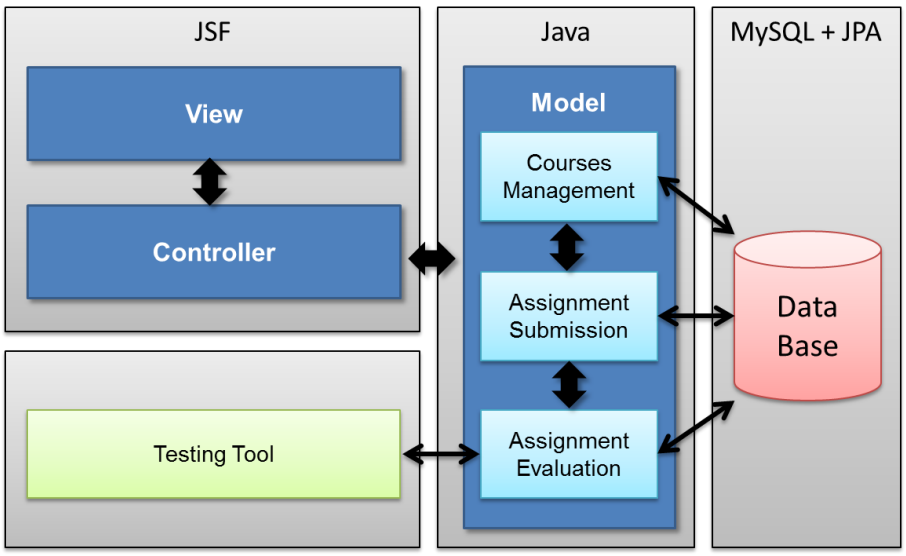


Figura 3.9: PROGTEST: Arquitetura

A camada de modelo é composta basicamente por três módulos principais: (1) módulo de gerenciamento de cursos, responsável pelo gerenciamento de cursos, trabalhos, alunos e acesso de usuários; (2) módulo de submissão, que realiza o controle da submissão dos trabalhos, assim como, o controle das datas de início e término dos cursos e prazos para envio dos trabalhos; e (3) módulo de avaliação, que avalia os trabalhos submetidos e calcula uma nota para o mesmo. Os três módulos possuem acesso a um banco de dados, onde são armazenados os dados utilizados pelo ambiente. Além disso, o módulo de avaliação é integrado às ferramentas de teste, que avaliam a adequação dos conjuntos de teste em relação a critérios de teste associados.

3.2.4 PROGTEST: Aspectos de Desenvolvimento

O ambiente PROGTEST foi inicialmente proposto no trabalho de Corte (Barbosa et al., 2008; Corte, 2006; Corte et al., 2006, 2007). No entanto, durante a graduação, o autor deste trabalho realizou duas iniciações cientificas que deram continuidade ao desenvolvimento e evolução do ambiente PROGTEST.

O primeiro trabalho de iniciação cientifica consistiu no projeto e implementação do módulo de gerenciamento de cursos do ambiente PROGTEST, uma vez que, no estágio anterior de desenvolvimento da PROGTEST, encontravam-se implementados apenas os módulos de submissão e avaliação. Além disso, os seguintes aspectos foram investigados durante o trabalho: (1) operações relacionadas ao acesso e controle do banco de dados; (2) desenvolvimento e avaliação da interface gráfica; (3) melhorias na integração da ferramenta de teste JABUTI à PROGTEST.

Já o segundo trabalho, realizado com apoio financeiro da FAPESP (Processo 09/00006-3), consistiu na comparação da PROGTEST a ambientes relacionados e sua evolução em função de características relevantes observadas.

Em relação à evolução do ambiente PROGTEST, ênfase foi dada a integração com a ferramenta JABUTISERVICE. Tal integração foi explorada afim de disponibilizar, aos usuários da PROGTEST, relatórios e informações providas pelas ferramentas de teste e que anteriormente não eram possíveis de serem acessados via PROGTEST.

Outras atividades realizadas foram o desenvolvimento da base de trabalhos oráculo e a realização de uma validação preliminar do ambiente PROGTEST.

Em continuação às atividades realizadas, neste trabalho, ênfase é dada a integração de diferentes ferramentas de teste ao ambiente PROGTEST, visando explorar outros critérios de teste bem como diferentes linguagens de programação. Tal integração é detalhada nas seções seguintes.

3.3 Integração de Ferramentas ao Ambiente PROGTEST

3.3.1 Análise das Ferramentas de Teste

Uma das primeiras atividades realizadas a fim de conduzir a integração de novas ferramentas de teste ao ambiente PROGTEST foi a análise de ferramentas de teste. A análise foi realizada a fim de identificar as principais características e diferenças entre as ferramentas que poderiam ser consideradas durante a integração.

Dentre as características observadas, destacam-se: (1) o artefato utilizado pelas ferramentas para derivar os requisitos de teste; (2) os mecanismos pelos quais as ferramentas podem ser acessadas; e (3) as formas pelas quais os relatórios são apresentados. Tais características são discutidas a seguir.

Artefato para Derivação dos Requisitos de Teste

Uma das características observadas nas ferramentas de teste refere-se ao artefato utilizado para derivar os requisitos de teste. De modo geral, enquanto algumas ferramentas derivam os requisitos de teste a partir do código fonte do programa, outras derivam tais requisitos a partir de um código intermediário. Exemplos de código intermediário são o Java *bytecode* e o código objeto resultante da compilação de um programa escrito em linguagem C/C++.

Tal característica foi um fator importante a ser considerado durante a integração de ferramentas ao ambiente, uma vez que a PROGTEST deve ser capaz de fornecer o código que cada ferramenta solicitar.

Além disso, a ordem em que as operações são realizadas varia de acordo com cada ferramenta. Nas ferramentas que utilizam um código intermediário, a derivação dos requisitos de teste ocorre somente depois da compilação do programa. Por outro lado, com as ferramentas que utilizam o código fonte, os requisitos são derivados antes do programa ser compilado.

Mecanismos de Acesso

Outra característica importante observada refere-se aos mecanismos pelos quais as ferramentas podem ser acessadas. É por meio desses mecanismos que a PROGTEST pode acessar as funcionalidades das ferramentas.

Basicamente, o acesso às ferramentas de teste pode ser realizado por meio de: (1) interfaces com o usuário, como por exemplo uma interface por linha de comando ou uma interface gráfica; (2) ferramentas de automatização de *build*, como por exemplo o ANT e o MAVEN; (3) ambientes de desenvolvimento integrado, como o ECLIPSE e o IDEA; e (4) bibliotecas, como as fornecidas pelo JUNIT e os *Stubs* utilizados para acessar ferramentas em um *web service*.

A Tabela 3.2 mostra os mecanismos pelos quais as ferramentas de teste investigadas neste trabalho são acessadas. De fato, dependendo dos mecanismos de acesso de cada ferramenta, sua integração com a PROGTEST torna-se mais ou menos viável. Em síntese, ferramentas que podem ser executadas com o ANT são bastante viáveis de se integrar com a PROGTEST. As que possuem interface por linha de comando também podem ser integradas sem maiores esforços. No entanto, ferramentas que só podem ser executadas por meio de uma interface gráfica, por exemplo, tornam a integração mais complexa e, muitas vezes, inviável.

Ferramentas	Interface com Usuário		IDE		Ferramenta de Automatização de *Build*				Bibliotecas
	Interface Gráfica	Linha de Comando	Eclipse	IDEA	Makefile	Ant	Maven 1	Maven 2	
JUnit		X	X			X			X
CUnit		X							
Clover		X	X	X		X	X	X	
Cobertura		X				X	X	X	
EMMA		X			X	X			
Gcov e Lcov		X							
GroboCodeCoverage						X			
Hansel		X	X			X			
JaBUTi	X	X							
JaBUTiService									X
Poke-Tool	X	X							
Jester		X							
Jumble		X	X						
muJava	X								
Proteum	X	X							

Tabela 3.2: Análise das Ferramentas de Teste: Mecanismos de Acesso

Apresentação de Relatórios

Outra característica relevante observada nas ferramentas de teste refere-se às formas pelas quais os relatórios são apresentados. A partir de relatórios, a PROGTEST obtém as informações sobre os resultados dos casos de teste e da adequação do conjunto de teste aos critérios apoiados pelas ferramentas. Essas informações são utilizadas para sugerir a nota do aluno.

A Tabela 3.3 mostra as formas de apresentação de relatórios de cada ferramenta. As ferramentas de linha de comando permitem a visualização dos dados de cobertura pelo console. Da mesma forma, as ferramentas com interface gráfica ou que são integradas a um ambiente de programação permitem a visualização dos dados por meio de suas interfaces gráficas ou por meio das interfaces gráficas dos ambientes aos quais estão integradas. Algumas ferramentas, como o JUNIT permitem obter dados por meio de estruturas de dados retornadas por métodos implementados nas bibliotecas fornecidas por elas. Além disso, ferramentas em um *web service*, como a JABUTISERVICE, também têm como saída variáveis e estruturas de dados retornadas pelos *Stubs*. Por fim, a maioria das ferramentas também permite que relatórios possam ser gerados ou exportados em diversos formatos de arquivo.

Tabela 3.3: Análise das Ferramentas de Teste: Apresentação de Relatórios

Ferramentas	Interface Gráfica	Console	Texto Simples	Páginas HTML	Arquivo XML	Arquivo PDF	Arquivo CSV	Estrutura de Dados
JUnit	X	X			X			X
CUnit		X			X			
Clover	X	X		X	X	X		
Cobertura				X	X			
EMMA			X	X	X			
Gcov e Lcov		X	X	X				
GroboCodeCoverage				X	X			
Hansel	X	X						
JaBUTi	X			X				
JaBUTiService								X
Poke-Tool	X	X	X					
Jester		X			X			
Jumble		X						
muJava	X							
Proteum	X	X		X	X			

Assim como os mecanismos de acesso, as formas de apresentação de relatórios interferem na viabilidade de integração. Receber as informações em uma estrutura de dados, por exemplo, pode ser a estratégia menos complexa. Para obter as informações a partir de relatórios em arquivos estruturados, como XML e CSV, pouco esforço com implementação é exigido. Ler um relatório em um arquivo de texto simples e armazenar as informações relevantes também não é uma tarefa complicada. Já em relação a ferramentas que somente exibem os resultados no console, a obtenção das informações pode ser um pouco mais complexa. Uma estratégia seria redirecionar a saída do console para um arquivo de texto. Por fim, obter informações de relatórios exibidos em uma interface gráfica seria a tarefa com maior complexidade.

3.3.2 Ferramentas como *Plugins*

Em sua versão anterior (antes da realização deste trabalho), a PROGTEST possuía duas ferramentas integradas a ela: o *framework* JUNIT e a JABUTISERVICE. Com o intuito de integrar outras ferramentas, foi feita uma análise do ambiente e percebeu-se que a PROGTEST não era extensível o suficiente para que a maioria das ferramentas investigadas pudessem ser integradas à ela. Além disso, o código responsável por realizar a execução do JUNIT e da ferramenta JABUTISERVICE estavam bastante acoplados ao código da PROGTEST e era específico para cada uma destas ferramentas, contrariando a ideia de um ambiente em que diversas ferramentas de teste pudessem ser integradas.

Neste sentido, antes de realizar a integração de outras ferramentas, foi necessário estabelecer um meio de comunicação padrão entre o ambiente PROGTEST e as ferramentas integradas. A ideia foi permitir que a execução das ferramentas ocorresse de maneira isolada da PROGTEST,

mantendo apenas uma interface padrão para troca de informações. Observa-se, no entanto, que estabelecer um meio de comunicação padrão entre a PROGTEST e as ferramentas de teste não é uma tarefa trivial. Conforme discutido anteriormente, cada ferramenta de teste possui características diferentes das demais.

A fim de minimizar os problemas identificados, foi estabelecida a integração das ferramentas de teste de forma a permitir sua operação como *plugins* dentro do ambiente PROGTEST. Assim, uma interface de comunicação entre a PROGTEST e os *plugins* foi definida e um código adicional foi implementado para cada ferramenta de teste a ser integrada a fim de adequar a interface da ferramenta em questão à essa interface de comunicação estabelecida.

Basicamente, cada *plugin* deve conter um arquivo de configuração, como ilustrado na Figura 3.10. O arquivo fornece à PROGTEST informações sobre o *plugin* e a ferramenta de teste associada à ele. Na linha 2, por exemplo, são especificados o nome da ferramenta, a linguagem que apoia, o compilador utilizado e o formato dos casos de teste.

```
1  <progtest-addon>
2      <tool name="JaBUTiService" language="Java" compiler="Javac" test-↘
       →format="JUnit 4" cmdfile="cmdlines.txt" outfile="output.txt">
3          <criterion id="1" name="All-Nodes-ei" key="all.nodes.ei"/>
4          <criterion id="2" name="All-Nodes-ed" key="all.nodes.ed"/>
5          <criterion id="3" name="All-Edges-ei" key="all.edges.ei"/>
6          <criterion id="4" name="All-Edges-ed" key="all.edges.ed"/>
7          <criterion id="5" name="All-Uses-ei" key="all.uses.ei"/>
8          <criterion id="6" name="All-Uses-ed" key="all.uses.ed"/>
9          <criterion id="7" name="All-Potencial-Uses-ei" key="all.↘
           →potencial.uses.ei"/>
10         <criterion id="8" name="All-Potencial-Uses-ed" key="all.↘
           →potencial.uses.ed"/>
11     </tool>
12 </progtest-addon>
```

Figura 3.10: *Plugin*: Arquivo de Configuração

Ainda na linha 2, são definidos dois arquivos: (1) `cmdfile`, com instruções para PROGTEST de como o *plugin* deve ser executado; e (2) `outfile`, o qual, após a execução do *plugin*, é disponibilizado à PROGTEST com os resultados da análise de cobertura. Além do `outfile`, relatórios XML também podem ser fornecidos pelos *plugins* e carregados pela PROGTEST para serem exibidos aos usuários.

Por fim, nas linhas de 3 à 10, são especificados os critérios apoiados pela ferramenta associada ao *plugin*. No exemplo, 8 critérios estão sendo especificados.

3.3.3 Construção de *Plugins*

A Tabela 3.4 mostra as ferramentas selecionadas para integração ao ambiente PROGTEST. A escolha priorizou em manter as ferramentas que já se encontravam integradas ao ambiente, a

introdução de outras linguagens de programação (em especial, a linguagem C) e a introdução de critérios baseados em erros (em especial, o critério Análise de Mutantes).

Os critérios de teste que cada ferramenta apoia podem ser observados na Tabela 3.5. Ressalta-se que as ferramentas JUNIT e CUNIT não apoiam critérios de teste, apenas permitem avaliar, para um conjunto de teste, a porcentagem de casos de teste que foram bem sucedidos, ou seja, que não falharam ao serem executados em um determinado programa.

Tabela 3.4: Ferramentas de Teste Integradas ao Ambiente PROGTEST

	Java	**C**
Frameworks de Teste de Unidade	JUNIT	CUNIT
Ferramentas de Teste Estrutural	JABUTISERVICE	GCOV
Ferramentas de Teste Baseado em Erros	JUMBLE	PROTEUM

Tabela 3.5: Ferramentas de Teste Integradas ao Ambiente PROGTEST- Critérios de Teste

	JUNIT	CUNIT	JABUTISERVICE	GCOV	JUMBLE	PROTEUM
Taxa de Sucesso[1]	X	X				
Todos-Nós			X	X		
Todas-Arestas			X	X		
Todos-Usos			X			
Todos-Potenciais-Usos			X			
Análise de Mutantes					X	X

Assim, um *plugin* de cada ferramenta foi construído para a PROGTEST. Os principais aspectos relacionados a construção dos *plugins* são discutidos a seguir.

Plugin JUNIT

O JUNIT é uma das ferramentas que já se encontravam integradas ao ambiente PROGTEST. Neste sentido, sua integração foi evoluída a fim de adapta-la como um *plugin* para a PROGTEST.

O *plugin* JUNIT construído, executa as classes de teste dos trabalhos por meio do método `JUnitCore.runClasses(Clazz clazz)`, disponibilizado pela biblioteca do JUNIT.

A Figura 3.11 mostra a linha de código do *plugin* que executa uma classe de teste. O objeto `Result` retornado, contém as informações sobre a execução dos casos de teste, como o número de casos de teste executados e o número de casos de teste que falharam, dentre outras.

```
1 Result result = JUnitCore.runClasses(clazz);
```

Figura 3.11: Código Responsável em Executar uma Classe de Teste no JUNIT

Após a execução dos casos de teste as informações obtidas são processadas, sendo os resultados e relatórios associados enviados à PROGTEST.

[1] Avalia a porcentagem de casos de teste que não causaram falhas.

Plugin JABUTISERVICE

Assim como o JUNIT, a JABUTISERVICE também se encontrava integrada ao ambiente PROGTEST, sendo evoluída a partir do desenvolvimento de um *plugin* da ferramenta para a PROGTEST.

As funcionalidades fornecidas pela JABUTISERVICE são acessadas por meio de um conjunto de operações pré-definidas pela ferramenta. Tais operações foram definidas com um alto grau de granularidade, de maneira que, para realizar a atividade de teste, uma sequência dessas operações deve ser utilizada.

O código da Figura 3.12 é responsável em realizar uma conexão com o serviço. Se a conexão for bem sucedida um objeto *stub* é criado. Por meio do *stub* as operações definidas pelo serviço podem ser invocadas. O código da Figura 3.13, por exemplo, utiliza o *stub* para criar um projeto de teste na JABUTISERVICE.

```
1 JaBUTiService1_0Stub stub = new JaBUTiService1_0Stub(endPoint);
```

Figura 3.12: Código Responsável em Realizar uma Conexão com a JABUTISERVICE

```
1 CreateProjectResponse output = stub.createProject(input);
```

Figura 3.13: Código Responsável em Criar um Projeto na JABUTISERVICE

Plugin JUMBLE

A ferramenta JUMBLE apoia o critério baseado em erros Análise de Mutantes para programas escritos em linguagem Java. O comando da Figura 3.14 exemplifica como testar uma classe em Java com a JUMBLE. No comando, `AbstractFoo` é classe do programa que está sento testada e `FooTest1` e `FooTest2` são as classes de teste. Os parâmetros `-i`, `-k`, `-w`, `-r` e `-j`, especificam quais operadores de mutação devem ser aplicados.

```
1 java -jar jumble.jar -i -k -w -r -j AbstractFoo FooTest1 FooTest2
```

Figura 3.14: Exemplo de Comando Utilizado Para Executar a JUMBLE

Durante a execução, a JUMBLE exibe no console o *score* de mutação obtido para a classe e informações sobre os mutantes que não foram mortos pelo conjunto de teste.

Para executar a JUMBLE, o *plugin* utiliza o método `Runtime.getRuntime().exec(String args[])`, fornecido pela linguagem Java. Por meio do método é possível executar comandos semelhantes ao do exemplo via código Java.

A saída produzida pela JUMBLE é coletada e gravada em um arquivo texto. Após a execução dos casos de teste, o *plugin* processa as informações gravadas e fornece à PROGTEST os resultados e relatórios associados.

Plugin CUNIT

Dentre as ferramentas que apoiam a linguagem C, o CUNIT foi a primeira a ser integrada ao ambiente PROGTEST. A ideia de integrar o CUNIT teve por motivação estabelecer para a PROGTEST um formato padrão de especificação de casos de teste para programas em C. Uma vez que os conjuntos de teste são implementados no formato CUNIT, eles podem ser facilmente mantidos e automaticamente executados pela PROGTEST. Neste sentido, um *plugin* foi construído capaz de executar os conjuntos de teste CUNIT, provendo informações sobre quais casos de teste falharam e quais casos de teste foram bem sucedidos.

O CUNIT é fornecido como uma biblioteca estática. Considerando o compilador GCC, o parâmetro `-lcunit` deve ser utilizado para que o GCC vincule a biblioteca durante o processo de compilação do programa e conjunto de teste. Para executar os casos de teste com o CUNIT, deve-se apenas iniciar o arquivo executável resultante do processo de compilação.

De maneira semelhante ao *plugin* JUMBLE, o *plugin* CUNIT utiliza o método `Runtime.getRuntime().exec(String args[])`, para (1) executar o GCC, compilando os programas e casos de teste; e (2) iniciar o arquivo executável gerado, resultando na execução dos casos de teste.

O CUNIT pode prover os resultados dos casos de teste no console ou em um arquivo XML. Como nos arquivos XML os resultados se encontram estruturados, a obtenção dos resultados pelo *plugin* a partir de arquivos XML se tornou a opção mais viável.

Após a execução da ferramenta, o *plugin* processa os arquivos XML gerados e envia para a PROGTEST os resultados e relatórios associados.

Plugin GCOV

A ferramenta GCOV foi integrada à PROGTEST com objetivo de inserir apoio a critérios estruturais para programas em C.

De forma semelhante ao CUNIT, a GCOV também opera em conjunto com o compilador GCC. Durante o processo de compilação, o parâmetro `--coverage` deve ser utilizado para que o GCC adicione no código objeto gerado instruções associadas a atividade de teste. Após executar o arquivo executável gerado, tais instruções gravam em um arquivo específico, quais trechos do código foram executados.

Executando o GCOV por meio do comando exemplificado na Figura 3.15, a ferramenta exibe no console a porcentagem de cobertura para cada critério de teste.

```
1 gcov -a -b AbstractFoo.c
```

Figura 3.15: Exemplo de Comando Utilizado Para Executar o GCOV

No *plugin* GCOV, os comandos também são executados por meio do método `Runtime.getRuntime().exec(String args[])`. Para a obtenção dos resultados, a mesma estratégia do *plugin* JUMBLE é utilizada, a qual consiste em coletar a saída produzida e gravá-la em arquivos textos. Após a execução dos casos de teste, os arquivos texto são processados e os resultados e relatórios enviados a PROGTEST.

Plugin PROTEUM

A ferramenta PROTEUM foi integrada à PROGTEST com objetivo de inserir apoio a critérios baseados em erros (Análise de Mutantes) para programas em C.

A PROTEUM, basicamente, é composta por um conjunto de subprogramas, cada um responsável em prover um determinado conjunto de funcionalidades. O subprograma `ptest` por exemplo é responsável pelo gerenciamento de projetos de teste. Já o `tcase` é responsável pelo gerenciamento de casos de teste.

Neste sentido, a execução da PROTEUM é realizada pelo *plugin* por meio de um *script*, desenvolvido especialmente para o *plugin*. O *script* executa uma sequência de subprogramas da PROTEUM com as configurações adequadas, de forma que, ao fim da execução, o *score* de mutação do conjunto de teste em questão para um determinado arquivo do programa pode ser obtido.

Após executar o *script* para cada arquivo do programa, o *plugin* processa os arquivos gerados pela PROTEUM e envia os resultados e relatórios para a PROGTEST.

A execução do *script* também é realizada via `Runtime.getRuntime().exec(String args[])`.

3.4 Evolução da Base de Trabalhos Oráculo

A PROGTEST disponibiliza aos professores uma base de trabalhos oráculo. A base de trabalhos tem por finalidade facilitar o trabalho do professor, que pode usar os trabalhos oráculo disponíveis ao invés de definir, implementar e testar o seu próprio programa.

Uma vez que novas ferramentas de teste foram integradas à PROGTEST, a base de trabalhos foi evoluída a fim de adequar os trabalhos aos novos critérios e linguagens de programação apoiados por tais ferramentas.

A Tabela 3.6 mostra os trabalhos oráculo desenvolvidos para apoiar os novos critérios adotados. Tais trabalhos foram desenvolvidos tanto para a linguagem Java como para a linguagem C. Para os programas em linguagem Java foi construído um conjunto de teste 100%-adequado

aos critérios de teste apoiados pelas ferramentas JABUTISERVICE e JUMBLE. De forma semelhante, para os programas em C foi construído um conjunto de teste 100%-adequado aos critérios de teste apoiado pelas ferramentas GCOV e PROTEUM.

Tabela 3.6: Trabalhos Oráculo em C

#	Título	Descrição
1	Valor Máximo	Identifica o valor máximo de uma sequência de números inteiros.
2	Valor Mínimo	Identifica o valor mínimo de uma sequência de números inteiros.
3	Valor Máximo e Mínimo	Identifica o valor máximo e mínimo de uma sequência de números inteiros.
4	Identificador	Identifica se uma *string* é um identificador válido.
5	Triângulo	Identifica se um triângulo é isósceles, equilátero ou escaleno.
6	Fatorial	Calcula o fatorial de um número.
7	Fibonacci	Calcula a sequência de Fibonacci.
8	Ordenação	Ordena uma sequência de números inteiros.
9	Calendário	Determina o dia da semana de uma determinada data.
10	Palíndromo	Determina se uma palavra é palíndromo.

No caso da ferramenta JUMBLE, todos os operadores de mutação apoiados pela ferramenta (Tabela 3.7) foram considerados. No entanto, no caso da ferramenta PROTEUM, devido ao grande número de operadores, a quantidade de mutantes gerados inviabilizou a utilização de todos os operadores.

Tabela 3.7: JUMBLE: Operadores de Mutação

Operador	Descrição
Conditionals	Substitui cada condição pela sua negação.
Binary Arithmetic Operation	Substitui cada operação aritmética binária por outra operação.
Increments	Incrementos e decrementos são substituídos pelo seu sinal oposto.
Inline Constants	Modifica o valor literal de constantes.
Class Pool Constants	Modifica o valor literal de *Strings*.
Return Values	Modifica os valores de retorno dos métodos.
Switch Statements	Modifica os comandos `switch`, trocando cada `case` pelo `default case` ou por outro `case`.

Neste sentido, para os programas em C, foram considerados os operadores essenciais definidos no trabalho de (Barbosa, 1998). Tais operadores são apresentados na Tabela 3.8.

Por fim, os demais trabalhos em linguagem Java, que já estavam presentes na base, tiveram seus conjuntos de teste evoluídos, a fim de torna-los 100%-adequado ao critério Análise de Mutantes apoiado pela ferramenta JUMBLE. A Tabela 3.9 mostra o conjunto final de trabalhos oráculo em Java disponível na PROGTEST.

3.5 Considerações Finais

Neste capítulo foram apresentados aspectos relacionados à integração de ferramentas ao ambiente PROGTEST. Inicialmente, diversas ferramentas de teste foram identificadas e analisadas,

Tabela 3.8: PROTEUM: Operadores Essenciais

Sigla	Operador	Descrição
OLBN	*Logical Operator by Bitwise Operator*	Substitui operadores cada operador lógico por operadores bit-a-bit.
ORRN	*Relational Operator Mutation*	Substitui cada operador relacional por outros operadores relacionais.
SMTC	*n-trip* `continue`	Interrompe execução de laços após duas execuções.
SSDL	*Statement Delection*	Deleta cada comando ou bloco de comandos.
SWDD	`while` *Replacement by* `do-while`	Substitui cada comando `while` por *do-while*.
VDTR	*Domain Traps*	Força cada valor escalar assumir valores negativo, zero e positivo.
VTWD	*Twiddle Mutations*	Substitui cada valor escalar pelo seu sucessor e predecessor.
Cccr	*Constant for Constant Replacement*	Substitui cada constante por outras constantes.
Ccsr	*Constant for Scalar Replacement*	Substitui cada constante por valores escalares.

Tabela 3.9: Trabalhos Oráculo em Java

#	Título	Descrição
1	Valor Máximo	Identifica o valor máximo de uma sequência de números inteiros.
2	Valor Mínimo	Identifica o valor mínimo de uma sequência de números inteiros.
3	Valor Máximo e Mínimo	Identifica o valor máximo e mínimo de uma sequência de números inteiros.
4	Identificador	Identifica se uma *string* é um identificador válido.
5	Triângulo	Identifica se um triângulo é isósceles, equilátero ou escaleno.
6	Fatorial	Calcula o fatorial de um número.
7	Fibonacci	Calcula a sequência de Fibonacci.
8	Ordenação	Ordena uma sequência de números inteiros.
9	Calendário	Determina o dia da semana de uma determinada data.
10	Palíndromo	Determina se uma palavra é palíndromo.
11	Lista	Implementação de lista.
12	Pilha	Implementação de pilha.
13	Fila	Implementação de fila.
14	Árvore Binária	Implementação de árvore binária.
15	Grafo	Implementação de grafo.
16	Busca em Profundidade	Realiza busca em profundidade em grafos.
17	Busca em Largura	Realiza busca em largura em grafos.
18	CFC	Obtém o número de componentes fortemente conectadas em grafos.
19	Algoritmo de Prim	Implementa o algoritmo de Prim.
20	Tabela *Hash*	Implementação de Tabela *Hash*.
21	Multiplicação de Matriz	Realiza a multiplicação de duas matrizes.
22	Casamento de *Strings*	Procura a ocorrência de uma *string* em outra *string*.

a fim de identificar as principais características das ferramentas de teste que poderiam facilitar ou dificultar sua integração.

A partir da análise realizada, a PROGTEST foi evoluída a fim de torná-la extensível para a utilização de diferentes ferramentas de teste. Para cada ferramenta considerada, um *plugin* específico foi construído. Ainda, a base de trabalhos oráculo da PROGTEST foi evoluída a fim de adequá-la às novas ferramentas de teste e linguagens de programação consideradas.

Além da integração de ferramentas de teste, um dos objetivos deste trabalho consiste na validação do ambiente PROGTEST. Assim, no próximo capítulo é detalhado como o ambiente

PROGTEST foi validado. Os principais resultados obtidos a partir das validações conduzidas também são discutidos.

CAPÍTULO

4

Aplicação e Validação do Ambiente PROGTEST

4.1 Considerações Iniciais

Um dos principais objetivos deste trabalho foi a integração de diferentes ferramentas de teste ao ambiente PROGTEST; no capítulo anterior foi apresentado como tal objetivo foi alcançado. Outro objetivo relacionado consiste em validar o ambiente PROGTEST. Assim, neste capítulo é discutido como a PROGTEST foi validada e os principais resultados obtidos a partir desta atividade.

Na Seção 4.2 são discutidas validações conduzidas considerando trabalhos de programação em Java. Basicamente, a PROGTEST foi aplicada em ambientes reais de ensino, considerando sua integração com p *framework* JUNIT e as ferramentas de teste estrutural (JABUTISERVICE) e teste baseado em erros (JUMBLE).

Na Seção 4.3 são detalhadas validações realizadas com programas em C. Além do *framework* CUNIT, como apoio ao teste estrutural, foi utilizada a ferramenta GCOV; para o teste baseado em erros, a ferramenta PROTEUM.

Por fim, na Seção 4.4 são discutidas as principais vantagens e limitações da PROGTEST identificadas a partir das validações realizadas.

4.2 Validação com Programas Java

Nesta seção, são detalhadas as validações realizadas a fim de verificar o comportamento da PROGTEST com programas escritos em Java. Inicialmente, foi considerada a utilização do ambiente com ferramentas de apoio ao teste estrutural. Em seguida, validações foram realizadas considerando, além do teste estrutural, ferramentas de apoio ao teste baseado em erros.

4.2.1 Validação com Teste Estrutural

A primeira validação da PROGTEST foi conduzida no contexto de um projeto de iniciação científica (Processo FAPESP 09/00006-3), realizado pelo autor deste trabalho. Neste projeto, foram desenvolvidos diversos programas e conjuntos de teste em Java, sendo que alguns deles possuíam defeitos. Em seguida, o autor submeteu os programas e conjuntos de teste na PROGTEST a fim de verificar se o ambiente, integrado a ferramentas de teste estrutural, era capaz de identificar os defeitos. Detalhes sobre como essa validação foi conduzida são descritos no Apêndice A.

Neste trabalho, foi dada continuidade as atividades de validação do ambiente PROGTEST. Com relação à validação de programas em Java, considerando critérios e ferramentas estruturais, validações foram realizadas em ambientes reais de ensino, conforme detalhado nas seções seguites.

4.2.1.1 Validação 1: Validação com Alunos de Pós-Graduação

Dentre as validações realizadas em ambiente reais de ensino, uma foi conduzida com um conjunto de alunos em uma disciplina de pós-graduação (Validação 1). Nesta seção é detalhado como a validação foi conduzida e os resultados obtidos a partir da sua realização.

Descrição do Ambiente

A Validação 1 foi realizada no 1º semestre de 2011, na disciplina Verificação, Validação e Teste de Software, oferecida aos alunos de mestrado e doutorado do Programa de Pós-Graduação em Ciências da Computação e Matemática Computacional do ICMC/USP.

A disciplina possuía 8 alunos matriculados: 6 alunos de mestrado e 2 alunos de doutorado. Todos os alunos atuavam na área de Engenharia de Software.

Durante a disciplina, foram abordados em sala de aula tópicos relacionados a teste de software, em especial, critérios das técnicas funcionais e estruturais. Além disso, foram realizadas demonstrações de ferramentas de apoio a atividade de teste, como o *framework* JUNIT (Beck e Gamma, 2010) e a ferramenta JABUTI (Vincenzi et al., 2003). Além disso, também foi

realizada aos alunos uma demonstração do ambiente PROGTEST e de suas principais funcionalidades.

Como parte das atividades da disciplina, foi solicitado aos alunos que implementassem o programa `Cal` (calendário), utilizando a linguagem Java, o qual também deveria ser testado com o apoio de ferramentas de teste. Durante o andamento do trabalho, os alunos deveriam submeter seus trabalhos à PROGTEST, a fim de obterem um *feedback* sobre a qualidade dos seus programas e dos conjuntos de teste.

Definição do Trabalho

A fim de possibilitar a submissão dos trabalhos dos alunos na PROGTEST, uma definição de trabalho foi criada no ambiente. Dentre as propriedades a serem especificadas destacam-se os critérios de teste a serem considerados na avaliação e os pesos associados. Em síntese, foram considerados, além da correção[1] calculada pelo JUNIT, os critérios estruturais Todos-Nós (`All-Nodes-ei` e `All-Nodes-ed`), Todas-Arestas (`All-Edges-ei` e `All-Edges-ed`), Todos-Usos (`All-Uses-ei` e `All-Uses-ed`) e Todos-Potenciais-Usos (`All-Potencial-Uses-ei` e `All-Potencial-Uses-ed`).

Para cada um dos critérios estruturais foram atribuídos pesos 1, enquanto para a correção foi atribuído peso 8. A ideia é que, juntos, os 8 critérios estruturais com peso 1 tivessem a mesma relevância que a correção do programa e conjunto de teste. Em outras palavras, metade da nota do aluno seria reflexo da correção do programa e casos de teste avaliada pelo JUNIT e a outra metade reflexo da qualidade do programa e casos de teste avaliados segundo as coberturas obtidas pelos critérios estruturais.

A especificação do trabalho fornecida aos alunos encontra-se no Apêndice B.1. Como programa do professor, um trabalho oráculo para o programa `Cal` foi desenvolvido, sendo composto por um programa que representa uma solução para o problema e um conjunto de teste 100%-adequado aos critérios considerados. No total, 15 casos de teste foram projetados para o programa `Cal`.

Análise dos Resultados

Os trabalhos submetidos pelos alunos foram coletados e os resultados fornecidos pela PROGTEST analisados.

A Tabela 4.1 mostra o resultado das execução dos casos de teste. Considerando, como exemplo, o Aluno 3, tem-se que os 12 casos de teste projetados pelo aluno não causaram falha no programa dele ($P_{St_i} - T_{St_i}$). No entanto, ao executar os casos de teste do aluno no programa do professor ($P_{Inst} - T_{St_i}$), 4 causaram falha no programa, indicando a presença de casos de teste incorretos no programa do aluno. Ainda, ao executar os casos de teste do professor no

[1] porcentagem dos casos de teste que não falharam

programa do aluno ($P_{St_i} - T_{Inst}$), 4 dos 15 casos de teste do professor causaram falha no programa, indicando a presença de defeitos no programa do aluno.

Tabela 4.1: Validação 1: Resultado da Execução dos Casos de Teste

Aluno	$P_{Inst} - T_{Inst}$	$P_{St_i} - T_{St_i}$	$P_{Inst} - T_{St_i}$	$P_{St_i} - T_{Inst}$
1	15/15	5/5	0/5	0/15
2	15/15	15/15	15/15	8/15
3	15/15	12/12	8/12	11/15
4	15/15	17/17	17/17	15/15
5	15/15	65/65	65/65	15/15
6	15/15	16/16	16/16	15/15
7	15/15	12/12	10/12	11/15
8	15/15	9/9	9/9	15/15

Analisando os códigos dos alunos e os resultados das execuções dos casos de teste, foi possível observar que: (1) o Aluno 1 não utilizou a interface definida na especificação do trabalho, o que resultou na falha do programa do instrutor para todos os casos de teste do aluno e vice-versa; (2) foram identificados problemas de lógica no programa do Aluno 2 que, para alguns dados específicos de entrada, apresentou falhas; e (3) os alunos 3 e 7 não realizaram o tratamento de exceções de maneira adequada, o que também resultou em problemas.

A Tabela 4.2 mostra as coberturas totais obtidas por cada aluno e a nota sugerida pela PROGTEST. Tais coberturas são calculadas considerando a porcentagem de casos de teste que não causaram falhas e as coberturas obtidas para os critérios estruturais.

Tabela 4.2: Validação 1: Coberturas e Notas

Aluno	$P_{Inst} - T_{Inst}$	$P_{St_i} - T_{St_i}$	$P_{Inst} - T_{St_i}$	$P_{St_i} - T_{Inst}$	**Nota Sugerida**
1	100%	99.69%	0.00%	0.00%	3.32
2	100%	98.88%	95.88%	75.81%	9.02
3	100%	93.44%	79.25%	82.56%	8.51
4	100%	100.00%	98.50%	95.88%	9.81
5	100%	99.56%	100.00%	96.63%	9.87
6	100%	100.00%	100.00%	100.00%	100.00
7	100%	95.19%	87.94%	84.19%	8.91
8	100%	96.38%	97.50%	94.81%	9.62

Embora não tenha ocorrido problemas na execução dos casos de teste dos alunos em seus programas ($P_{St_i} - T_{St_i}$), observa-se que alguns alunos não obtiveram 100% de cobertura em $P_{St_i} - T_{St_i}$, uma vez que seus casos de teste não eram 100%-adequados aos critérios estruturais considerados na avaliação.

Com relação às coberturas dos casos de teste dos alunos no programa do professor ($P_{Inst} - T_{St_i}$), os alunos cujos casos de teste não causaram falhas no programa do professor obtiveram uma cobertura relativamente alta (acima de 90%). Por outro lado os alunos 1, 3 e 7, cujos conjuntos de teste já haviam apresentado problemas, obtiveram coberturas inferiores a 90% no programa do professor. Da mesma forma, as coberturas do conjunto de teste do professor

com respeito aos programas dos alunos ($P_{St_i} - T_{Inst}$) foram inferiores à 90% nos trabalhos dos alunos 1, 2, 3, 7, cujos programas apresentavam defeitos.

De fato as diferenças entre as notas dos alunos representaram a diferença de qualidade dos trabalhos submetidos por eles, de forma que, os alunos que submeteram trabalhos com mais qualidade tiveram notas maiores.

Durante a análise dos resultados, algumas limitações no ambiente PROGTEST puderam ser observadas. Uma delas refere-se à presença de requisitos de teste não executáveis. Para exemplificar, a Figura 4.1 mostra as coberturas obtidas pelo Aluno 5 em cada critério de teste considerado. Com é possível observar, os casos de teste projetados pelo aluno para o teste de seu programa não foi 100%-adequado ao critério `All-Potencial-Uses-ei`, obtendo uma cobertura de 93.0%. No entanto, ao analisar o código fornecido pelo aluno, observa-se que os requisitos de teste não cobertos pelo conjunto de teste do aluno eram, de fato, requisitos não executáveis, ou seja, que não podem ser cobertos.

GENERAL COVERAGES

Criteria	P_Inst - T_Inst	P_St - T_St	P_Inst - T_St	P_St - T_Inst
JUnit/Pass Rate	100.0%	100.0%	100.0%	100.0%
JaBUTiService/All-Nodes-ei	100.0%	100.0%	100.0%	97.0%
JaBUTiService/All-Nodes-ed	100.0%	100.0%	100.0%	100.0%
JaBUTiService/All-Edges-ei	100.0%	100.0%	100.0%	86.0%
JaBUTiService/All-Edges-ed	100.0%	100.0%	100.0%	100.0%
JaBUTiService/All-Uses-ei	100.0%	100.0%	100.0%	86.0%
JaBUTiService/All-Uses-ed	100.0%	100.0%	100.0%	100.0%
JaBUTiService/All-Potencial-Uses-ei	100.0%	93.0%	100.0%	77.0%
JaBUTiService/All-Potencial-Uses-ed	100.0%	100.0%	100.0%	100.0%

Figura 4.1: Validação 1: Coberturas Obtidas pelo Aluno 5

Além disso, embora o conjunto de teste do professor seja 100%-adequado aos critérios estruturais quando executados contra o programa do professor ($P_{Inst} - T_{Inst}$), o mesmo conjunto de teste não é 100%-adequado quando executado contra o trabalho do aluno ($P_{St_i} - T_{Inst}$). Por exemplo, a cobertura do conjunto de teste do professor no programa do Aluno 5 foi de 97.0% para o critério `All-Nodes-ei`, 86.0% para o critérios `All-Edges-ei`, 86.0% para o critério `All-Uses-ei` e 77.0% para o critério `All-Potencial-Uses-ei`.

Isto ocorre porque o algoritmo desenvolvido pelo Aluno 5 é diferente do algoritmo do programa do professor, gerando requisitos de teste diferentes. Assim, o conjunto de teste projetado para o programa do professor pode não ser 100%-adequado para o programa do aluno, uma vez que não foi projetado para ele.

Tais limitações são melhores discutidas na Seção 4.4.

Melhorias Realizadas

Com base nos *feedback* fornecido pelos alunos em relação a utilização da PROGTEST, algumas melhorias foram realizadas no ambiente. Basicamente, foram adicionadas informações que pudessem melhor ajudar os alunos a entenderem os problemas em seus programas e casos de teste.

Uma das melhorias realizadas refere-se à disponibilização de informações sobre a execução do conjunto de teste do professor no programa do aluno ($P_{St_i} - T_{Inst}$). Na versão utilizada pelos alunos, somente era possível visualizar as coberturas obtidas pelo conjunto de teste do professor, enquanto os demais relatórios disponibilizados pela PROGTEST permitiam que os alunos visualizassem com mais detalhes somente informações sobre a execução de seus conjuntos de teste em seus próprios programas ($P_{St_i} - T_{St_i}$).

Neste sentido, foram acrescentados à PROGTEST relatórios que permitem aos alunos analisarem com mais detalhes os resultados da execução do conjunto de teste do professor em seus programas ($P_{St_i} - T_{Inst}$). Assim, os mesmos relatórios gerados para o conjunto de teste do aluno agora também são gerados para o conjunto de professor.

Outra melhoria realizada refere-se ao relatório fornecido pelo *plugin* JUNIT (Seção 3.3.2). Na versão utilizada pelos alunos, este relatório especificava quantos casos de teste falharam ao serem executados sem, no entanto, especificar quais eram esses casos de teste. Essa informação é fundamental para que os alunos saibam exatamente quais casos de teste do seu conjunto encontram-se com problemas.

Por esse motivo, foi acrescentado ao relatório uma lista de falhas geradas pelo conjunto de teste, conforme ilustrado na Figura 4.2. Para cada falha gerada pelo conjunto de teste é apresentado o nome do caso de teste em que a falha ocorreu, o nome da classe de teste em que o caso de teste se encontra, a saída esperada e a saída gerada pelo programa.

Ainda, se mensagens de erro forem definidas nos casos de teste, elas também serão exibidas quando o caso de teste falhar. A visualização de mensagens de erro pode ser bastante útil aos alunos quando se referem à execução do conjunto de teste do professor no programa do aluno. Por meio delas, o professor pode fornecer dicas aos alunos sobre quais funcionalidades do programa não estão sendo desempenhadas corretamente.

Por fim, outra melhoria realizada refere-se à adição de grafos de fluxo de controle nos relatórios providos pelo *plugin* JABUTISERVICE (Seção 3.3.2). A Figura 4.3 mostra parte de um relatório fornecido pelo *plugin*. Nele é fornecido aos alunos um grafo de fluxo de controle para um determinado método do programa e informações sobre os requisitos de teste derivados para o critério `All-Edges-ei`. No grafo, é possível visualizar a aresta não coberta pelo conjunto de teste, destacada na cor vermelha. Embora já fosse possível identificar quais requisitos estruturais eram cobertos e quais não eram, uma visualização gráfica dos requisitos pode melhor guiar e incentivar os alunos na realização das atividades de teste.

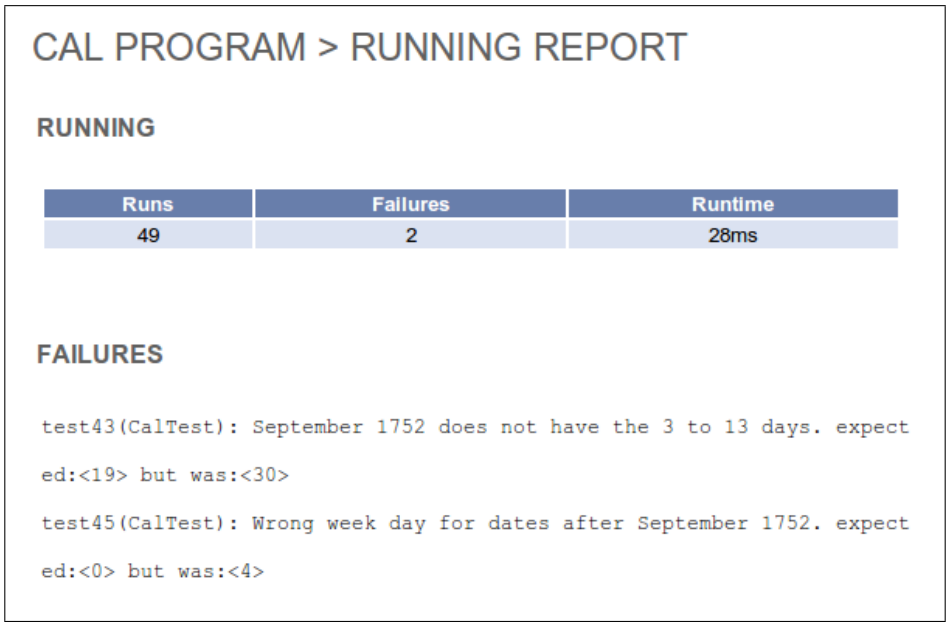


Figura 4.2: Relatório Provido pelo *Plugin* do JUNIT

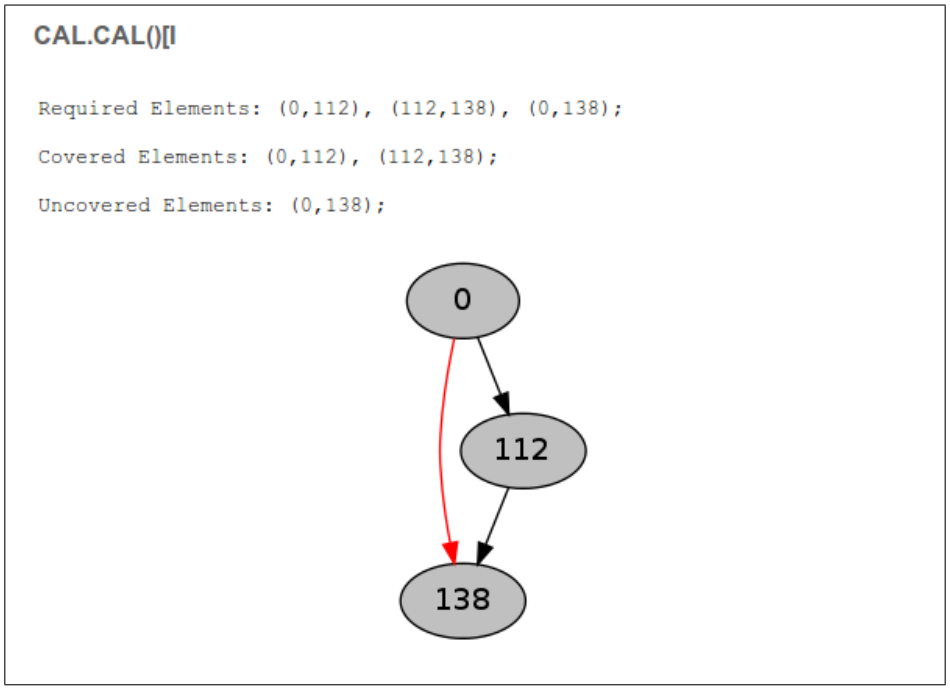


Figura 4.3: Relatório Provido pelo *Plugin* do JABUTISERVICE

4.2.1.2 Validação 2: Validação com Alunos de Graduação

Outra validação conduzida envolvendo a aplicação da PROGTEST foi realizada com alunos de graduação (Validação 2). Tal validação e os resultados obtidos a partir da sua realização são apresentados a seguir.

Descrição do Ambiente

A validação do ambiente PROGTEST com alunos de graduação foi realizada no 2º semestre de 2011, na disciplina de Tópicos Especiais em Engenharia de Software. A disciplina é oferecida pelo ICMC/USP aos alunos do curso de Bacharelado em Informática. No total, 24 alunos participaram da validação.

Na disciplina, os alunos tiveram aulas expositivas sobre técnicas e critérios funcionais e estruturais, além de aulas em laboratório, onde foram apresentados a ferramentas de apoio.

A fim de familiarizar os alunos na utilização das técnicas e das ferramentas de teste, foi solicitado aos alunos que realizassem as seguintes atividades: (1) implementar o programa `Cal` aplicando o *Testing Driven Development* (TDD), utilizando o *framework* JUNIT para implementar os casos de teste; e (2) testar o programa desenvolvido segundo critérios funcionais e estruturais, utilizando a ferramenta JABUTI para apoiar a aplicação dos critérios estruturais.

Por fim, foi solicitado aos alunos que submetessem os seus trabalhos à PROGTEST e que melhorassem a qualidade dos seus programas e conjuntos de teste de acordo com *feedback* fornecido pelo ambiente.

Definição do Trabalho

Na Validação 2 foram considerados, além da correção avaliada pelo JUNIT, os critérios estruturais Todos-Nós (`All-Nodes-ei` e `All-Nodes-ed`), Todas-Arestas (`All-Edges-ei` e `All-Edges-ed`) e Todos-Usos (`All-Uses-ei` e `All-Uses-ed`). O critério Todos-Potenciais-Usos (`All-Potencial-Uses-ei` e `All-Potencial-Uses-ed`) não foi utilizado, uma vez que o critério não foi abordado com os alunos em sala de aula.

De forma similar à Validação 1, para cada um dos critérios estruturais foi atribuídos pesos 1, enquanto para a correção foi atribuído peso 6. A ideia é que, juntos, os 6 critérios estruturais com peso 1 tivessem a mesma relevância que a correção do programa e casos de teste.

Com relação ao trabalho oráculo, o mesmo programa foi considerado. No entanto, a fim de estabelecer uma avaliação mais precisa dos trabalhos dos alunos, o conjunto de teste foi reforçado, totalizando 49 casos de teste.

Análise dos Resultados

Com relação aos defeitos identificados no programas e casos de teste dos alunos, a Tabela 4.3 mostra o resultado das execuções dos casos de teste no JUNIT. Considerando como exemplo o Aluno 23, observa-se que: (1) seu programa comportou-se corretamente para os 14 casos de teste projetados por ele; (2) o programa do professor também comportou-se corretamente para os 14 casos de teste do aluno; e (3) o programa do aluno comportou-se corretamente para os 49 casos de teste projetados pelo professor.

Tabela 4.3: Validação 2: Resultado da Execução dos Casos de Teste

Aluno	$P_{Inst} - T_{Inst}$	$P_{St_i} - T_{St_i}$	$P_{Inst} - T_{St_i}$	$P_{St_i} - T_{Inst}$	**Nota Sugerida**
1	49/49	6/6	6/6	47/49	9.32
2	49/49	21/21	19/21	0/49	6.47
3	49/49	48/48	0/48	0/49	3.14
4	49/49	11/11	11/11	49/49	9.72
5	49/49	14/14	14/14	47/49	9.68
6	49/49	20/20	20/20	48/49	9.89
7	49/49	51/51	50/51	49/49	9.54
8	49/49	41/41	40/41	47/49	9.67
9	49/49	28/28	26/28	42/49	9.23
10	49/49	7/7	0/7	0/49	1.67
11	49/49	28/28	27/28	49/49	9.46
12	49/49	32/32	31/32	49/49	9.89
13	49/49	19/19	0/19	0/49	2.3
14	49/49	23/23	21/23	42/49	9.56
15	49/49	13/13	12/13	42/49	9.2
16	49/49	46/46	45/46	49/49	9.68
17	49/49	9/9	0/9	42/49	6.17
18	49/49	12/15	12/15	49/49	9.27
19	49/49	49/49	0/49	49/49	6.63
20	49/49	15/15	12/15	41/49	9.26
21	49/49	31/31	30/31	49/49	9.84
22	49/49	16/16	15/16	41/49	9.47
23	49/49	14/14	14/14	49/49	9.66
24	49/49	20/20	17/20	41/49	9.2

Por outro lado, considerando o Aluno 9, temos que: (1) o programa do aluno se comportou corretamente para os 28 casos de teste projetados por ele; (2) o programa do professor falhou para 2 dos 28 casos de teste do aluno, indicando possíveis defeitos no projeto de casos de teste do aluno; e (3) o programa do aluno falhou para 7 dos 49 casos de teste do professor, indicando a presença de possíveis defeitos no programa do aluno.

Em relação as notas, dos 24 alunos que participaram da validação: (1) 18 conseguiram atigir notas maiores que 9.00; (2) 3 obtiveram notas medianas, entre 5.00 e 7.00; e (3) 3 obtiveram notas baixas, menores que 5.00.

A Tabela 4.4 mostra as coberturas totais obtidas por cada aluno. Observa-se que os alunos que obtiveram notas medianas também obtiveram uma cobertura de 0.00% nas colunas $P_{Inst} - T_{St_i}$ ou $P_{St_i} - T_{Inst}$. Além disso, os alunos que obtiveram as notas mais baixas, atingiram uma cobertura de 0.00% em ambas colunas $P_{St_i} - T_{Inst}$ e $P_{Inst} - T_{St_i}$. Tais colunas, referem-se à execução do conjunto de teste do professor no programa do aluno e à execução do conjunto de teste do aluno no programa do professor, respectivamente.

Isso mostra que os alunos tiveram dificuldades em compreender como deveria ser implementada a interface entre seus programas e casos de teste. Uma vez que o aluno não implementou a interface do seu programa e casos de teste de acordo com a especificação, ao realizar as execuções trocadas entre o trabalho do professor (oráculo) e o trabalho do aluno, um erro

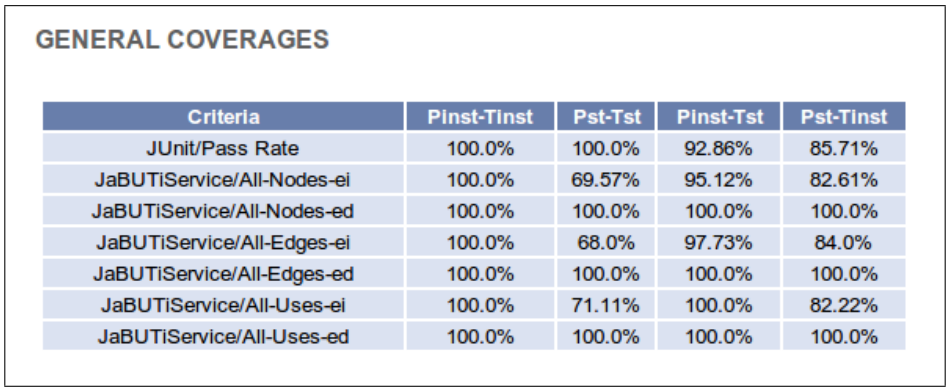
Tabela 4.4: Validação 2: Coberturas e Notas

Aluno	$P_{Inst} - T_{Inst}$	$P_{St_i} - T_{St_i}$	$P_{Inst} - T_{St_i}$	$P_{St_i} - T_{Inst}$	**Nota Sugerida**
1	100.00%	87.06%	94.74%	97.73%	9.32
2	100.00%	99.64%	94.57%	0.00%	6.47
3	100.00%	94.33%	0.00%	0.00%	3.14
4	100.00%	95.26%	97.12%	99.12%	9.72
5	100.00%	96.18%	96.49%	97.71%	9.68
6	100.00%	98.33%	99.66%	98.59%	9.89
7	100.00%	95.85%	98.42%	92.07%	9.54
8	100.00%	96.53%	98.18%	95.24%	9.67
9	100.00%	92.39%	95.83%	88.59%	9.23
10	100.00%	50.00%	0.00%	0.00%	1.67
11	100.00%	92.69%	97.28%	93.73%	9.46
12	100.00%	98.83%	97.84%	100.00%	9.89
13	100.00%	69.09%	0.00%	0.00%	2.3
14	100.00%	100.00%	93.90%	92.86%	9.56
15	100.00%	90.91%	93.81%	91.32%	9.2
16	100.00%	96.99%	98.32%	95.18%	9.68
17	100.00%	93.22%	0.00%	91.92%	6.17
18	100.00%	89.11%	89.71%	99.42%	9.27
19	100.00%	98.83%	0.00%	100.00%	6.63
20	100.00%	98.31%	87.66%	91.84%	9.26
21	100.00%	99.04%	96.04%	100.00%	9.84
22	100.00%	97.73%	94.53%	91.84%	9.47
23	100.00%	94.92%	97.66%	97.15%	9.66
24	100.00%	95.47%	91.9%	88.64%	9.2

de compilação ocorre, fazendo com que o aluno obtenha 0.00% de cobertura. Tal aspecto é melhor discutido na Seção 4.4. Ressalta-se, entretanto, que a maioria dos alunos conseguiu implementar a interface de seus programas adequadamente.

Considere como exemplo o Aluno 9. As coberturas obtidas pelo aluno para cada critério de teste, em cada execução, são apresentadas na Figura 4.4. Considerando os critérios estruturais, nota-se que para os critérios dependentes de exceção (`All-Nodes-ed`, `All-Edges-ed` e `All-Uses-ed`), não houve problemas e o aluno conseguiu uma cobertura de 100% em todas as execuções. Por outro lado, o mesmo não ocorre em relação aos critérios independentes de exceção (`All-Nodes-ei`, `All-Edges-ei` e `All-Uses-ei`). Para o critério `All-Nodes-ei`, por exemplo, ao executar os casos de teste do aluno contra seu próprio programa ($P_{St_i} - T_{St_i}$), a cobertura obtida foi de 68.0%. No entanto, ao executar os casos de teste do aluno contra o programa do professor ($P_{Inst} - T_{St_i}$), a cobertura obtida para o critério `All-Nodes-ei` foi de 97.73%.

Tal diferença entre as cobeturas está relacionada à diferença de qualidade entre o programa do aluno e o programa do professor. Ao analisar o programa do aluno, observou-se que o código não estava adequadamente modularizado. Trechos idênticos de código, por exemplo, encontravam-se repetidos em três pontos diferentes do programa do aluno. De fato, a falta de modularidade no programa resulta em um aumento significativo na quantidade de requisitos de



GENERAL COVERAGES

Criteria	Pinst-Tinst	Pst-Tst	Pinst-Tst	Pst-Tinst
JUnit/Pass Rate	100.0%	100.0%	92.86%	85.71%
JaBUTiService/All-Nodes-ei	100.0%	69.57%	95.12%	82.61%
JaBUTiService/All-Nodes-ed	100.0%	100.0%	100.0%	100.0%
JaBUTiService/All-Edges-ei	100.0%	68.0%	97.73%	84.0%
JaBUTiService/All-Edges-ed	100.0%	100.0%	100.0%	100.0%
JaBUTiService/All-Uses-ei	100.0%	71.11%	100.0%	82.22%
JaBUTiService/All-Uses-ed	100.0%	100.0%	100.0%	100.0%

Figura 4.4: Validação 2: Coberturas Obtidas pelo Aluno 9

teste e, consequentemente, exige um conjunto de teste mais forte para se satisfazer os critérios em questão. Ao observar a coluna $P_{St_i} - T_{Inst}$, é possível perceber que a baixa modularidade do programa do aluno faz com que o conjunto do professor também não atinja coberturas maiores que 90% quando executado contra ele.

Ainda, na coluna $P_{Inst} - T_{St_i}$, o aluno obteve cobertura de 92.06% no JUNIT, ou seja, um ou mais casos de teste projetados pelo aluno falhou ao ser executado contra o programa do professor, indicando possíveis defeitos nos casos de teste do aluno. Da mesma forma, na coluna $P_{St_i} - T_{Inst}$, o aluno obteve cobertura de 85,71%, ou seja, casos de teste do professor falharam ao serem executados contra o programa do aluno, indicando possíveis defeitos no programa do aluno.

Ao final do processo de submissão e avaliação automática, o Aluno 9 obteve nota 9.23. Embora o trabalho do aluno tenha recebido coberturas baixas para os critérios estruturais independentes de exceções, as coberturas de 100% nos critérios estruturais dependentes de exceção elevaram sua nota. Ainda, as coberturas calculadas nas execuções com o JUNIT (100%, 92.86% e 85.71%) também contribuíram para que o aluno recebesse uma nota acima de 9.0.

Ressalta-se, no entanto, que uma configuração diferente por parte do professor nos pesos atribuídos aos critérios de teste durante a criação do trabalho poderia fornecer um resultado diferente. De fato, se pesos maiores fossem atribuídos aos critérios independentes de exceção, as baixas coberturas obtidas pelo aluno em tais critérios teriam uma maior influência em sua nota final.

Além disso, embora a avaliação realizada pela PROGTEST no trabalho do aluno tenha sido adequada, um *feedback* mais preciso sobre a necessidade de melhor modularizar o seu código poderia ter melhor incentivado e ajudado o aluno a melhorar o seu código.

O *feedback* imediato é uma das características importantes da PROGTEST, pois ajuda os alunos a identificar e corrigir problemas em seus trabalhos antes da avaliação final realizada pelo professor. Por esse motivo, foi solicitado aos alunos que preenchessem um formulário com o histórico de suas submissões.

A Tabela 4.5 mostra o histórico de submissões fornecido pelo Aluno 6. O aluno documentou a realização de 5 submissões. Na primeira submissão, o aluno conseguiu uma cobertura de 96.6% executando os seus casos de teste contra o seu programa ($P_{St} - T_{St}$), no entanto, obteve 0% para as demais execuções ($P_{Inst} - T_{St}$ e $P_{St} - T_{Inst}$). Novamente, ressalta-se a dificuldade dos alunos em implementar a inteface dos seus programas e casos de teste conforme a especificação.

Tabela 4.5: Validação 2: Histórico de Submissões do Aluno 6

#	$P_{St} - T_{St}$	$P_{Inst} - T_{St}$	$P_{St} - T_{Inst}$	**Nota**	**Problemas Identificados**
1	96.6%	0%	0%	3.15	
2	96.6%	98.5%	97.58%	9.76	Test38 (CalTest): Wrong week day for dates before September 1752. expected <6> but was: 2. Test44 (CalTest): Wrong week day for dates before September 1752. expected <6> but was: 2.
3	98.14%	97.03%	97.58%	9.76	Test38 (CalTest): Wrong week day for dates before September 1752. expected <6> but was: 2. Test44 (CalTest): Wrong week day for dates before September 1752. expected <6> but was: 2.
4	98.09%	99.66%	98.59%	9.88	Test45 (CalTest): Wrong week day for dates after September 1752. expected <0> but was: <4>.
5	98.33%	99.66%	98.59%	9.89	Test45(CalTest): Wrong week day for dates after September 1752. expected:<0> but was:<4>

Na segunda submissão é possível observar que o aluno conseguiu adequar a interface dos seus programas e casos de teste conforme a especificação, melhorando sua nota de 3.15 para 9.76. No entanto, o aluno identificou a presença de problemas no programa, documentando as falhas e mensagens fornecidas pela PROGTEST. Mais especificamente, dois casos de teste do professor falharam no programa do aluno; ambos referem-se ao cálculo do dia da semana para datas antes de `Setembro de 1752`.

Na terceira submissão, embora a nota do aluno não tenha sofrido alteração, é possível perceber que o aluno fez alterações no seu conjunto de teste. A cobertura do conjunto de teste do aluno no seu próprio programa ($P_{St} - T_{St}$) subiu de 96.6% para 98.14%. No entanto, problemas no conjunto de teste fizeram com que a cobertura do conjunto de teste do aluno no programa do professor ($P_{Inst} - T_{St}$) decrecesse de 98.5% para 97.03%. A cobertura dos casos de teste do professor no programa do aluno ($P_{St} - T_{Inst}$) mostra que poucas (ou nenhuma) alteração foi realizada no programa do aluno, uma vez que a cobertura se manteve em 97.58%. Ainda, a PROGTEST continuou apresentando os mesmos problemas no programa do aluno.

Na quarta submissão, nota-se que o aluno realizou alterações em seu programa e correções no conjunto de teste. Embora tenha havido um pequeno decréscimo na cobertura do programa do aluno no programa do aluno ($P_{St} - T_{St}$), as demais coberturas e a nota final subiram. Além disso, os casos de teste (`Test38` e `Test44`) do professor não apresentaram falhas como nas submissões anteriores; entretanto, um outro caso de teste (`Test45`) do professor falhou nesta

submissão, mostrando que a nova versão do programa do aluno apresentava falhas para datas posteriores a `Setembro de 1752`.

Na última submissão, o aluno conseguiu melhorar a cobertura de seus casos de teste em seu programa ($P_{St} - T_{St}$), que subiu de 98.09% para 98.33%. Apesar disso, seu programa continuou falhando para um dos casos de teste do professor. De fato, embora o aluno não tenha acesso aos valores de entrada dos casos de teste do professor, os casos de teste `Test38` e `Test44` testam o programa do aluno com as datas `Fevereiro de 1752` e `Agosto de 1752`. Por outro lado, o caso de teste `Test45` testa o trabalho dos alunos com a data `Outubro de 1752`. Isto mostra que o problema no programa do aluno refere-se a datas no ano de 1752, ou seja, nos limites entre o fim do calendário Juliano e começo do calendário Gregoriano. Para os demais casos de teste referentes a datas anteriores e posteriores a `Setembro de 1752`, o programa do aluno comportou-se corretamente. Neste caso, um *feedback* mais preciso ao aluno poderia ter ajudado o aluno a corrigir seu programa de forma mais efetiva.

4.2.2 Validação com Teste Baseado em Erros

As validações apresentadas anteriormente possibilitaram analisar o comportamento da PROGTEST quanto à utilização de critérios estruturais para avaliar o trabalho dos alunos.

Como apresentado na Seção 3.3, a ferramenta JUMBLE (Irvine et al., 2007) foi integrada à PROGTEST a fim de fornecer apoio ao critério Análise de Mutantes para programas em Java. Neste sentido, viu-se a necessidade de realizar uma validação considerando tal critério de teste. Contudo, o critério e ferramentas associadas não tinham sido apresentadas aos alunos em detalhes. Além disso, tratava-se da primeira validação da PROGTEST com respeito ao critério Análise de Mutantes.

Neste sentido, as validações com teste baseado em erros foram conduzidas pelo autor deste trabalho, que coletou os trabalhos desenvolvidos tanto pelos alunos de pós-graduação como pelos alunos graduação e os submeteu novamente à PROGTEST para serem avaliados considerando, além dos critérios estruturais, o critério Análise de Mutantes. Tais validações são detalhadas nas seções seguintes.

4.2.2.1 Validação 3: Trabalhos dos Alunos de Pós-Graduação

Nesta seção é detalhada a Validação 3, a qual visou explorar a avaliação dos trabalhos dos alunos de pós-graduação, considerando o critério Análise de Mutantes.

Definição do Trabalho

Na Validação 3, o conjunto de operadores de mutação, utilizados para aplicar o critério Análise de Mutantes, é formado por todos os operadores apoiados pela ferramenta

JUMBLE, a saber: (1) `Conditionals`, substitui cada condição pela sua negação; (2) `Binary Arithmetic Operation`, substitui cada operação aritimética binária por outra operação; (3) `Increments`, incrementos e decrementos são substituídos pelo seu sinal oposto; (4) `Inline Constants`, modifica o valor literal de constantes; (5) `Class Pool Constants`, modifica o valor literal de *Strings*; (6) `Return Values` modifica os valores de retorno dos métodos; e (7) `Switch Statements`, modifica os comandos `switch`, trocando cada `case` pelo `default case` ou por outro `case`.

De maneira semelhante às demais validações, para cada um dos seis critérios estruturais foram atribuídos pesos 1, para a correção foi atribuído peso 6 e para o critério Análise de Mutantes foi atribuído peso 6.

Com relação ao trabalho oráculo, o mesmo programa da validaçao com alunos de graduação foi utilizado. Além disso, três novos casos de teste foram adicionados ao conjunto, de modo a torná-lo 100%-adequado ao critério Análise de Mutantes.

Análise dos Resultados

A Tabela 4.6 mostra os resultados obtidos para os trabalhos dos alunos de pós-graduação. Com respeito às notas finais, observa-se que a maioria dos trabalhos encontram-se abaixo das notas obtidas quando o critério Análise de Mutantes não foi considerado. Considerando o Aluno 3 como exemplo, na Validação 1, sua nota final foi de 8.51. Com o critério Análise de Mutantes, o aluno obteve nota 6.67.

Tabela 4.6: Validação 3: Coberturas e Notas

Aluno	$P_{Inst} - T_{Inst}$	$P_{St_i} - T_{St_i}$	$P_{Inst} - T_{St_i}$	$P_{St_i} - T_{Inst}$	**Nota Sugerida**
1	100.0%	92.81%	0.0%	0.0%	3.09
2	100.0%	76.28%	63.06%	50.73%	6.34
3	100.0%	86.39%	52.04%	61.78%	6.67
4	100.0%	95.33%	63.9%	65.38%	7.49
5	100.0%	98.39%	98.67%	99.36%	9.88
6	100.0%	93.67%	98.67%	64.74%	8.57
7	100.0%	86.54%	56.63%	63.61%	6.89
8	100.0%	94.33%	94.04%	93.64%	9.4

Por outro lado, considerando o Aluno 5, na Validação 1, sua nota foi de 9.87; com a Análise de Mutantes a nota do aluno foi 9.88, indicando que tanto seu programa como seu conjunto de teste tem boa qualidade.

É importante observar que, além da qualidade dos conjuntos de teste, um dos motivos que contribuíram para a redução da nota dos alunos refere-se a uma característica específica da ferramenta JUMBLE, que considera um escore de mutação como 0.00% quando um ou mais casos de teste falham. Para exemplificar, a Figura 4.5 mostra as coberturas obtidas pelo Aluno 3 para cada critério de teste. O escore de mutação do conjunto de teste do aluno quando executado contra o seu próprio programa ($P_{St_i} - T_{St_i}$) é de 72.0%, o que contribui para a redução da nota

do aluno. No entanto, o que mais contribui para que a nota do aluno seja menor são os escores calculados pela JUMBLE para o conjunto de teste do aluno contra o programa do professor ($P_{Inst} - T_{St_i}$) e para o conjunto de teste do professor no programa do aluno ($P_{St_i} - T_{Inst}$) – ambos são de 0.00%.

Criteria	Pinst-Tinst	Pst-Tst	Pinst-Tst	Pst-Tinst
JUnit/Pass Rate	100.0%	100.0%	58.33%	92.31%
JaBUTiService/All-Nodes-ei	100.0%	82.76%	95.12%	91.38%
JaBUTiService/All-Nodes-ed	100.0%	100.0%	100.0%	100.0%
JaBUTiService/All-Edges-ei	100.0%	76.47%	93.18%	92.65%
JaBUTiService/All-Edges-ed	100.0%	100.0%	100.0%	100.0%
JaBUTiService/All-Uses-ei	100.0%	70.29%	96.49%	81.16%
JaBUTiService/All-Uses-ed	100.0%	100.0%	100.0%	100.0%
JaBUTiService/All-Potencial-Uses-ei	100.0%	67.74%	97.46%	79.03%
JaBUTiService/All-Potencial-Uses-ed	100.0%	100.0%	100.0%	100.0%
Jumble/Mutants Analysis	100.0%	72.0%	0.0%	0.0%

Figura 4.5: Validação 3: Coberturas Obtidas pelo Aluno 3

De fato, a JUMBLE exige que o conjunto de teste sendo executado não cause falhas, para depois aplicar a Análise de Mutantes. Uma vez que o conjunto de teste do aluno foi avaliado pelo JUNIT em 58.33% de correção quando executado contra o programa do professor, a JUMBLE avalia o conjunto com 0.0% de escore. Da mesma forma, como o programa do aluno possui defeitos, o JUNIT avalia o conjunto de teste do professor contra o programa do aluno em 93.31% e, consequentemente, o escore calculado pela JUMBLE nesta execução também é de 0.0%.

Por outro lado, observando as coberturas obtidas pelo Aluno 5, apresentadas na Figura 4.6, nota-se que não foram identificados defeitos no programa e no conjunto de teste do aluno. Assim, o JUNIT avalia a correção para todas as execuções em 100%. Consequentemente, a JUMBLE realiza a Análise de Mutantes normalmente para cada uma das execuções, calculando os escores de: (1) 96.0% para o conjunto de teste do aluno contra seu próprio programa; (2) 96.0% para o conjunto de teste do aluno contra o programa do professor; e (3) 99.0% do conjunto de teste do professor no programa do aluno.

Nesse sentido, os trabalhos avaliados foram classificados em três grupos: (1) trabalhos com problemas nas interfaces entre o programa e os casos de teste; (2) trabalhos com defeitos identificados tanto no programa e como no conjunto de teste; e (3) trabalhos sem defeitos identificados.

Conforme ilustrado na Tabela 4.7, de acordo com o grupo em que um determinado trabalho pertence, a nota sugerida para ele situa-se dentro de uma determinada faixa de valor. O trabalho do Grupo 1, por exemplo, recebeu nota 3.09; os trabalhos do Grupo 2 entre 6.34 e 6.89 e assim por diante.

Por fim, a diferença de notas entre os trabalhos que pertencem a um mesmo grupo foi determinada de acordo com: (1) os escores de mutação dos conjuntos de teste com correção avaliada

Criteria	Pinst-Tinst	Pst-Tst	Pinst-Tst	Pst-Tinst
JUnit/Pass Rate	100.0%	100.0%	100.0%	100.0%
JaBUTiService/All-Nodes-ei	100.0%	100.0%	100.0%	100.0%
JaBUTiService/All-Nodes-ed	100.0%	100.0%	100.0%	100.0%
JaBUTiService/All-Edges-ei	100.0%	100.0%	100.0%	100.0%
JaBUTiService/All-Edges-ed	100.0%	100.0%	100.0%	100.0%
JaBUTiService/All-Uses-ei	100.0%	100.0%	100.0%	100.0%
JaBUTiService/All-Uses-ed	100.0%	100.0%	100.0%	100.0%
JaBUTiService/All-Potencial-Uses-ei	100.0%	93.43%	100.0%	92.58%
JaBUTiService/All-Potencial-Uses-ed	100.0%	100.0%	100.0%	100.0%
Jumble/Mutants Analysis	100.0%	96.0%	96.0%	99.0%

Figura 4.6: Validação 3: Coberturas Obtidas pelo Aluno 5

Tabela 4.7: Validação 3: Classificação dos Trabalhos

Grupo	Alunos	Notas Sugeridas
1	1	3.09
2	2, 3 e 7	De 6.34 à 6.89
3	4, 5, 6, 8	De 7.49 à 9.88

em 100%; (2) as coberturas obtidas pelos critérios estruturais; e (3) a quantidade de casos de teste que falharam durante a execução no JUNIT; segundo os pesos atribuídos a cada critério de teste.

4.2.2.2 Validação 4: Trabalhos dos Alunos de Graduação

Nesta seção é detalhada a Validação 43, a qual visou explorar a avaliação dos trabalhos dos alunos de graduação, considerando o critério Análise de Mutantes.

Definição do Trabalho

Assim como na Validação 3, o conjunto de operadores de mutação utilizados é formado por todos os operadores apoiados pela ferramenta JUMBLE. Quanto à configuração dos pesos, para cada um dos seis critérios estruturais foram atribuídos pesos 1, para a correção foi atribuído peso 6 e para o critério Análise de Mutantes foi atribuído peso 6. Por fim, o trabalho oráculo utilizado foi o mesmo utilizado na Validação 3.

Análise dos Resultados

A Tabela 4.8 mostra os resultados obtidos para os trabalhos dos alunos de graduação. Semelhante a Validação 3, observa-se que a maioria dos trabalhos encontram-se abaixo das notas obtidas quando o critério Análise de Mutantes não foi considerado. Considerando o Aluno 9 como exemplo, na Validação 2, sua nota final foi de 9.23. Com o critério Análise de Mutantes, o aluno obteve nota 7.71.

Tabela 4.8: Validação 4: Coberturas e Notas

Aluno	$P_{Inst} - T_{Inst}$	$P_{St_i} - T_{St_i}$	$P_{Inst} - T_{St_i}$	$P_{St_i} - T_{Inst}$	**Nota Sugerida**
1	100%	76.28%	88.09%	78.57%	8.1
2	100%	95.97%	77.29%	0.0%	5.78
3	100%	89.6%	0.0%	0.0%	2.99
4	100%	91.71%	94.74%	98.14%	9.49
5	100%	90.42%	92.19%	78.55%	8.71
6	100%	94.19%	98.2%	78.77%	9.04
7	100%	90.42%	78.89%	82.88%	8.41
8	100%	89.43%	78.8%	75.97%	8.14
9	100%	81.27%	77.86%	72.19%	7.71
10	100%	33.33%	0.0%	0.0%	1.11
11	100%	87.42%	78.17%	90.28%	8.53
12	100%	96.4%	78.66%	98.4%	9.12
13	100%	42.91%	0.0%	0.0%	1.43
14	100%	97.8%	76.16%	77.31%	8.38
15	100%	78.1%	75.65%	75.46%	7.64
16	100%	93.19%	78.85%	89.61%	8.72
17	100%	85.27%	0.0%	76.18%	5.38
18	100%	74.93%	75.65%	98.1%	8.29
19	100%	95.6%	0.0%	98.4%	6.47
20	100%	94.38%	73.19%	16.54%	6.14
21	100%	97.04%	76.54%	98.0%	9.05
22	100%	95.88%	75.94%	76.92%	8.29
23	100%	89.91%	92.39%	94.77%	9.24
24	100%	90.36%	76.28%	16.54%	6.11

Na Validação 2, os problemas identificados no trabalho do Aluno 9 foram refletidos em sua nota final, mas isto não impediu que ele obtivesse uma nota superior a 9.0. Considerando o critério Análise de Mutantes, os defeitos no programa e casos de teste do Aluno 9 tiveram consequências maiores, resultando em uma nota final mediana.

Por outro lado, considerando o Aluno 23, na Validação 2, sua nota foi de 9.66; com a Análise de Mutantes a nota do aluno decresceu para 9.24. Embora a sua nota tenha sido reduzida, esta continuou acima de 9.0, indicando que tanto seu programa como seu conjunto de teste tem boa qualidade.

Semelhante à Validação 3, os trabalhos avaliados foram classificados em quatro grupos: (1) trabalhos com problemas nas interfaces entre o programa e os casos de teste; (2) trabalhos com defeitos identificados tanto no programa e como no conjunto de teste; (3) trabalhos com defeitos identificados apenas no programa ou apenas no conjunto de teste; e (4) trabalhos sem defeitos identificados.

Conforme ilustrado na Tabela 4.9, de acordo com o grupo em que um determinado trabalho pertence, a nota sugerida para ele situa-se dentro de uma determinada faixa de valor. Os trabalhos do Grupo 1, por exemplo, receberam notas entre 1.11 e 6.47; os trabalhos do Grupo 2 entre 6.63 e 7.36 e assim por diante.

Tabela 4.9: Validação 4: Classificação dos Trabalhos

Grupo	Alunos	Notas Sugeridas
1	2, 3, 10, 13, 17 e 19	De 1.11 à 6.47
2	8, 9, 14, 15, 18, 20, 22 e 24	De 7.64 à 8.38
3	1, 5, 6, 7, 11, 12, 16 e 21	De 8.1 à 9.12
4	4 e 23	De 9.24 à 9.49

A diferença de notas entre os trabalhos que pertencem a um mesmo grupo foi determinada de acordo com: (1) os escores de mutação dos conjuntos de teste com correção avaliada em 100%; (2) as coberturas obtidas pelos critérios estruturais; e (3) a quantidade de casos de teste que falharam durante a execução no JUNIT; segundo os pesos atribuídos a cada critério de teste.

4.3 Validação com Programas C

Como visto na Seção 3.3, ferramentas que apóiam o teste de programas em C foram integradas ao ambiente PROGTEST. Em vista da introdução de tais ferramentas, validações foram realizadas a fim de verificar o comportamento do ambiente PROGTEST com programas em C.

As validações foram conduzidas tomando um dos trabalhos da base de trabalhos oráculo. Com base no trabalho oráculo, diferentes implementações foram produzidas, alterando o algoritmo do programa, inserindo defeitos e reduzindo o conjunto de teste. Em seguida, uma definição de trabalho foi criada na PROGTEST considerando o trabalho oráculo. As implementações foram, então, submetidas à PROGTEST para avaliação. A ideia foi verificar o impacto que cada alteração e defeito inserido implicaria nas coberturas e notas finais.

O trabalho oráculo considerado foi o `Sort`, que se refere ao problema de ordenação de números inteiros. A Tabela 4.10 mostra as características do trabalho oráculo e das implementações produzidas. O trabalho oráculo implementa o algoritmo `QuickSort`. As implementações, além do `QuickSort`, incluem outros 6 algoritmos de ordenação (`BubbleSort`, `HeapSort`, `InsertionSort`, `MergeSort`, `SelectionSort` e `ShellSort`).

Uma vez que o programa oráculo é considerado como uma solução 100% correta para o problema, defeitos no programa foram inseridos em algumas implementações. Basicamente, os defeitos inseridos fazem com que os programas ordenem inversamente os números inteiros, ou seja, considerando a sequência $S = \{5, 4, 6, 3, 7, 2, 8, 1, 9\}$, tais programas produzem a saída $S = \{9, 8, 7, 6, 5, 4, 3, 2, 1\}$ ao invés da saída correta, que seria $S = \{1, 2, 3, 4, 5, 6, 7, 8, 9\}$.

De forma semelhante, em algumas implementações, defeitos foram inseridos nos conjuntos de teste. O defeito refere-se ao caso de teste que avalia a saída do programa quando números negativos são considerados como valores de entrada. Assim, para os valores de entrada $S = \{-1, -2, -3, -4, -5, -6, -7, -8, -9, -10\}$, tal caso de teste considera como saída esperada a sequência $S = \{-1, -2, -3, -4, -5, -6, -7, -8, -9, -10\}$, ao invés da saída correta, que seria $S = \{-10. - 9, -8, -7, -6, -5, -4, -3, -2, -1\}$.

Tabela 4.10: Validação 5: Trabalhos

Trabalhos	Algoritmo	Programa	Conjunto de Teste
Oráculo	QuickSort	Sem Defeito	Sem Defeito
Implementação 1	BubbleSort	Sem Defeito	Sem Defeito
Implementação 2	HeapSort	Sem Defeito	Sem Defeito
Implementação 3	InsertionSort	Sem Defeito	Sem Defeito
Implementação 4	MergeSort	Sem Defeito	Sem Defeito
Implementação 5	QuickSort	Sem Defeito	Sem Defeito
Implementação 6	SelectionSort	Sem Defeito	Sem Defeito
Implementação 7	ShellSort	Sem Defeito	Sem Defeito
Implementação 8	BubbleSort	Sem Defeito	Com Defeito
Implementação 9	HeapSort	Sem Defeito	Com Defeito
Implementação 10	InsertionSort	Sem Defeito	Com Defeito
Implementação 11	MergeSort	Sem Defeito	Com Defeito
Implementação 12	QuickSort	Sem Defeito	Com Defeito
Implementação 13	SelectionSort	Sem Defeito	Com Defeito
Implementação 14	ShellSort	Sem Defeito	Com Defeito
Implementação 15	BubbleSort	Sem Defeito	Reduzido
Implementação 16	HeapSort	Sem Defeito	Reduzido
Implementação 17	InsertionSort	Sem Defeito	Reduzido
Implementação 18	MergeSort	Sem Defeito	Reduzido
Implementação 19	QuickSort	Sem Defeito	Reduzido
Implementação 20	SelectionSort	Sem Defeito	Reduzido
Implementação 21	ShellSort	Sem Defeito	Reduzido
Implementação 22	BubbleSort	Com Defeito	Sem Defeito
Implementação 23	HeapSort	Com Defeito	Sem Defeito
Implementação 24	InsertionSort	Com Defeito	Sem Defeito
Implementação 25	MergeSort	Com Defeito	Sem Defeito
Implementação 26	QuickSort	Com Defeito	Sem Defeito
Implementação 27	SelectionSort	Com Defeito	Sem Defeito
Implementação 28	ShellSort	Com Defeito	Sem Defeito
Implementação 29	BubbleSort	Com Defeito	Com Defeito
Implementação 30	HeapSort	Com Defeito	Com Defeito
Implementação 31	InsertionSort	Com Defeito	Com Defeito
Implementação 32	MergeSort	Com Defeito	Com Defeito
Implementação 33	QuickSort	Com Defeito	Com Defeito
Implementação 34	SelectionSort	Com Defeito	Com Defeito
Implementação 35	ShellSort	Com Defeito	Com Defeito
Implementação 36	BubbleSort	Com Defeito	Reduzido
Implementação 37	HeapSort	Com Defeito	Reduzido
Implementação 38	InsertionSort	Com Defeito	Reduzido
Implementação 39	MergeSort	Com Defeito	Reduzido
Implementação 40	QuickSort	Com Defeito	Reduzido
Implementação 41	SelectionSort	Com Defeito	Reduzido
Implementação 42	ShellSort	Com Defeito	Reduzido

Ainda, em algumas implementações, os conjuntos de teste foram reduzidos a um único caso de teste, que por sua vez exercita os programas no melhor caso, ou seja, com uma sequência de números já ordenada $S = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}$.

Com as implementações produzidas, duas validações foram realizadas: (1) a Validação 5, considerando os critérios e ferramentas de teste estrutural; e (2) a Validação 6, que considera os mesmos critérios da Validação 5, porém, também considerando o critérios Análise de Mutantes. Tais validações são detalhadas a seguir.

4.3.1 Validação 5: Validação com Teste Estrutural

Nesta seção é detalhada a Validação 5, a qual visou explorar a avaliação de trabalhos em linguagem C, considerando critérios de teste estruturais.

Definição do Trabalho

Na Validação 5, além da correção avaliada pelo CUNIT (Kumar e St.Clair, 2005), foram considerados os critérios estruturais Todos-Nós (`All-Nodes`) e Todas-Arestas (`All-Edges`), apoiados pela ferramenta GCOV[2].

Para cada um dos critérios estruturais foram atribuídos pesos 1 e para a correção foi atribuído peso 2. Assim, metade da nota do aluno seria reflexo da correção do programa e casos de teste e metade reflexo da qualidade do programa e casos de teste avaliados segundo as coberturas obtidas pelos critérios estruturais.

Análise dos Resultados

A Tabela 4.11 mostra a coberturas e notas obtidas para cada uma das implementações. As implementações de 1 a 7 consistem nos 7 algoritmos de ordenação considerados e nenhum defeito foi inserido no programa e conjunto de teste. Como esperado, a cobertura obtida em cada execução foi de 100% e a nota 10.0 foi sugerida pela PROGTEST.

Por outro lado, nas implementações de 8 a 14, um defeito foi inserido no conjunto de teste. Assim, ao executa-lo, o caso de teste com defeito provoca uma falha nos programas, gerando uma cobertura de 90% em $P_{St_i} - T_{St_i}$ e $P_{Inst} - T_{St_i}$, quando executado com o CUNIT. Tal fato pode ser observado na Figura 4.7, que mostra as coberturas da Implementação 8 para cada um dos critérios de teste. Considerando também os critérios estruturais, cujas coberturas foram de 100%, as coberturas totais para $P_{St_i} - T_{St_i}$ e $P_{Inst} - T_{St_i}$ foram de 95%. A nota sugerida para as implementações foi de 9.67.

As implementações de 15 a 21 possuem o conjunto de teste com um único caso de teste. Ao contrário das implementações anteriores, para este grupo, o algoritmo foi determinante no cálculo da nota sugerida. De fato, para cada algoritmo implementado, diferentes requisitos de teste para os critérios estruturais são gerados. No caso das implementações 16 e 19, o único caso de teste dos conjuntos é suficiente para cobrir todos os requisitos de teste gerados para os algoritmos `HeapSort` e `QuickSort`, gerando uma cobertura de 100% para os critérios

[2]http://gcc.gnu.org/onlinedocs/gcc/Gcov.html

Tabela 4.11: Validação 5: Coberturas e Notas

Implementações	$P_{Inst} - T_{Inst}$	$P_{St_i} - T_{St_i}$	$P_{Inst} - T_{St_i}$	$P_{St_i} - T_{Inst}$	**Nota Sugerida**
1 a 7	100.00%	100%	100%	100%	10.0
8 a 14	100.00%	95%	95%	100%	9.67
15	100.00%	80%	100%	100%	9.33
16	100.00%	100%	100%	100%	10.0
17	100.00%	95.45%	100%	100%	9.85
18	100.00%	91.15%	100%	100%	9.71
19	100.00%	100%	100%	100%	10.0
20	100.00%	97.06%	100%	100%	9.9
21	100.00%	90%	100%	100%	9.67
22 a 28	100.00%	60%	100%	60%	7.33
29 a 35	100.00%	60%	95%	60%	7.17
36 a 38	100.00%	50%	100%	60%	7.0
39	100.00%	43.15%	100%	60%	6.77
40 a 42	100.00%	50%	100%	60%	7.0

GENERAL COVERAGES

Criteria	Pinst-Tinst	Pst-Tst	Pinst-Tst	Pst-Tinst
CUnit/Pass Rate	100.0%	90.0%	90.0%	100.0%
GCov/All-Nodes	100.0%	100.0%	100.0%	100.0%
GCov/All-Edges	100.0%	100.0%	100.0%	100.0%

Figura 4.7: Validação 5: Coberturas Obtidas pela Implementação 8

estruturais e, consequentemente, uma nota 10.0. Por outro lado, no caso da Implementação 15, por exemplo, nenhuma troca é realizada, uma vez que o caso de teste executa os programas com uma sequência de números já ordenada. Assim, os requisitos de teste relacionados ao trecho de código que realiza a troca de dois elementos da sequência não são executados, causando uma menor cobertura para os critérios estruturais, conforme ilustrado na Figura 4.8.

GENERAL COVERAGES

Criteria	Pinst-Tinst	Pst-Tst	Pinst-Tst	Pst-Tinst
CUnit/Pass Rate	100.0%	100.0%	100.0%	100.0%
GCov/All-Nodes	100.0%	60.0%	100.0%	100.0%
GCov/All-Edges	100.0%	60.0%	100.0%	100.0%

Figura 4.8: Validação 5: Coberturas Obtidas pela Implementação 15

A variação das notas é justificada da seguinte maneira. A ideia é verificar se os alunos conseguiram testar os programas que desenvolveram adequadamente. Por um lado, temos o caso das implementações 16 e 19, que por serem melhor estruturadas, precisaram de menos casos de teste para atingir 100% de cobertura. Por outro lado temos casos como o da Implementação 15, cuja nota (9.33), inferior a nota das implementações 16 e 19 (10.0), é justa, uma vez que

representa a situação em que o aluno não conseguiu desenvolver um conjunto de teste que fosse 100%-adequado para o seu programa.

No entanto, uma limitação pode ser observada. A baixa qualidade do conjunto de teste não foi detectada quando executada contra o programa oráculo ($P_{Inst} - T_{St_i}$). Como veremos na Seção 4.3.2, a introdução de critérios de teste mais fortes, como Análise de Mutantes, pode ser fundamental para avaliar de maneira mais rigorosa o conjunto de teste dos alunos. Uma outra solução seria considerar um programa oráculo que para ser testado fornecesse um desafio maior. Assim, mesmo que o conjunto de teste atingissem uma cobertura de 100% em $P_{St_i} - T_{St_i}$, precisariam ser mais forte para atingirem 100% em $P_{Inst} - T_{St_i}$.

Nas implementações de 22 a 28, defeitos no programa foram inseridos. Observando como exemplo as coberturas $P_{St_i} - T_{St_i}$ e $P_{St_i} - T_{Inst}$ obtidas pela Implemetação 22 (Figura 4.9), ambam foram avaliadas em 20% pelo CUNIT, ressaltando os defeitos no programa. Uma vez que coberturas de 100% foram obtidas para os critérios estruturais, as coberturas totais para $P_{St_i} - T_{St_i}$ e $P_{St_i} - T_{Inst}$ foram calculadas em 60% e a nota sugerida às implementações foi de 7.33.

GENERAL COVERAGES

Criteria	Pinst-Tinst	Pst-Tst	Pinst-Tst	Pst-Tinst
CUnit/Pass Rate	100.0%	20.0%	100.0%	20.0%
GCov/All-Nodes	100.0%	100.0%	100.0%	100.0%
GCov/All-Edges	100.0%	100.0%	100.0%	100.0%

Figura 4.9: Validação 5: Coberturas Obtidas pela Implementação 22

Mantendo o defeito nos programas, defeitos no conjunto de teste foram incluídos nas implementações de 29 a 35. Na Figura 4.10 as coberturas para a Implementação 29 podem ser observadas. Uma vez que o defeito no programa foi mantido, a correção avaliada pelo CUNIT para $P_{St_i} - T_{St_i}$ e $P_{St_i} - T_{Inst}$ manteve-se em 20%. Adicionalmente, a cobertura $P_{Inst} - T_{St_i}$ foi avaliada em 90%, identificando o defeito no conjunto de teste. Assim, as coberturas totais para $P_{St_i} - T_{St_i}$ e $P_{St_i} - T_{Inst}$ também foram calculadas em 60%. As coberturas totais para $P_{Inst} - T_{St_i}$ foram de 95% e a nota sugerida para os trabalhos foi de 9.17.

GENERAL COVERAGES

Criteria	Pinst-Tinst	Pst-Tst	Pinst-Tst	Pst-Tinst
CUnit/Pass Rate	100.0%	20.0%	90.0%	20.0%
GCov/All-Nodes	100.0%	100.0%	100.0%	100.0%
GCov/All-Edges	100.0%	100.0%	100.0%	100.0%

Figura 4.10: Validação 5: Coberturas Obtidas pela Implementação 29

Por fim, nas implementações de 36 a 42, os defeitos nos programas foram mantidos e o conjunto de teste reduzido a um único caso de teste. Comparando as implementações de 36 a 42 com as implementações de 15 a 21, que também tiveram seus conjuntos de teste reduzidos a um único caso de teste, temos que nas implementações de 15 a 21, a maioria dos programas não atingiram 100% de cobertura para os critérios estruturais em $P_{St_i} - T_{St_i}$. Isto ocorre porque o caso de teste considerado executa os programas com uma sequência já ordenada de números. Sendo assim, os trechos de código referentes a troca de elementos na sequência não são executados.

Nas implementações de 36 a 42, o mesmo caso de teste é utilizado. No entanto, uma vez que os programas possuem defeitos e realizam a ordenação inversa das sequências, os trechos de código responsáveis em trocar elementos na sequência são executados, resultando em uma cobertura de 100% para os critérios estruturais.

A Figura 4.11 mostra as coberturas obtidas para a Implementação 36. A cobertura $P_{St_i} - T_{Inst}$ continuou sendo avaliada em 20% pelo CUNIT, identificando os defeitos no programa do aluno. Já a cobertura $P_{St_i} - T_{St_i}$ foi de 0.0%, uma vez que o único caso de teste do conjunto falhou. Quanto aos critérios estruturais a cobertura foi de 100%.

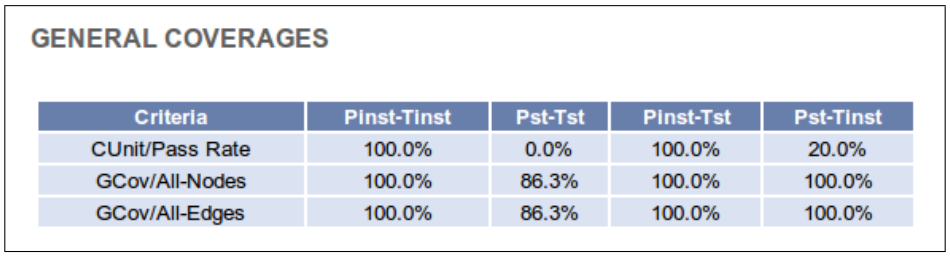
GENERAL COVERAGES

Criteria	Pinst-Tinst	Pst-Tst	Pinst-Tst	Pst-Tinst
CUnit/Pass Rate	100.0%	0.0%	100.0%	20.0%
GCov/All-Nodes	100.0%	100.0%	100.0%	100.0%
GCov/All-Edges	100.0%	100.0%	100.0%	100.0%

Figura 4.11: Validação 5: Coberturas Obtidas pela Implementação 36

Basicamente, das 7 implementações, apenas a Implementação 39 conportou-se de maneira diferente. Observando as coberturas para a Implementação 39 (Figura 4.12) é possível observar que o resultado das execuções no CUNIT foi o mesmo avaliado para a Implementação 36. No entanto, a cobertura $P_{St_i} - T_{St_i}$ foi de 86.3% para os critérios estruturais. Nesse sentido a cobertura total da Implementação 39 para $P_{St_i} - T_{St_i}$ foi de 43.15% contra as coberturas de 50% das implementações de 36 a 38 e 40 a 42. A nota final da Implementação 39 foi de 6.77 e a das demais implementações 7.0.

De maneira geral, foi possível perceber que a PROGTEST é capaz de identificar adequadamente os defeitos, embora a utilização de critérios de teste mais rigorosos ou de um programa oráculo mais desafiador seja fundamental para avaliar a qualidade dos casos de teste.



GENERAL COVERAGES

Criteria	Pinst-Tinst	Pst-Tst	Pinst-Tst	Pst-Tinst
CUnit/Pass Rate	100.0%	0.0%	100.0%	20.0%
GCov/All-Nodes	100.0%	86.3%	100.0%	100.0%
GCov/All-Edges	100.0%	86.3%	100.0%	100.0%

Figura 4.12: Validação 5: Coberturas Obtidas pela Implementação 39

4.3.2 Validação 6: Validação com Teste Baseado em Erros

Nesta seção é detalhada a Validação 6, a qual visou explorar a avaliação de trabalhos em linguagem C, considerando critérios de teste estruturais e o critério Análise de Mutantes.

Definição do Trabalho

Na validação 6, as implementações foram resubmetidas à PROGTEST para serem avaliados considerando, além da correção e teste estrutural, o critério Análise de Mutantes, apoiado pela ferramenta PROTEUM (Delamaro et al., 1993).

O conjunto de operadores utilizados para a aplicação da Análise de Mutantes consistem nos seguintes operadores essenciais identificados no trabalhos de (Barbosa, 1998): (1) `Logical Operator by Bitwise Operator (OLBN)`, substitui operadores cada operador lógico por operadores bit-a-bit; (2) `Relational Operator Mutation (ORRN)`, substitui cada operador relacional por outros operadores relacionais; (3) `n-trip continue (SMTC)` interrompe execução de laços após duas execuções; (4) `Statement Delection (SSDL)` deleta cada comando ou bloco de comandos; (5) `while Replacement by do-while (SWDD)` substitui cada comando `while` por *do-while*; (6) `Domain Traps (VDTR)` força cada valor escalar assumir valores negativo, zero e positivo; (7) `Twiddle Mutations (VTWD)`, substitui cada valor escalar pelo seu sucessor e predecessor; (8) `Constant for Constant Replacement (Cccr)`, substitui cada constante por outras constantes; e (9) `Constant for Scalar Replacement (Ccsr)`, substitui cada constante por valores escalares.

De maneira semelhante às demais validações, para cada um dos critérios estruturais foram atribuídos pesos 1, para a correção foi atribuído peso 2 e para a Análise de Mutantes foi atribuído peso 2.

Análise dos Resultados

A Tabela 4.12 mostra as coberturas e notas obtidas para cada implementação. De início, observa-se que tanto o trabalho oráculo quanto as implementações de 1 a 7, em que nenhum defeito foi inserido e cujos conjuntos de teste eram relativamente “fortes”, não atingem coberturas iguais a 100% e, consequentemente, notas iguais a 10.0. Tal fato ocorre devido à presença

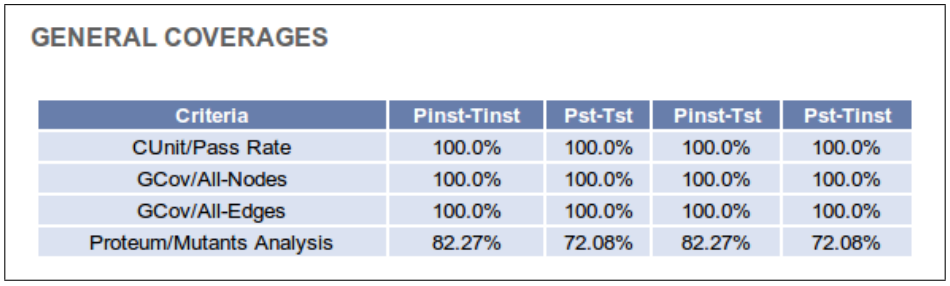
de mutantes equivalentes, ou seja, mutantes que não podem ser mortos ao aplicar a Análise de Mutantes, contabilizando um escore de mutação menor que 100% para os trabalhos.

Tabela 4.12: Validação 6: Coberturas e Notas

Implementação	$P_{Inst} - T_{Inst}$	$P_{St_i} - T_{St_i}$	$P_{Inst} - T_{St_i}$	$P_{St_i} - T_{Inst}$	**Nota Sugerida**
1	94.09%	90.69%	94.09%	90.69%	9.38
2	94.09%	93.09%	94.09%	93.09%	9.54
3	94.09%	92.44%	94.09%	92.44%	9.5
4	94.09%	91.91%	94.09%	91.91%	9.46
5	94.09%	94.09%	94.09%	94.09%	9.61
6	94.09%	91.16%	94.09%	91.16%	9.41
7	94.09%	92.35%	94.09%	92.35%	9.49
8	94.09%	87.36%	90.76%	90.69%	9.15
9	94.09%	89.75%	90.76%	93.09%	9.31
10	94.09%	89.11%	90.76%	94.09%	9.27
11	94.09%	88.87%	90.76%	91.91%	9.24
12	94.09%	90.76%	90.76%	94.09%	9.38
13	94.09%	87.83%	90.76%	91.16%	9.18
14	94.09%	89.02%	90.76%	92.35%	9.26
15	94.09%	59.39%	85.56%	90.69%	8.03
16	94.09%	90.2%	85.56%	93.09%	9.14
17	94.09%	73.19%	85.56%	92.44%	8.55
18	94.09%	74.8%	85.56%	91.91%	8.59
19	94.09%	85.56%	85.56%	94.09%	9.02
20	94.09%	76.95%	85.56%	91.16%	8.64
21	94.09%	71.98%	85.56%	92.35%	8.51
22	94.09%	63.59%	94.09%	63.59%	7.57
23	94.09%	63.12%	94.09%	63.12%	7.54
24	94.09%	65.33%	94.09%	65.33%	7.69
25	94.09%	73.33%	94.09%	73.33%	8.22
26	94.09%	65.19%	94.09%	65.19%	7.68
27	94.09%	64.9%	94.09%	64.9%	7.66
28	94.09%	63.95%	94.09%	63.95%	7.6
29	94.09%	63.59%	90.76%	63.59%	7.45
30	94.09%	63.53%	90.76%	63.12%	7.44
31	94.09%	65.56%	90.76%	65.33%	7.58
32	94.09%	73.33%	90.76%	73.33%	8.1
33	94.09%	65.48%	90.76%	65.29%	7.57
34	94.09%	64.9%	90.76%	64.9%	7.54
35	94.09%	63.95%	90.76%	63.95%	7.48
36	94.09%	54.33%	85.56%	63.59%	6.96
37	94.09%	46.65%	85.56%	63.12%	6.69
38	94.09%	54.0%	85.56%	65.33%	7.01
39	94.09%	51.06%	85.56%	73.33%	7.18
40	94.09%	54.46%	85.56%	65.19%	7.02
41	94.09%	53.21%	85.56%	64.9%	6.97
42	94.09%	54.83%	85.56%	63.9%	6.99

Observando, por exemplo, a Implementação 1, cujas coberturas são apresentadas na Figura 4.13, observa-se que as coberturas avaliadas pelo CUNIT e pelo GCOV permanecem em 100%. No entanto, os escores de mutação só atingem 82.27% para o programa oráculo e 72.08% para

o programa da implementação. Com escores de mutação menores que 100.0%, as coberturas totais calculadas para a implementação são de 90.69%, 94.09% e 90.69% para $P_{St_i} - T_{St_i}$, $P_{Inst} - T_{St_i}$ e $P_{St_i} - T_{Inst}$, respectivamente.



GENERAL COVERAGES

Criteria	Pinst-Tinst	Pst-Tst	Pinst-Tst	Pst-Tinst
CUnit/Pass Rate	100.0%	100.0%	100.0%	100.0%
GCov/All-Nodes	100.0%	100.0%	100.0%	100.0%
GCov/All-Edges	100.0%	100.0%	100.0%	100.0%
Proteum/Mutants Analysis	82.27%	72.08%	82.27%	72.08%

Figura 4.13: Validação 6: Coberturas Obtidas pela Implementação 1

A fim de amenizar tal limitação, a cobertura obtida para o trabalho oráculo ($P_{Inst} - T_{Inst}$) é utilizada. Uma vez que o trabalho oráculo deve representar a solução correta para o problema, a cobertura calculada para $P_{Inst} - T_{Inst}$ é assumida como a cobertura máxima que pode ser atingida para o programa do professor. Assim, durante o calculo da nota final, a cobertura em $P_{Inst} - T_{St_i}$ é ajustada pela PROGTEST em função da cobertura de $P_{Inst} - T_{Inst}$. Para a Implementação 1, por exemplo, tem-se que:

$$P_{Inst} - T_{St_i ajustada} = \frac{P_{Inst} - T_{St_i}}{P_{Inst} - T_{Inst}} = \frac{0.9409}{0.9409} = 1(100\%)$$

Dessa forma, o fato da Implementação 1 ter obtido uma cobertura de 94.09% em $P_{Inst} - T_{St_i}$, não reduz a nota, uma vez que seu conjunto de teste conseguiu atingir a cobertura máxima possível para o programa do oráculo.

No entanto, tal ajuste não pode ser realizado com as coberturas $P_{St_i} - T_{St_i}$ e $P_{St_i} - T_{Inst}$, pois não é possível determinar automaticamente se as coberturas de 90.69% em $P_{St_i} - T_{St_i}$ e $P_{St_i} - T_{Inst}$ correspondem a cobertura máxima possível no programa ou se são os conjuntos de teste que não foram "fortes" o suficiente para atingir uma cobertura maior. Por este motivo, as coberturas em $P_{St_i} - T_{St_i}$ e $P_{St_i} - T_{Inst}$ são mantidas e a implementação recebe uma nota de 9.38. Cabe, então, ao professor, posteriormente analisar a veracidade dos mutantes equivalentes reportados e atribuir à nota os pontos associados.

Ainda em relação aos mutantes equivalentes, um outro aspecto que pode ser observado é a diferença na nota final das implementações de acordo com o algoritmo. Considerando as implementações de 1 a 7, por exemplo, as notas variam de 9.38 a 9.61. Tal variação ocorre em virtude de que a porcentagem de mutantes equivalentes gerados é diferente para cada algoritmo.

Apesar das limitações observadas, é possível perceber que a avaliação continua consistente conforme os defeitos e alterações realizadas nos programas e conjuntos de teste. Ao comparar, por exemplo, as coberturas e notas das implementações de 1 a 7 com as coberturas e notas das

implementações de 8 a 14, observa-se que nas implementações de 8 a 14, cujos conjuntos de teste possuem defeitos, houve uma queda nas coberturas $P_{St_i} - T_{St_i}$ e $P_{Inst} - T_{St_i}$ e nas notas finais. A Implementação 8, por exemplo, obteve cobertura de 87.36% para $P_{St_i} - T_{St_i}$ e de 90.76% para $P_{Inst} - T_{St_i}$, contra as coberturas de 90.69% e 94.09% obtidas pela Implementação 1, que consiste no mesmo algoritmo.

Ressalta-se que, de forma semelhante à Implementação 1, durante o cálculo da nota da Implementação 8, a cobertura $P_{Inst} - T_{St_i}$ foi ajustada da seguinte forma:

$$P_{Inst} - T_{St_i ajustada} = \frac{P_{Inst} - T_{St_i}}{P_{Inst} - T_{Inst}} = \frac{0.9076}{0.9409} = 0.9646(96.46\%)$$

Assim, somente a diferença entre a cobertura máxima possível no programa oráculo e a cobertura obtida pelo conjunto de teste da implementação foi descontada da nota final. A nota sugerida para a Implementação 8 foi de 9.15, contra a nota 9.38 da Implementação 1, ressaltando o defeito no conjunto de teste da Implementação 8.

Da mesma forma, comparando, por exemplo, as implementações de 1 a 7 com as implementações de 22 a 28, as quais possuem defeitos no programa, é possível perceber que houve um decréscimo nas coberturas $P_{St_i} - T_{St_i}$ e $P_{St_i} - T_{Inst}$. A Implementação 22, por exemplo, possui uma cobertura de 63.59% em $P_{St_i} - T_{St_i}$ e $P_{St_i} - T_{Inst}$, contra uma cobertura de 90.69% da Implementação 1, que consiste no mesmo algoritmo.

A nota sugerida para a Implementação 22 foi de 7.57, contra 9.38 sugerida para a Implementação 1, ressaltando os defeitos no program da Implementação 22.

Uma das vantagens observadas em utilizar a Análise de Mutantes foi a avaliação mais rigorosa do conjunto de teste dos alunos. Considerando as implementações de 15 a 21 e 36 a 42, cujos conjuntos de teste foram reduzidos a um único caso de teste, na Validação 5, a qualidade "baixa" dos casos de teste não foram identificadas pela PROGTEST, uma vez que critérios relativamente "fracos" estavam sendo utilizados.

Por outro lado, com a Análise de Mutantes, a cobertura $P_{Inst} - T_{St_i}$, que avalia a qualidade do conjunto de teste, foi reduzida. Para as implementações de 15 a 21 e 36 a 42, a cobertura foi calculada em 85.56%, inferior à cobertura de 90.76% calculada para as implementações com defeitos no conjuto de teste (8 a 14 e 29 a 35). Esta, por sua vez, é inferior à cobertura de 94.09%, calculada para as implementações sem defeitos no conjunto de teste (1 a 7 e 22 a 28).

4.4 Limitações e Melhorias

Na seção anterior, validações do ambiente PROGTEST foram realizadas. Durante a execução e análise dos resultados, algumas limitações do ambiente PROGTEST puderam ser identificadas. Nesta seção, tais limitações são sintetizadas.

Requisitos Não Executáveis

Um aspecto observado durante as validações refere-se aos requisitos não executáveis (considerando os critérios estruturais) e mutantes equivalentes (considerando o critério Análise de Mutantes). Uma vez que tais requisitos de teste não podem ser satisfeitos, o conjunto de teste projetado para o programa em questão nunca atingirá 100% de adequação ao critério de teste em questão e, consequentemente, o aluno nunca irá conseguir atingir a nota máxima.

Dentre as abordagens que poderiam ser exploradas a fim de minimizar tal limitação, encontra-se a execução dos conjuntos de teste de todos os alunos contra o programa, para estimar a probabilidade de um determinado requisito de teste não coberto ser não executável. A ideia é que, quanto mais conjuntos de teste forem executados contra o programa em questão e quanto mais fortes forem esses conjuntos de teste, maior será a probabilidade do requisito não coberto também ser não executável.

Outra abordagem que pode ser explorada é permitir que os alunos selecionem na PROGTEST os requisitos não cobertos que identificaram como não executáveis. A PROGTEST calcularia, então, uma sugestão de nota provisória, considerando os requisitos marcados como não executáveis. O professor ficaria a cargo de, via PROGTEST, verificar a veracidade dos requisios selecionados como não executáveis e reportar quando algum requisito identificado pelo aluno como não executável for executável. Após isso, a nota final pode ser sugerida pela PROGTEST considerando as correções realizadas pelo professor.

A primeira abordagem teria como vantagem prover um método automático para identificar os requisitos não executáveis, embora não forneceria um resultado preciso sobre os requisitos não executável. Ainda, aspectos relacionados a desempenho e sincronização devem ser avaliados com cuidado ao explorar tal abordagem. Por outro lado, a segunda abordagem não forneceria um meio completamente automático para considerar os requisitos não executável, mas permitiria refletir a situação real dos requisitos não executáveis nos programas do aluno.

Interface entre Programas e Casos de Teste

Uma vez que os programas e conjuntos de teste do professor e dos alunos são trocados param serem executados uns com os outros, é fundamental que a interface entre o programa do aluno e seus casos de teste sejam identicas as interfaces entre o programa do professor e seus casos de teste.

Embora nas validações com os alunos de pós-graduação e graduação, a interface entre programas e casos de teste tenha sido detalhadamente especificada no enunciado fornecido aos alunos, em ambos grupos, alguns alunos tiveram dificuldades em implementar os seus trabalhos com a interface definida.

Neste sentido, a existência de uma interface comum entre os programas e conjuntos de teste do professor e dos alunos é um aspecto que deve ser melhor investigado. Uma abordagem que

pode ser explorada seria introduzir na PROGTEST a possibilidade do professor fornecer, durante a criação de um trabalho, arquivos que definem a interface do programa a ser implementado pelos alunos. Por sua vez, os alunos deverão realizar o *download* da interface fornecida pelo professor, completando-a com a sua implementação.

Os arquivos que definem a interface do programa poderia conter tanto as assinaturas dos métodos a serem implementados e testados pelos alunos como, também, métodos já implementados e prontos para os alunos utilizarem em conjunto com suas implementações.

No entanto, ressalta-se que melhorias na interface da PROGTEST também poderiam ser exploradas a fim de prover ao aluno um *feedback* mais adequado sobre os problemas na interface de seus programas e casos de teste.

Relatórios sobre as Execuções Trocadas

Durante as validações com os alunos de pós-graduação e com os alunos de graduação, um dos aspectos questionados pelos alunos e reportados por eles em seus relatórios foi sobre a possibilidade de visualizar informações detalhadas (tais como, grafos de fluxo de controle, casos de teste que falharam, etc.) sobre os resultados da execução de seus casos de teste no programa do professor ($P_{Inst} - T_{St_i}$) e sobre a execução dos casos de teste do professor em seus programas ($P_{St_i} - T_{Inst}$). Tais informações, em alguns casos, são essenciais para que os alunos consigam melhorar seus programas e conjuntos de teste e, consequentemente, atingirem nota máxima.

Nesse sentido, a visualização de relatórios sobre a execução dos casos de teste do professor no programa do aluno ($P_{St_i} - T_{Inst}$) foi introduzida na PROGTEST após a validação com os alunos de pós-graduação e foram essenciais para que os alunos de graduação pudessem identificar os problemas em seus programas.

Em relação às informações sobre a execução de seus conjuntos de teste contra o programa do professor ($P_{Inst} - T_{St_i}$), a cobertura obtida para os critérios estruturais, por exemplo, pode não ser de 100%, mesmo que o conjunto de teste obtenha uma cobertura de 100% para o seu programa. Neste caso, informações tais como o grafo de fluxo de controle do programa do professor são essenciais para que o aluno consiga projetar novos casos de teste para cobrir 100% do programa do professor.

No entanto, fornecer informações sobre o programa e conjunto de teste do professor pode prejudicar o processo de aprendizagem. Uma vez que os alunos tem conhecimento sobre como o trabalho oráculo foi desenvolvido, sua tendência é implementar seus programas e casos de teste da mesma forma, ao invés de projetarem os seus própios algoritmos e casos de teste.

Afim de amenizar tal limitação foi acrescentado às propriedades do trabalho opções para que o professor defina se o aluno irá visualizar: (1) os relatórios gerados a partir da execução do seu conjunto de teste no seu programa ($P_{St_i} - T_{St_i}$); (2) os relatórios gerados a partir da

execução do seu conjunto de teste no programa do professor ($P_{Inst} - T_{St_i}$); e (3) os relatórios gerados a partir da execução do conjunto de teste do professor no seu programa ($P_{St_i} - T_{Inst}$).

Assim, fica a cargo do professor definir se é conveniente que um aluno visualize cada grupo de relatórios. No entanto, mecanismos capazes de permitir que o professor defina com mais detalhes quais informações deverão ser disponibilizadas aos alunos ainda devem ser melhor explorados.

Uma abordagem interessante seria liberar informações aos alunos de acordo com seu progresso nas atividades de depuração e teste, a exemplo do que ocorre na MARMOSET (Seção 2.3.2.2) (Spacco et al., 2006a,b). Informações sobre os casos de teste do professor, por exemplo, poderiam somente ser liberadas a um aluno quando o conjunto de teste dele atingisse 100% de adequação para o seu programa em cada um dos critérios de teste considerados.

Teste Estrutural e Baseado em Erros no Programa Oráculo

Quando um aluno desenvolve um conjunto de teste para o seu programa considerando os critérios estruturais e baseados em erros, ao atingir 100% de adequação aos critérios estruturais para o seu programa, em alguns casos, o conjunto de teste pode ainda não ser 100% adequado aos critérios para o programa do professor.

De fato, exigir que o conjunto de teste do aluno seja 100% adequado ao programa do professor pode estimular os alunos a desenvolverem conjuntos de teste mais fortes. No entanto, mesmo sem conseguir 100% de adequação para o programa do professor, ele realizou uma boa atividade de teste, pois conseguiu adequar o seu conjunto de teste 100% para o seu programa.

De maneira semelhante, se o conjunto de teste do professor não for 100% adequado aos critérios estruturais e baseados em erros para o programa do aluno, não significa necessariamente que há problemas no programa do aluno, mas sim, que é possível, do ponto de vista de teste de software, implementar o programa de uma forma melhor, ou seja, de uma forma em que o programa possa ser testado mais facilmente.

Neste sentido, em vista das observações apresentadas, de acordo com a abordagem adotada pelo professor, ele pode não achar conveniente considerar na avaliação as coberturas $P_{Inst} - T_{St_i}$ e $P_{St_i} - T_{Inst}$ para os critérios estruturais e baseados em erros.

Em vista de tal situação, foram adicionadas às propriedades dos trabalhos na PROGTEST, a possibilidade de definir exatamente quais coberturas a PROGTEST deverá considerar na avaliação. Nelas, o professor pode selecionar, para cada critério de teste, quais coberturas deverão ser consideradas na avaliação.

Se, para os critérios estruturais, por exemplo, somente os *checkbox* da coluna $P_{St_i} - TSt_i$ forem selecionados, as coberturas obtidas para o conjunto de teste do aluno no programa do professor ($P_{Inst} - T_{St_i}$) e as coberturas do conjunto de teste do professor para o programa do aluno ($P_{St_i} - T_{Inst}$), para os critérios estruturais, não serão consideradas na avaliação.

Aspectos Relacionados à Avaliação

A PROGTEST fornece diversas opções que permitem que o professor possa controlar quais aspectos do trabalho do aluno serão avaliados e o peso que cada aspecto terá na avaliação.

Uma dessas opções refere-se a escolha dos critérios de teste e ferramentas associadas a serem considerados na avaliação. Com os *frameworks* de teste (JUNIT e CUNIT) é possível avaliar a correção dos programas e casos de teste. Já com os critérios estruturais e baseados em erros é possivel avaliar a qualidade dos programas e conjuntos de teste. Ainda, com o uso critérios mais fortes, como a Análise de Mutantes, é possível deixar a avaliação mais rigorosa, enquanto o uso de somente critérios mais fracos, como o Todos-Nós, permite uma avaliação menos rigorosa.

Outra opção refere-se aos pesos atribuídos a cada critério. Pesos maiores para critérios maiores resultam em uma avaliação mais rigorosa, enquanto pesos maiores para critérios mais fracos resultam em uma avaliação menos rigorosa. Por outro lado, pesos maiores para o teste funcional farão com que a correção dos programas e conjuntos de teste tenham maior impacto na nota sugerida pela PROGTEST, enquanto para os critérios estruturais e baseados em erros farão com que a qualidade dos programas e casos de teste tenham maior impacto.

Por sua vez, o professor pode definir, para cada critério, quais coberturas deverão ser consideradas na avaliação. A cobertura $P_{St_i} - T_{St_i}$ refere-se à correção e qualidade da atividade de teste realizada pelo aluno; a cobertura $P_{Inst} - T_{St_i}$ refere-se à correção e qualidade do conjunto de teste do aluno; e a cobertura $P_{St_i} - T_{Inst}$ refere-se à correção e qualidade do programa do aluno. Assim, a utilização ou não de cada uma das coberturas define quais aspectos do trabalho serão avaliados. Os pesos e os critérios definidos para cada cobertura definem o impacto que cada aspecto do trabalho terá na nota final.

Por fim, observa-se que as características de implementação do trabalho oráculo também podem interferir no resultado da avaliação. Um programa oráculo menos modularizado/estruturado pode ser mais difícil de ser testado, exigindo que os alunos implementem conjuntos de teste mais fortes para conseguir 100% de adequação em $P_{Inst} - T_{St_i}$. Por outro lado, conjuntos de teste oráculos mais fortes e bem projetados permitem avaliar os programas dos alunos com mais rigor, aumentando a chances de encontrar defeitos nos programas dos alunos.

Ressalta-se que tais opções e as suas combinações devem ser melhor exploradas, a fim de prover aos professores uma ideia mais clara e objetiva sobre como configurar a PROGTEST para obter a avaliação desejada.

Aspectos Relacionados à Interface

Por meio do *feedback* fornecidos pelos alunos de pós-graduação e graduação, foi possível perceber que os alunos tiveram dificuldades em interpretar os resultados fornecidos pelo relatório geral de avaliação fornecido pela PROGTEST. Ainda, alguns alunos reportaram a dificul-

dade em encontrar determinadas informações que estavam presentes em relatórios específicos de uma determinada ferramenta as quais eram importantes para prosseguirem com a atividade de depuração e teste.

Neste sentido, como interpretar automaticamente os resultados e de como apresenta-los aos alunos também é um aspecto importante a ser melhor explorado. O fornecimento de informações mais claras e objetivas aos alunos é fundamental para que eles possam identificar e entender os problemas em seus programas e casos de teste, além de incentiva-los a continuarem com as atividades de teste e depuração.

4.5 Considerações Finais

Neste capítulo foram apresentados aspectos relacionados à validação do ambiente PROGTEST. Em linhas gerais, foram consideradas a validação do ambiente PROGTEST tanto em ambientes controlados como em cenários reais de ensino. Ainda, validações foram realizadas a fim de aplicar o ambiente PROGTEST considerando o teste baseado em erros e programas em C.

Além disso, a partir das validações realizadas foi possível identificar as principais limitações e possibilidades de melhorias no ambiente PROGTEST.

No próximo capítulo são sintetizadas as principais contribuições fornecidas por este trabalhos e as atividades a serem realizadas em trabalhos futuros.

CAPÍTULO

5

Conclusão e Trabalhos Futuros

5.1 Visão Geral

O ensino de fundamentos de programação não é uma tarefa trivial – muitos estudantes têm dificuldades em compreender os conceitos abstratos de programação (Lahtinen et al., 2005) e possuem visões erradas sobre a atividade de programação (Edwards, 2004).

Dentre as iniciativas investigadas a fim de amenizar os problemas associados, destaca-se o ensino conjunto de conceitos básicos de programação e de teste de software. A introdução da atividade de teste pode ajudar o desenvolvimento das habilidades de compreensão e análise nos estudantes, já que para sua condução é necessário que os alunos conheçam o comportamento dos seus programas (Edwards, 2004). Além disso, experiências recentes têm sugerido que a atividade de teste poderia ser ensinada o mais cedo possível. A ideia é que alunos que aprendem teste mais cedo podem se tornar melhores testadores e desenvolvedores uma vez que o teste força a integração e aplicação de teorias e habilidades de análise, projeto e implementação (Barbosa et al., 2003; Jones, 2001; Patterson et al., 2003).

Como mecanismo para apoiar o ensino integrado de fundamentos de programação e teste de software, foi proposta a PROGTEST– um ambiente para submissão e avaliação automática de trabalhos práticos de programação dos alunos, baseado em atividades de teste de software.

Em linhas gerais, o ambiente PROGTEST permite que os alunos submetam seus trabalhos práticos de programação. Utilizando ferramentas de teste integradas, o ambiente automaticamente avalia o trabalho dos alunos, fornecendo a eles um *feedback* imediato sobre os seus trabalhos. Em sua primeira versão, estavam integrados ao ambiente PROGTEST: (1) o JUNIT,

apoiando a execução automática de casos de teste; e (2) a ferramenta JABUTISERVICE, fornecendo apoio a aplicação de critérios de teste estrutural; ambos apoiando o teste de programas escritos em Java.

Este trabalho visou a dar continuidade ao desenvolvimento do ambiente PROGTEST, tendo como principal objetivo investigar a integração de novas ferramentas de teste ao ambiente. Tal integração foi explorada a fim de propiciar a utilização, por parte dos alunos, de diferentes técnicas e critérios na condução de seus testes. Além disso, a submissão e avaliação de programas em diferentes linguagens de programação também foi considerada e incorporadas ao ambiente.

Com base no estudo e análise de diferentes ferramentas de teste, a PROGTEST foi evoluída a fim de torná-la extensível para a integração de novas ferramentas. Embora duas ferramentas de teste já estivessem integradas ao ambiente PROGTEST, percebeu-se que o ambiente não era extensível o suficiente para integrar as demais ferramentas investigadas. Além disso, o código responsável pela execução do JUNIT e da ferramenta JABUTISERVICE estava bastante dependente do código da PROGTEST e era específico para cada uma destas ferramentas, contrariando a ideia de um ambiente em que diversas ferramentas de teste pudessem ser integradas.

Após a evolução, a PROGTEST tornou-se capaz de aceitar diferentes ferramentas na forma de *plugins*. Neste sentido, *plugins* foram construídos para as ferramentas JUNIT e JABUTISERVICE. Adicionalmente, um *plugin* foi construído para a ferramenta JUMBLE, fornecendo apoio ao critério Análise de Mutantes para programas escritos em linguagem Java.

A fim de introduzir na PROGTEST apoio a diferentes linguagens de programação, também foram construídos *plugins* para as ferramentas CUNIT, GCOV e PROTEUM, que apoiam o teste de programas escritos em linguagem C. Respectivamente, as ferramentas fornecem apoio: (1) à construção e execução automática de casos de teste; (2) à aplicação de critérios de teste estrutural – Todos-Nós e Todas-Arestas; e (3) à aplicação de critérios baseados em erros – Análise de Mutantes.

Por fim, o ambiente PROGTEST foi aplicado e validado. Inicialmente, validações foram realizadas considerando programas escritos em linguagem Java. Uma validação preliminar com programas em Java já havia sido realizada no projeto de iniciação científica conduzido pelo autor deste trabalho (Processo FAPESP 09/00006-3). Assim, neste trabalho, a PROGTEST foi aplicada e validada em ambientes reais de ensino, considerando a utilização das ferramentas JUNIT e JABUTISERVICE integradas à ela. Além disso, a fim de verificar o comportamento da PROGTEST com teste baseado em erros, os trabalhos desenvolvidos pelos alunos foram ressubmetidos à PROGTEST, considerando, além das ferramentas JUNIT e JABUTISERVICE, a utilização da ferramenta JUMBLE.

No contexto de linguagem C, uma validação preliminar foi também foi realizada. Um conjunto de trabalhos foram construídos, sendo que alguns possuíam problemas em seus programas e/ou casos de teste. Tais trabalhos foram submetidos à PROGTEST considerando as ferramentas

CUNIT, GCOV e PROTEUM. Os resultados foram analisados a fim de verificar o impacto que cada problema presente em um trabalho causa na avaliação realizada pela PROGTEST.

5.2 Contribuições de Pesquisa

Dentre as principais contribuições de pesquisa obtidas a partir da realização deste trabalho, destacam-se:

- Evolução da PROGTEST a fim de torná-la extensível para a integração de diferentes ferramentas de teste;
- Introdução de critérios de teste da técnica baseada em erros, em especial, o critério Análise de Mutantes;
- Integração de novas ferramentas de teste à PROGTEST, provendo ao ambiente apoio a submissão e avaliação automática de trabalhos práticos de programação em diferentes linguagens de programação, em especial, C e Java;
- Evolução da base de trabalhos oráculo em Java da PROGTEST e desenvolvimento de uma base de trabalhos em C;
- Validação do ambiente PROGTEST, identificando suas principais vantagens, limitações e pontos de melhoria.

5.3 Trabalhos Futuros

Como discutido na Seção 4.4, a partir das validações realizadas, foram identificados diversos aspectos do ambiente PROGTEST que podem ser melhor investigados em trabalhos futuros. Dentre tais aspectos, destacam-se:

- Desenvolvimento de mecanismos para o tratamento de requisitos não executáveis;
- Desenvolvimento de mecanismos que forneçam aos alunos uma ideia mais clara sobre como a interface entre seus programas e casos de teste deve ser desenvolvida;
- Desenvolvimento de mecanismos que permitam os professores configurar quais informações do trabalho oráculo poderão ser visualizadas pelos alunos e em quais situações será permitida sua visualização;
- Estudo sobre como as diversas opções de configuração de avaliação da PROGTEST podem ser combinadas e como cada combinação pode impactar o resultado da avaliação;

- Estudo sobre como melhorar o *feedback* fornecido aos alunos;
- Desenvolvimento de relatórios que possam fornecer *feedback* mais claro aos alunos a respeito da situação de seus trabalhos.

Ainda, em continuação as atividades conduzidas neste trabalhos, as seguintes atividades devem ser realizadas:

- Integração do ambiente PROGTEST em ambientes virtuais de aprendizagem, tais como MOODLE e SAKAI.
- Estudo de viabilidade e desenvolvimento de novos *plugins* ao ambiente PROGTEST, considerando outros critérios de teste (tais como critérios funcionais) e outras linguagens de programação (tais como, Pascal e C++);
- Planejamento e condução de experimentos controlados e sistemáticos envolvendo a utilização do ambiente PROGTEST em ambientes reais de ensino. Em especial, está sendo planejado um experimento envolvendo a aplicação da PROGTEST na disciplina de Introdução à Ciências da Computação. Este experimento deverá ser conduzido no 1º semestre de 2012.

5.4 Produção Científica

Por fim, como parte do resultados obtidos a partir da realização deste trabalho, destacam-se as seguintes publicações:

- SOUZA, D. M.; MALDONADO, J. C.; BARBOSA, E. F.. **ProgTest: An Environment for the Submission and Evaluation of Programming Assignments based on Testing Activities.** In: *24th Conference on Software Engineering Education and Training (CSEET 2011)*, Maio 2011, Honolulu (Hawaii), EUA.
- SOUZA, D. M.; BARBOSA, E. F.; MALDONADO, J. C.. **Uma Contribuição à Submissão e Avaliação Automática de Trabalhos de Programação com base em Atividades de Teste.** In: *II Congresso Brasileiro de Software: Teoria e Prática (CBSoft 2011) - XXV Simpósio Brasileiro de Engenharia de Software (SBES 2011) - XVIII Sessão de Ferramentas*, Setembro 2011, São Paulo.
- SOUZA, D. M.; MALDONADO, J. C.; BARBOSA, E. F.. **ProgTest: Apoio Automatizado ao Ensino de Programação e Teste de Software**. In: *22º Simpósio Brasileiro de Informática na Educação - 17º Workshop de Informática na Escola (SBIE - WIE)*, 2011, Aracaju - SE.

Além disso, o seguinte artigo foi submetido ao periódico *IEEE Transactions on Educations*:

- SOUZA, D. M.; BARBOSA, E. F.; SOUZA, S. R. S.; OLIVEIRA, B. H.; MALDONADO, J. C.. **Automated Evaluation of Programming Assignments based on Testing Activities**, 8p., *IEEE Transactions on Educations*, 2012.

Ressalta-se que outros artigos vêm sendo elaborados e deverão ser submetidos a conferências e periódicos de qualidade.

Referências

Acree, A. T.; Budd, T. A.; DeMillo, R. A.; Lipton, R. J.; Sayward, F. G. *Mutation analysis*. Relatório Técnico GIT-ICS-79/08, 1979.

Allen, E.; Cartwright, R.; Stoler, B. DrJava: A lightweight pedagogic environment for java. In: *33rd ACM SIGSE Technical Symposium on Computer Science Education (SIGSE'02)*, Cincinnati, Kentucky, 2002, p. 137–141.

Alphonce, C.; Ventura, P. Object orientation in cs1-cs2 by design. In: *Proceedings of the 7th annual conference on Innovation and technology in computer science education*, ITiCSE'02, New York, NY, USA: ACM, 2002, p. 70–74 (*ITiCSE'02*,).

Ayewah, N.; Pugh, W.; Morgenthaler, J. D.; Penix, J.; Zhou, Y. Evaluating static analysis defect warnings on production software. In: *PASTE'07: Proceedings of the 7th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, New York, NY, USA: ACM, 2007, p. 1–8.

Barbosa, E. F. *Uma contribuição para a determinação de um conjunto essencial de operadores de mutação no teste de programas C*. Dissertação de Mestrado, ICMC-USP, São Carlos, SP, 1998.

Barbosa, E. F.; Chaim, M. L.; Vincenzi, A. M. R.; Delamaro, M. E.; Jino, M.; Maldonado, J. C. Teste estrutural. In: Delamaro, M. E.; Maldonado, J. C.; Jino, M., eds. *Introdução ao Teste de Software*, Elsevier, p. 47–76, 2007.

Barbosa, E. F.; LeBlanc, R.; Guzdial, M.; Maldonado, J. C. Introducing testing practices into objects and design course. In: *16th Conference on Software Engineering Education and Training (CSEET 2003)*, Madrid, Spain, 2003, p. 179–286.

Barbosa, E. F.; Maldonado, J. C.; Vincenzi, A. M. R. Towards the determination of sufficient mutant operators for C. *STVR*, v. 11, n. 2, p. 113–136, 2001.

Barbosa, E. F.; Silva, M. A. G.; Corte, C. K. D.; Maldonado, J. C. Integrated teaching of programming foundations and software testing. In: *38th Annual Frontiers in Education Conference (FIE 2008)*, Saratoga Springs, NY, 6p., CD-ROM, 2008.

Barbosa, E. F.; Vincenzi, A. M. R.; Delamaro, M. E.; Maldonado, J. C. Teste Estrutural e de Mutação no Contexto de Programas OO. In: *IV Escola Regional de Informática de Minas Gerais (ERI-MG)*, Belo Horizonte, MG, 2005, p. 313–362.

Barriocanal, E. G.; Urbán, M. A. S.; Cuevas, I. A.; Pérez, P. D. An experience in integrating automated unit testing practices in an introductory programming course. In: *In ACM SIGCSE Bulletin*, 2002, p. 125–128.

Beck, K. Simple smalltalk testing: With patterns. In: *The Smalltalk Report*, 1994, p. 16–18.

Beck, K. *Test driven development: By example*. Addison-Wesley, 2003.

Beck, K.; Gamma, E. Junit cookbook. `http://junit.sourceforge.net/doc/cookbook/cookbook.htm`, Último acesso em: 19/03/2012, 2010.

Beizer, B. *Software testing techniques*. 2nd ed. New York: Van Nostrand Reinhold Company, 1990.

Budd, T. A.; DeMillo, R. A.; Lipton, R. J.; Sayward, F. G. Theoretical and empirical studies on using program mutation to test the functional correctness of programs. In: *7th ACM Symposium on Principles of Programming Languages*, New York, NY, 1980, p. 220–233.

Chaim, M. L. *PokeTool – Uma ferramenta para suporte ao teste estrutural de programas baseado em análise de fluxo de dados*. Dissertação de Mestrado, DCA/FEEC/UNICAMP, Campinas, SP, 1991.

Corte, C. K. D. *Ensino integrado de fundamentos de programação e teste de software*. Dissertação de Mestrado, ICMC/USP, São Carlos, SP, 2006.

Corte, C. K. D.; Barbosa, E. F.; Maldonado, J. C. Estabelecimento de mecanismos de apoio ao ensino integrado de fundamentos de programação e teste de software. In: *XXVI Congresso da Sociedade Brasileira de Computação (SBC 2006) – XIV Workshop de Educação em Computação (WEI 2006)*, Campo Grande, MS, CD-ROM., 2006.

Corte, C. K. D.; Riekstin, A. C.; Silva, M. A. G.; Barbosa, E. F.; Maldonado, J. C. PROGTEST: Ambiente para submissão e avaliação de trabalhos práticos. In: *XVIII Simpósio Brasileiro de Informática na Educação (SBIE 2007) – Workshop sobre Ambientes de Apoio à Aprendizagem de Algoritmos e Programação*, São Paulo, SP, 8p., 2007.

Delamaro, M. E.; Maldonado, J. C. Interface mutation: A case study. In: *Workshop do Projeto de Validação e Teste de Sistemas de Operação*, águas de Lindóia – SP, 1997.

Delamaro, M. E.; Maldonado, J. C.; Jino, M. Conceitos básicos. In: Delamaro, M. E.; Maldonado, J. C.; Jino, M., eds. *Introdução ao Teste de Software*, Elsevier, p. 1–8, 2007a.

Delamaro, M. E.; Maldonado, J. C.; Jino, M. *Introdução ao teste de software*. 1 ed. Elsevier, 2007b.

Delamaro, M. E.; Maldonado, J. C.; Jino, M.; Chaim, M. L. Proteum: Uma ferramenta de teste baseada na análise de mutantes. In: *Software Tools Proceedings of the 7th Brazilian Symposium on Software Engineering*, Rio de Janeiro, RJ, 1993, p. 31–33.

Delamaro, M. E.; Vincenzi, A. M. R.; Barbosa, E. F.; Maldonado, J. C. Teste de mutação. In: Delamaro, M. E.; Maldonado, J. C.; Jino, M., eds. *Introdução ao Teste de Software*, Elsevier, p. 77–118, 2007c.

DeMillo, R. A.; Lipton, R. J.; Sayward, F. G. Hints on test data selection: Help for the practicing programmer. *IEEEC*, v. 11, n. 4, p. 34–43, 1978.

Dvornik, T.; Janzen, D. S.; Clements, J.; Dekhtyar, O. Supporting introductory test-driven labs with webide. In: *Proceedings of the 2011 24th IEEE-CS Conference on Software Engineering Education and Training*, CSEET'11, Washington, DC, USA: IEEE Computer Society, 2011, p. 51–60 (*CSEET'11*,).

Edwards, S. H. Using software testing to move students from trial-and-error to reflection-in-action. In: *35th SIGCSE Technical Symposium on Computer Science Education*, Norfolk, Virginia, USA, 2004, p. 26–30.

Edwards, S. H.; Perez-Quinones, M. A. Web-CAT: automatically grading programming assignments. In: *ITiCSE'08: Proceedings of the 13th annual conference on Innovation and technology in computer science education*, New York, NY, USA: ACM, 2008, p. 328–328.

Eler, M. M.; Endo, A. T.; Masiero, P. C.; Delamaro, M. E.; Maldonado, J. C.; Vincenzi, A. M. R.; Chaim, M. L.; Beder, D. M. JaBUTiService: a web service for structural testing of java programs. *Journal of Object Technology (JOT)*, 2009.

Fabbri, S. C. P. F.; Vincenzi, A. M. R.; Maldonado, J. C. Teste funcional. In: Delamaro, M. E.; Maldonado, J. C.; Jino, M., eds. *Introdução ao Teste de Software*, Elsevier, p. 9–26, 2007.

Fowler, M. *Patterns of enterprise application architecture*. Addison-Wesley, 2002.

Frankl, P. G.; Weyuker, E. J. A formal analysis of the fault-detecting ability of testing methods. *IEEE Transactions on Software Engineering*, v. 19, n. 3, p. 202–213, 1993.

Haley, A.; Zweben, S. Development and application of a white box approach to integration testing. *jss*, v. 4, p. 309–315, 1984.

Harrold, M. J.; Soffa, M. L. Selecting and using data for integration test. *IEEES*, v. 8, n. 2, p. 58–65, 1991.

Hickey, T. J. Scheme-based web programming as a basis for a cs0 curriculum. In: *In 35th SIGCSE Technical Symposium on Computer Science Education*, Norfolk, Virginia, USA, 2004, p. 353–357.

Higgins, C.; Hegazy, T.; Symeonidis, P.; Tsintsifas, A. The coursemarker cba system: Improvements over ceilidh. *Education and Information Technologies*, v. 8, p. 287–304, 2003.

Horgan, J. R.; Mathur, P. Assessing testing tools in research and education. *IEEES*, v. 9, n. 3, p. 61–69, 1992.

Hovemeyer, D.; Pugh, W. Finding more null pointer bugs, but not too many. In: *PASTE'07: Proceedings of the 7th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, New York, NY, USA: ACM, 2007, p. 9–14.

Irvine, S. A.; Pavlinic, T.; Trigg, L.; Cleary, J. G.; Inglis, S.; Utting, M. Jumble java byte code to measure the effectiveness of unit tests. *Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION, 2007. TAICPART-MUTATION 2007*, p. 169–175, 2007.

Janzen, D. S.; Saiedian, H. Test-driven learning: intrinsic integration of testing into cs/se curriculum. In: *Proceedings of the 37th SIGSE technical symposium on Computer science education*, New York, NY, USA, 2006, p. 254–258.

Janzen, D. S.; Saiedian, H. Test-driven learning in early programming courses. In: *Proceedings of the 39th SIGSE technical symposium on Computer science education*, New York, NY, USA, 2008, p. 532–536.

Jin, Z.; Offut, A. J. Integration testing based on software couplings. In: *X Annual Conference on Computer Assurance (COMPASS 95)*, Gaithersburg, Maryland, 1995, p. 13–23.

Jones, E. L. An experimental approach to incorporating software testing into the computer science curriculum. In: *31st ASEE/IEEE Frontiers in Education Conference*, Reno, NV, 2001, p. 7–11.

Jorge, R. F.; Vincenzi, A. M. R.; Delamaro, M. E.; Maldonado, J. C. Teste de mutação: Estratégias baseadas em equivalência de mutantes para redução do custo de aplicação. In: *CLEI'2001 – XXVII Latin-American Conference on Informatics*, Mérida, Venezuela, 2001.

Joy, M.; Griffiths, N.; Boyatt, R. The boss online submission and assessment system. *J. Educ. Resour. Comput.*, v. 5, 2005.

Kumar, A.; St.Clair, J. Cunit - a unit testing framework for c. `http://cunit.sourceforge.net/doc/index.html`, Último acesso em: 19/03/2012, 2005.

Kurnia, A.; Lim, A.; Cheang, B. Online judge. *Comput. Educ.*, v. 36, p. 299–315, 2001.

Lahtinen, E.; Ala-Mutka, K.; Järvinen, H. A study of the difficulties of novice programmers. In: *ITiCSE-05: Proceedings of the 10th annual SIGCSE conference on Innovation and technology in computer science education*, New York, NY, USA: ACM Press, 2005, p. 14–18.

Linnenkugel, U.; Müllerburg, M. Test data selection criteria for (software) integration testing. In: *First International Conference on Systems Integration*, Morristown, NJ, 1990, p. 709–717.

Ma, Y.-S.; Offutt, J.; Kwon, Y.-R. Mujava: a mutation system for java. In: *ICSE'06: Proceedings of the 28th international conference on Software engineering*, New York, NY, USA: ACM, 2006, p. 827–830.

Maldonado, J. C. *Critérios potenciais usos: Uma contribuição ao teste estrutural de software*. Tese de Doutoramento, DCA/FEEC/UNICAMP, Campinas, SP, 1991.

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

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

McCabe, T. J. A software complexity measure. *IEEE Transactions on Software Engineering*, v. 2, n. 6, p. 308–320, 1976.

Meszaros, G. *Xunit test patterns: Refactoring test code*. Addison-Wesley, 2007.

Moore, I. Jester – a junit test tester. In: *2nd International Conference on Extreme Programming and Flexible Processes in Software Engineering (XP2001)*, 2001, p. 84–87.

Mresa, E.; Bottaci, L. Efficiency of mutation operators and selective mutation strategies: an empirical study. *The Journal of Software Testing, Verification and Reliability*, v. 9, n. 4, p. 205–232, 1999.

Myers, G. J. *The art of software testing*. 4 ed. Wiley, New York, 2004.

Offutt, A. J.; Lee, A.; Rothermel, G.; Untch, R. H.; Zapf, C. An experimental determination of sufficient mutant operators. *ACMSE*, v. 5, n. 2, p. 99–118, 1996.

Offutt, J.; Ma, Y.-S.; Kwon, Y.-R. The class-level mutants of mujava. In: *AST'06: Proceedings of the 2006 International Workshop on Automation of Software Test*, New York, NY, USA: ACM, 2006, p. 78–84.

Oram, A.; Talbott, S. *Managing projects with make*. O'Reilly Media, 1991.

Patterson, A.; Kölling, M.; Rosenberg, J. Introducing unit testing with bluej. In: *In 8th Annual Conference on Innovation and Technology in Computer Science Education (ITiCSE'03)*, Thessaloniki, Greece, 2003.

Prado, M. P.; Campanha, D. N.; Souza, S. R. S.; Maldonado, J. C. Um conjunto de artefatos para apoio à definição de estudos experimentais em teste de software. In: *Eselaw'2008 - 5th Experimental Software Engineering Latin American Workshop*, Salvador - BA, 2008.

Pressman, R. S. *Software engineering – a practitioner's approach*. 6 ed. McGraw-Hill, 2006.

Rapps, S.; Weyuker, E. J. Data flow analysis techniques for program test data selection. In: *6th International Conference on Software Engineering*, Tokio, Japan, 1982, p. 272–278.

Rapps, S.; Weyuker, E. J. Selecting software test data using data flow information. *IEEE Transactions on Software Engineering*, v. SE-11, n. 4, p. 367–375, 1985.

Spacco, J.; Hovemeyer, D.; Pugh, W.; Emad, F.; Hollingsworth, J. K.; Padua-Perez, N. Experiences with marmoset: designing and using an advanced submission and testing system for programming courses. In: *ITICSE'06: Proceedings of the 11th annual SIGCSE conference on Innovation and technology in computer science education*, New York, NY, USA: ACM, 2006a, p. 13–17.

Spacco, J.; Pugh, W.; Ayewah, N.; Hovemeyer, D. The marmoset project: an automated snapshot, submission, and testing system. In: *OOPSLA'06: Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*, New York, NY, USA: ACM, 2006b, p. 669–670.

Tremblay, G.; Guérin, F.; Pons, A.; Salah, A. Oto, a generic and extensible tool for marking programming assignments. *Softw. Pract. Exper.*, v. 38, p. 307–333, 2008.

Vilela, P. R. S. *Critérios potenciais usos de integração: Definição e análise*. Tese de Doutoramento, DCA/FEEC/UNICAMP, Campinas, SP, 1998.

Vincenzi, A. M. R. *Desenvolvimento baseado em componentes: Conceitos e técnicas*. Editora Ciência Moderna Ltda., 2005.

Vincenzi, A. M. R.; Maldonado, J. C.; Barbosa, E. F.; Delamaro, M. E. Unit and integration testing strategies for C programs using mutation-based criteria. *STVR*, v. 11, n. 4, 2001.

Vincenzi, A. M. R.; Wong, W. E.; Delamaro, M. E.; Maldonado, J. C. JaBUTi: A coverage analysis tool for java programs. In: *XVII Simpósio Brasileiro de Engenharia de Software (SBES 2003)*, Manaus, AM, 2003.

Zhu, H. A formal analysis of the subsume relation between software test adequacy criteria. *IEEESE*, v. SE-22, n. 4, p. 248–255, 1996.

Ziviani, N. *Projeto de algoritmos com implementações em java e C++*. Thomson, 2005.

APÊNDICE

A

Validação Preliminar com Programas Java

Com o objetivo de fornecer uma validação preliminar da PROGTEST foi conduzido um experimento envolvendo a submissão e avaliação de trabalhos práticos de programação por meio da utilização da ferramenta. Ressalta-se que esta validação foi conduzida pelo autor deste trabalho de mestrado, durante seu projeto de iniciacao cientifica (Processo FAPESP 09/00006-3).

Como implementações de referência foram utilizados os trabalhos cadastrados na base de trabalhos oráculo da PROGTEST. Ao todo, 20 trabalhos oráculo foram considerados. Dentre eles, alguns exploram a habilidade do aluno em resolver problemas através do desenvolvimento de algoritmos (por exemplo, calcular a sequência de Fibonacci, ordenar uma sequência de números, calcular o custo da multiplicação de uma matriz). Outros trabalhos consistem em verificar se o aluno entendeu o comportamento de um determinado algoritmo (por exemplo, busca em largura e busca em profundidade em um grafo). Finalmente, há trabalhos que tem como objetivo verificar se o aluno compreendeu os conceitos e operações relativas a diferentes estruturas de dados (tais como pilhas, filas, listas, árvores binárias e grafos).

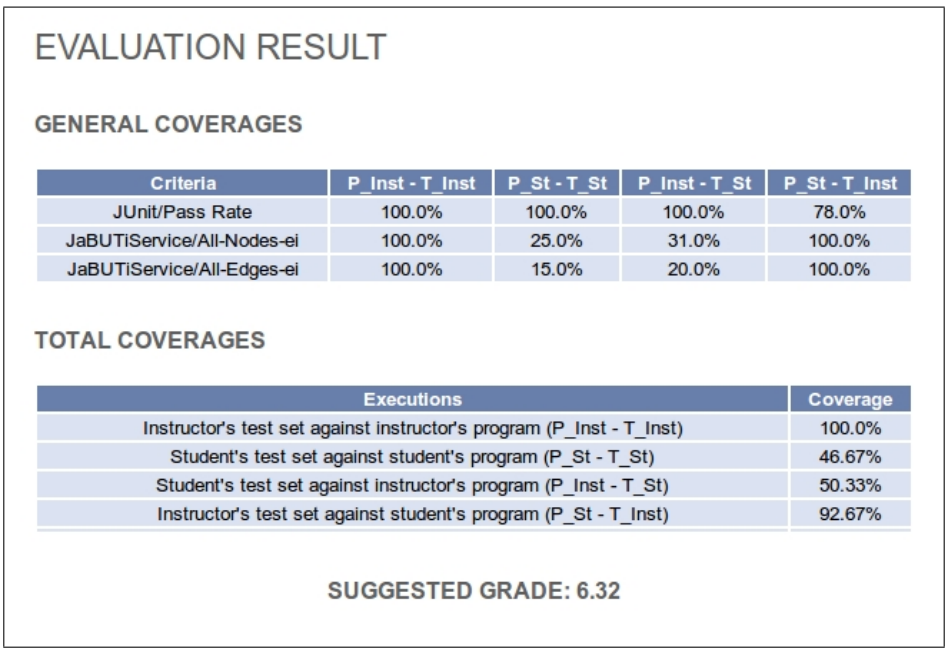
Assim, para cada trabalho oráculo diponível na base de trabalhos oráculo da PROGTEST foram geradas oito implementações e conjuntos de teste associados, representando os trabalhos dos alunos. Em algumas implementações foram inseridos erros típicos que poderiam ser cometidos pelos alunos. Além disso, os conjuntos de teste gerados não eram 100%-adequados aos critérios funcionais e estruturais.

A fim de verificar se a avaliação realizada pela ferramenta era coerente, os trabalhos dos alunos foram submetidos na PROGTEST. Para ilustrar, na Tabela A.1 são apresentadas as oito implementações e conjuntos de teste gerados para o trabalho `Sort`, o qual envolve a ordenação de números inteiros; e as coberturas totais e notas sugeridas para cada implementação. A implementação 4, por exemplo, utiliza como base o algoritmo `Merge Sort`, contém erros pequenos e o conjunto de teste associado apresenta qualidade intermediária.

Tabela A.1: Validação Preliminar: Implementações, Coberturas e Notas (`Sort`)

Implementação	Algorítimo	Erro	Conjunto de Teste	P_{Inst}-T_{Inst}	P_{St_i}-T_{St_i}	P_{Inst}-T_{St_i}	P_{St_i}-T_{Inst}	Suggested Grade
1	Bubblesort	Nenhum	Forte	100	100	100	100	10
2	Quicksort	Pequeno	Forte	100	96.33	100	96.33	9.76
3	Insertion Sort	Nenhum	Medio	100	52.33	50.33	100	6.76
4	Mergesort	Pequeno	Medio	100	46.67	50.33	92.67	6.32
5	Heapsort	Grande	Forte	100	74.33	100	74.33	8.29
6	Selection Sort	Nenhum	Fraco	100	39	39	100	5.93
7	Shellsort	Grande	Fraco	100	38.67	39	56.33	4.47
8	Bubblesort	Pequeno	Com erro de projeto	100	75.67	100	61	7.89

A Figura A.1 mostra o resultado das execuções trocadas para a implementação 4, sendo exibidas: (1) as coberturas obtidas por cada execução para cada critério; (2) a cobertura total de cada execução; e (3) a nota sugerida para a implementação. As coberturas totais são calculadas a partir da média ponderada das coberturas obtidas em cada critério. Da mesma forma, a nota sugerida é calculada pela média ponderada das coberturas $P_{St} - T_{St}$, $P_{Inst} - T_{St}$ e $P_{St} - T_{Inst}$. Os pesos para o cálculo são definidos pelo professor durante a criação do trabalho. No caso do experimento foram utilizados pesos iguais a 1.



EVALUATION RESULT

GENERAL COVERAGES

Criteria	P_Inst - T_Inst	P_St - T_St	P_Inst - T_St	P_St - T_Inst
JUnit/Pass Rate	100.0%	100.0%	100.0%	78.0%
JaBUTiService/All-Nodes-ei	100.0%	25.0%	31.0%	100.0%
JaBUTiService/All-Edges-ei	100.0%	15.0%	20.0%	100.0%

TOTAL COVERAGES

Executions	Coverage
Instructor's test set against instructor's program (P_Inst - T_Inst)	100.0%
Student's test set against student's program (P_St - T_St)	46.67%
Student's test set against instructor's program (P_Inst - T_St)	50.33%
Instructor's test set against student's program (P_St - T_Inst)	92.67%

SUGGESTED GRADE: 6.32

Figura A.1: Validação Preliminar: Resultado da Avaliação da Implementação 4

Considerando ainda a implementação 4, a cobertura do conjunto de teste do aluno em relação ao programa do professor ($P_{Inst} - T_{St_i}$) foi de 92.67% enquanto a cobertura do conjunto de teste do professor em relação a seu próprio programa ($P_{Inst} - T_{Inst}$) foi de 100%. Tal resultado fornece indicativos de que o conjunto de teste do aluno apresenta problemas.

A cobertura do conjunto de teste do aluno em relação a seu próprio programa ($P_{St_i} - T_{St_i}$) também foi baixa (46.67%), resultado esperado tendo em vista que a qualidade do conjunto de teste do aluno é intermediária. Já a cobertura do conjunto de teste do professor em relação ao programa do aluno foi de 50.33%, indicando que a implementação do aluno, tambem apresenta problemas.

Os resultados apresentados nesta seção representam uma validação preliminar da PROGTEST. De modo geral, tais resultados foram consistentes com as características dos trabalhos considerados. Além disso, o comportamento de PROGTEST mostrou-se adequado, ou seja, a ferramenta foi capaz de compilar, executar e avaliar todos os programas e conjuntos de teste conforme o esperado. Ressalta-se, entretanto, que uma validação formal da ferramenta faz-se necessária. Nesse sentido, outros experimentos vêm sendo planejados e deverão ser conduzidos em curto prazo.

APÊNDICE

B

Especificação dos Trabalhos

B.1 Validação 1

A fim de realizar a validação do ambiente PROGTEST com os alunos de pós-graduação, a seguinte especificação de trabalho foi fornecida aos alunos para que pudessem realizar as atividades propostas:

Enunciado

Implementar e testar o programa `Cal` e submeter na PROGTEST. O programa deve ser testado com os critérios `Análise do Valor Limite`, `Todos-Nós` (`ei` e `ed`), `Todas-Arestas` (`ei` e `ed`), `Todos-Usos` (`ei` e `ed`) e `Todos-Potenciais-Usos` (`ei` e `ed`). Os casos de teste devem ser escritos utilizando o *framework* JUNIT. Além disso a ferramenta JABUTI deve ser utilizada como ferramenta de apoio durante a aplicação dos critérios estruturais.

Especificação do Programa

O programa `Cal` deve prover informações sobre o calendário de um determinado mês. Ele deve receber dois argumentos, o primeiro é o mês e o segundo o ano; e retornar o dia da semana em que o mês começa e quantos dias o mês possui. Se apenas um argumento for fornecido, o programa deve retornar os valores referentes ao mês fornecido considerando o ano corrente. Se nenhum argumento for fornecido, o programa deve retornar os valores referentes ao mês

corrente. O valor permitido para o ano é de 1 a 9999. Valores fora do intervalo válido ou argumentos não numéricos devem ser identificados.

No ano de 1752 houve a troca do calendário Juliano para o Gregoriano. Nesse ano houve um ajuste e no mês de setembro não existem os dias 3 a 13. Ou seja, temos o seguinte calendário para aquele mês:

Setembro 1752

Dom	Seg	Ter	Qui	Qua	Sex	Sab
		1	2	14	15	16
17	18	19	20	21	22	23
24	25	26	27	28	29	30

Além disso, a partir daquele ano estabeleceram-se as atuais regras para definir quando um ano é bissexto. Antes disso, todos os anos múltiplos de 4 eram bissextos.

Detalhes de Implementação

O programa `Cal` deverá ser implementado em uma classe com o nome `Cal`. Tal classe deverá implementar os seguintes métodos:

public static int[] cal();
public static int[] cal(int mes);
public static int[] cal(int mes, int ano);

onde: *int[0]* deverá armazenar o primeiro dia (da semana) do mês, conforme a tabela abaixo; e *int[1]* deverá armazenar o número de dias que o mês possui;

Dia da Semana	Valor
Domingo	0
Segunda	1
Terça	2
Quarta	3
Quinta	4
Sexta	5
Sábado	6

Observação: outros métodos podem ser implementados a fim de obter uma melhor modularidade no programa. No entanto, os casos de teste devem utilizar como interface os métodos definidos acima.

B.2 Validação 2

A especificação fornecida aos alunos de graduação foi a mesma utilizada na validação com os alunos de pós-graduação, sendo acrescentados os seguintes itens:

Execução do Programa na PROGTEST

1. Realizar a submissão do programa `Cal` e seus respectivos casos de teste na PROGTEST.
2. Anotar na `Tabela 1` as coberturas obtidas para P_{St}-T_{St}, P_{Inst}-T_{St}, P_{St}-T_{Inst} e a nota sugerida pela PROGTEST, onde:
   - P_{St}-T_{St} = Resultado da execução dos casos de teste do aluno no programa do aluno.
   - P_{Inst}-T_{St} = Resultado da execução dos casos de teste do aluno no programa do professor.
   - P_{St}-P_{Inst} = Resultado da execução dos casos de teste do professor no programa do aluno
3. Utilizar os relatórios fornecidos pela PROGTEST para identificar eventuais problemas no programa e/ou nos casos de teste submetidos.
4. Se algum problema for idenficado, descrevê-lo brevemente na `Tabela 1`, corrigir o programa e/ou casos de teste e repetir os passos de 1 a 4.
5. Identificar os elementos não executáveis e apresenta-los na `Tabela 2`.

Relatório

Apresentar um relatório com os resultados das execuções, contendo:

1. Descrição da cobertura, considerando os casos de teste iniciais.
2. Evolução passo a passo do aumento da cobertura.
3. `Tabela 1`
4. `Tabela 2`
5. Dificuldades encontradas.
6. Observações adicionais e sugestões de melhoria para o ambiente PROGTEST.

Tabela 1

#	$P_{St} - T_{St}$	$P_{Inst} - T_{St}$	$P_{St} - T_{Inst}$	**Nota**	**Problemas Identificados**
1					
2					
...					
n					

Tabela 2

#	**Critério**	**Método**	**Requisitos**
1			
2			
...			
n			