Aspectos de Desenvolvimento e Evolução de um Ambiente de Apoio ao Ensino de Programação e Teste de Software

Draylson Micael de Souza[1], **José Carlos Maldonado**[1], **Ellen Francine Barbosa**[1]

[1]Instituto de Ciências Matemáticas e de Computação (ICMC/USP)
Caixa Postal 668 – CEP 13566-970 – São Carlos – SP – Brasil

{draylson,jcmaldon,francine}@icmc.usp.br

Abstract. *Environments for submission and automated assessment of assignments have been developed as support for the teaching of programming and software testing. Among them, we highlight* PROGTEST. *However, the first version of* PROGTEST *support only Java language and structural testing criteria. In this paper we describe the evolution of* PROGTEST*, performed to make* PROGTEST *capable of supporting different programming languages and testing criteria. For this, we performed the analysis and integration to* PROGTEST *of different testing tools. The results obtained from the application of the new version of* PROGTEST *shows that it is capable of assess the programming and testing assignments more adequately.*

Resumo. *Ambientes de apoio a submissão e avaliação automática de trabalhos práticos vêm sendo desenvolvidos como ferramentas de apoio ao ensino de programação e teste de software. Dentre eles, destaca-se a* PROGTEST. *Em sua primeira versão, a* PROGTEST *apenas apoiava a linguagem Java e critérios de teste estruturais. Este artigo descreve a evolução da* PROGTEST *a fim de torná-la capaz de apoiar diferentes linguagens de programação e critérios de teste. Para isso, foram analisadas e integradas à* PROGTEST *diferentes ferramentas de teste. Os resultados obtidos a partir de uma validação mostram que a* PROGTEST*, com as novas ferramentas, pode avaliar os trabalhos de programação e teste de uma forma mais adequada.*

1. Introdução

O ensino de fundamentos de programação não é uma tarefa trivial – muitos alunos têm dificuldades em compreender os conceitos de programação [Lahtinen et al. 2005] e/ou possuem visões erradas sobre a atividade de programação [Edwards 2004]. Entre as iniciativas que têm sido investigadas a fim de minimizar tais problemas destaca-se o ensino conjunto de conceitos básicos de programação e de teste de software. A ideia é que a introdução da atividade de teste pode ajudar o desenvolvimento das habilidades de compreensão e análise nos alunos, 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 também pode ser ensinada o mais cedo possível [Dvornik et al. 2011, Edwards 2004, Janzen and Saiedian 2006, Souza et al. 2011a]. Basicamente, alunos que aprendem teste mais cedo podem se tornar melhores desenvolvedores uma vez que o teste força a integração e aplicação de teorias e habilidades de análise, projeto e implementação. Por

outro lado, o ensino de teste também não é uma atividade trivial. As dificuldades vão desde a carência de ambientes de apoio à falta de motivação dos alunos em realizar uma atividade de teste adequada [Edwards 2004].

Dentro desse contexto, ambientes automatizados que apoiam o ensino integrado de programação e teste vêm sendo desenvolvidos. Dentre eles, encontra-se a PROGTEST [Souza et al. 2011a, Souza et al. 2011b] – um ambiente de apoio à submissão e avaliação automática de trabalhos práticos de programação, baseado em atividades de teste de software. A partir da utilização de ferramentas de teste, a PROGTEST avalia automaticamente o trabalho dos alunos, fornecendo a eles *feedback* imediato sobre seus trabalhos. Em sua primeira versão [Souza et al. 2011a, Souza et al. 2011b] estavam integrados ao ambiente PROGTEST: (1) o *framework* JUNIT [Beck and Gamma 2010], apoiando a execução automática de casos de teste; e (2) a ferramenta JABUTISERVICE [Eler et al. 2009], apoiando a aplicação de critérios de teste estrutural. Em ambos os casos, a linguagem de programação considerada foi Java.

Apesar das vantagens observadas em sua primeira versão, dois aspectos importantes ainda limitavam o uso da PROGTEST. O primeiro refere-se ao fato de que a PROGTEST apenas apoiava a submissão e avaliação de programas em Java. Entretanto, em disciplinas introdutórias de programação, é comum os professores ensinarem os conceitos de programação utilizando linguagens de programação procedimentais, como por exemplo, a linguagem C. O segundo aspecto refere-se ao fato de que a PROGTEST apoiava apenas a aplicação de critérios de teste estruturais, impossibilitando que critérios de teste mais "fortes" e efetivos fossem considerados.

Nessa perspectiva, a PROGTEST foi evoluída a fim de possibilitar o apoio a linguagens de programação procedimental (em especial, C) e critérios de teste baseado em erros (em especial, Análise de Mutantes [DeMillo et al. 1978]). A evolução foi realizada a partir da análise e integração à PROGTEST de diferentes ferramentas de teste. Este artigo sintetiza as principais melhorias conduzidas. Além disso, uma validação da PROGTEST considerando a aplicação das novas ferramentas integradas também é apresentada.

O restante deste artigo esta organizado da seguinte forma. Na Seção 2 são descritas as principais características e funcionalidades do ambiente PROGTEST, com ênfase nos aspectos considerados em sua evolução. Na Seção 3 é descrita a validação do ambiente PROGTEST, bem como os resultados observados. Por fim, na Seção 4 são apresentadas as conclusões e trabalhos futuros a serem realizados.

2. O Ambiente PROGTEST

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, o trabalho dos professores é duplicado 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.

A PROGTEST, ilustrada na Figura 1, insere-se nesse contexto como um ambiente web para submissão e avaliação automática de trabalhos de programação, baseado em atividades de teste [Souza et al. 2011a, Souza et al. 2011b]. A ideia é fornecer apoio automatizado para avaliar os programas e casos de teste submetidos pelos alunos. Para

isso, ferramentas de teste estão integradas à PROGTEST, fornecendo apoio à aplicação de critérios de teste. Tanto a qualidade do código como a qualidade dos testes podem ser analisadas com base nos critérios adotados.

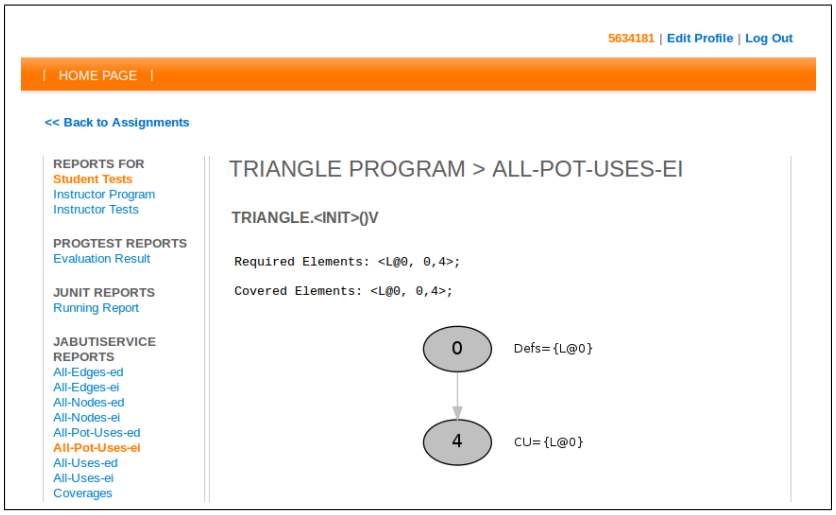

Figura 1. O Ambiente PROGTEST

As principais funcionalidades da PROGTEST podem ser acessadas por meio de duas visões – professor ou aluno. Em relação à visão do professor, a PROGTEST permite ao usuário criar novos cursos, matricular alunos e definir trabalhos de programação. Para definir um trabalho, o professor deve especificar 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.

O professor também deve fornecer um "trabalho oráculo", o qual consiste em: (1) um programa que implementa a solução correta para o trabalho proposto; e (2) um conjunto de teste para o programa fornecido, sendo que esse deve ser 100%-adequado aos critérios de teste considerados na avaliação.

Considerando a visão do aluno, este tem acesso a todos os trabalhos associados aos cursos em que ele está matriculado. Ao submeter uma solução para um dado trabalho, o aluno deve enviar seu programa e o conjunto de teste que utilizou para testá-lo.

Com base nos critérios e pesos definidos pelo professor, a PROGTEST calcula uma cobertura total do trabalho oráculo ($P_{Inst} - T_{Inst}$). Em seguida, a PROGTEST calcula a cobertura para as seguintes combinações:

1. programa do aluno com o conjunto de teste do aluno ($P_{St} - T_{St}$);
2. programa do professor com o conjunto de teste do aluno ($P_{Inst} - T_{St}$)
3. programa do aluno com o conjunto de teste do professor ($P_{St} - T_{Inst}$).

A partir das coberturas obtidas, uma nota para o trabalho do aluno é calculada:

$$Nota = \frac{p_1 * P_{St}T_{St} + p_2 * P_{Inst}T_{St} + p_3 * P_{St}T_{Inst}}{p_1 + p_2 + p_3}$$

onde os valores de p_1, p_2, p_3 também são definidos pelo professor.

Após ter submetido seu trabalho, o aluno pode visualizar o relatório de avaliação, além de outros relatórios fornecidos pelas ferramentas de teste integradas. Além disso, o aluno pode submeter novas versões do seu trabalho até atingir nota máxima.

2.1. PROGTEST: Comparação com Ambientes Similares

Entre os ambientes que fornecem apoio à submissão e avaliação automática de trabalhos de programação destacam-se a WEB-CAT [Edwards 2004] e a MARMOSET [Spacco et al. 2006]. 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 2004]. A WEB-CAT avalia os trabalhos dos alunos de acordo com três parâmetros: (1) legibilidade/projeto, que deve ser analisado manualmente por um assistente do professor; (2) estilo/codificação, analisado automaticamente por meio de ferramentas de análise estática, e (3) teste/correção, analisado automaticamente por meio de ferramentas de teste.

As ferramentas de teste e de análise estática são disponibilizadas na WEB-CAT por meio de *plugins*. Cada *plugin* possui critérios de avaliação e relatórios próprios, associados à uma linguagem de programação particular. Dentre as linguagens apoiadas, encontram-se *plugins* para Java, C, C++ e Pascal.

A MARMOSET também é um ambiente de submissão de trabalhos de programação. Os casos de teste utilizados para testar os projetos na MARMOSET são de quatro tipos: (1) testes dos alunos, fornecidos pelos próprios alunos; (2) testes públicos, fornecidos aos alunos antes de iniciarem seus trabalhos; (3) testes de liberação, escritos pelo professor e disponibilizados aos alunos em condições específicas ou após o prazo de entrega do trabalho; e (4) testes secretos, escritos pelo professor e disponibilizados aos alunos somente após o prazo de entrega dos trabalhos. Ainda, para projetos em Java, a MARMOSET realiza: (1) a identificação de defeitos por meio de ferramentas de análise estática; e (2) a análise de cobertura dos testes por meio de ferramentas de teste.

A principal diferença da PROGTEST em relação aos ambientes discutidos está na utilização de um programa de referência (trabalho oráculo) fornecido pelo professor para avaliar os trabalhos dos alunos. Considerando, como exemplo, o problema de calcular o fatorial de um número, uma possível solução é implementada no `Trabalho 1` (Figura 2). O `Programa 1` (Figura 2 (a)), implementa uma solução correta para o problema e o `Conjunto de Teste 1` (Figura 2 (b)) exercita todas as linhas e desvios de fluxo do `Programa 1`, ou seja, é 100%-adequado aos critérios Todos-Nós e Todas-Arestas.

```
1  public static int fatorial(int n) {
2        if(n <= 1)
3              return 1;
4        else
5              return n*fatorial(n-1);
6  }
7
8
```

(a) `Programa 1`

#	Entrada	Saída Esperada
1	0	1
2	1	1
3	2	2
4	3	6

(b) `Conjunto de Teste 1`

Figura 2. Trabalho 1 (Oráculo)

O `Trabalho 2`, apresentado na Figura 3, exemplifica uma solução incorreta. O `Programa 2` (Figura 3 (a)) calcula incorretamente o valor do fatorial de 4. Além disso,

embora o `Conjunto de Teste 2` (Figura 3 (b)) exercite todas as linhas e desvios de fluxo do `Programa 2`, o caso de teste que exercita o programa com entrada igual a 4 possui um defeito, não falhando quando executado contra o `Programa 2`.

```
1    public static int fatorial(int n) {
2          if(n <= 1)
3                return 1;
4          if(n == 4)
5                return 16;
6          else
7                return n*fatorial(n-1);
8    }
```

(a) `Programa 2`

#	Entrada	Saída Esperada
1	0	1
2	1	1
3	2	2
4	4	16

(b) `Conjunto de Teste 2`

Figura 3. Trabalho 2 (Solução Incorreta)

Considerando o `Trabalho 1` como uma implementação de referência fornecida pelo professor e o `Trabalho 2` como o trabalho desenvolvido por um aluno, a Tabela 1 ilustra o resultado da submissão do `Trabalho 2` nos ambientes de avaliação automática. A WEB-CAT e a MARMOSET avaliam a correção do programa e a adequação do conjunto de teste considerando apenas os valores marcados com asterisco (*), não identificando os problemas do `Trabalho 2`.

Por outro lado, a PROGTEST avalia o trabalho considerando todos os valores apresentados. Ao executar o conjunto de teste do aluno contra o programa de referência ($P_{Inst} - T_{St}$), a PROGTEST identifica o problema no conjunto de teste do aluno, uma vez que apenas 75% dos casos de teste do aluno não falharam. Adicionalmente, a PROGTEST identifica que o conjunto de teste do professor exercita apenas 85.71% das linhas de código e 80% dos desvios de fluxo do programa do aluno ($P_{St} - T_{Inst}$), mostrando que elementos do programa do aluno podem conter defeitos.

Tabela 1. Resultados da Avaliação do `Trabalho 2`

	$P_{Inst} - T_{Inst}$	$P_{St} - T_{St}$	$P_{Inst} - T_{St}$	$P_{St} - T_{Inst}$
Casos de Teste Não Falhos	100%	100% (*)	75%	100% (*)
Comandos Cobertos (Todos-Nós)	100%	100% (*)	100%	85.71%
Desvios de Fluxo Cobertos (Todas-Arestas)	100%	100% (*)	100%	80%

Outras vantagens associadas ao uso da PROGTEST também podem ser observadas: (1) o apoio à aplicação de critérios estruturais mais rigorosos, como os baseados em fluxo de dados (Todos-Usos e Todos-Potenciais-Usos); (2) a possibilidade de controlar o rigor da avaliação por meio de pesos atribuídos aos critério de teste; e (3) a disponibilização de uma base de trabalhos oráculos, permitindo que o professor apenas selecionar um trabalho da base, ao invés de implementar e testar sua própria implementação de referência.

2.2. Evolução do Ambiente PROGTEST

Como discutido anteriormente, em disciplinas introdutórias de programação é comum os professores ensinarem os conceitos de programação utilizando uma linguagem de programação procedimental. As ferramentas de teste integradas à primeira versão da PROGTEST apoiavam o teste de programas escritos apenas em linguagem Java, dificultando com isso o uso efetivo do ambiente em disciplinas introdutórias. Além disso, considerando a utilização da PROGTEST em disciplinas de teste de software, o ambiente não apoiava adequadamente a aplicação de critérios baseados em erros, como a Análise

de Mutantes [DeMillo et al. 1978], não fornecendo um apoio completo e mais sistemático ao ensino de teste de software.

Dentro desse cenário, a evolução do ambiente PROGTEST foi realizada de modo a torná-lo capaz de apoiar diferentes linguagens de programação e critérios de teste, por meio da integração de novas ferramentas de teste. Inicialmente, uma análise do ambiente foi realizada 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 estava bastante acoplado ao código da PROGTEST e era específico para cada uma dessas ferramentas, contrariando a ideia de um ambiente em que diversas ferramentas de teste pudessem ser integradas.

A partir da análise conduzida, antes de realizar a integração de outras ferramentas foi necessário estabelecer um meio de comunicação padrão entre o ambiente PROGTEST e as ferramentas. A ideia foi permitir que a execução das ferramentas ocorresse de maneira isolada da PROGTEST, mantendo apenas uma interface para troca de informações.

Com isso, foi estabelecida a integração das ferramentas de teste de forma a permitir sua operação como *plugins* dentro do ambiente PROGTEST. 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 à interface de comunicação estabelecida.

A Figura 4 mostra o código da PROGTEST responsável em executar um *plugin*. Em linhas gerais, as seguintes atividades são realizadas: (1) a PROGTEST lê, de um arquivo de configuração, o comando que inicia a execução do *plugin*; (2) os argumentos do comando são separados e armazenados em um vetor; (3) caso a ferramenta apoie o critério Análise de Mutantes, parâmetros que especificam quais operadores devem ser aplicados são acrescentados ao comando; (4) se um argumento consiste na especificação de um diretório ou arquivo, a PROGTEST substitui a especificação pelo caminho completo do arquivo ou diretório associado; (5) um processo é criado, associando a ele os argumentos; e (6) o processo é associado à uma nova *thread* que, em seguida, é iniciada, resultando na execução do *plugin*.

Após a execução, o *plugin* deve gerar um arquivo de saída contendo a cobertura obtida para cada critério de teste que a ferramenta associada apoia. Além do arquivo de saída, relatórios XML fornecidos pelos *plugins* também são carregados pela PROGTEST para serem exibidos aos usuários.

A Tabela 2 apresenta as ferramentas integradas ao ambiente PROGTEST. A escolha priorizou a introdução da linguagem de programação C e a introdução do critério Análise de Mutantes [DeMillo et al. 1978].

Tabela 2. Ferramentas de Teste Integradas ao Ambiente PROGTEST

	Java	**C**
Frameworks de Teste de Unidade	JUNIT [Beck and Gamma 2010]	CUNIT [CUnit Project 2012]
Ferramentas de Teste Estrutural	JABUTISERVICE [Eler et al. 2009]	GCOV [GCC Team 2012]
Ferramentas de Teste Baseado em Erros	JUMBLE [Irvine et al. 2007]	PROTEUM [Delamaro and Maldonado 2001]

```
1  String[] args = (cmdfile.readLine() + parameters).split(" ");
2
3  for (int i = 0; i < args.length; i++)
4          args[i] = args[i]
5                  .replace(TAG_ROOT, toolDir.getPath())
6                  .replace(TAG_SOURCE, srcDir.getPath())
7                  .replace(TAG_BINARIES, binDir.getPath())
8                  .replace(TAG_PROGRAM, progDir.getPath())
9                  .replace(TAG_TESTS, testDir.getPath())
10                 .replace(TAG_INTRUMENTED, instDir.getPath())
11                 .replace(TAG_LIBRARIES, libDir.getPath())
12                 .replace(TAG_TEMPORARIES, tmpDir.getPath())
13                 .replace(TAG_REPORTS, rptDir.getPath())
14                 .replace(FILE_SEPARATOR, File.separator);
15                 .replace(PATH_SEPARATOR, File.pathSeparator);
16
17 processBuilder = new ProcessBuilder(args);
18 process = processBuilder.start();
19
20 worker = new Worker(process);
21 worker.start();
```

Figura 4. Código Responsável em Executar um *Plugin*

3. Validação do Ambiente PROGTEST

Esta seção discute a validação da PROGTEST, considerando a linguagem C e o critério Análise de Mutantes [DeMillo et al. 1978]. A primeira versão da PROGTEST foi validada considerando a linguagem de programação Java e critérios de teste estruturais. Uma síntese dos resultados obtidos encontra-se em Souza et al. [2011a, 2011b].

Para as validações descritas neste artigo, uma definição de trabalho foi criada na PROGTEST considerando um dos trabalhos da base de trabalhos oráculo; no caso, o programa `Sort`. Com base no trabalho oráculo, diferentes implementações foram construídas, considerando 7 algoritmos diferentes. A Tabela 3 mostra as principais características das implementações produzidas. Algumas das implementações possuem defeitos em seus programas; em outras, defeitos foram inseridos nos conjuntos de teste; e em algumas implementações, a qualidade dos conjuntos de teste foi reduzida.

Tabela 3. Programas e Conjuntos de Teste

Implementações	Programa	Conjunto de Teste
Trabalho Oráculo	Sem Defeito	Sem Defeito
1 a 7	Sem Defeito	Sem Defeito
8 a 14	Sem Defeito	Com Defeito
14 a 21	Sem Defeito	Menor Qualidade
22 a 28	Com Defeito	Sem Defeito
29 a 35	Com Defeito	Com Defeito
36 a 42	Com Defeito	Menor Qualidade

Em seguida, as implementações foram submetidas à PROGTEST para avaliação. A ideia foi verificar o impacto que cada alteração e defeito inserido implicaria nas coberturas e notas finais. Com as implementações produzidas, duas validações foram realizadas: (1) a primeira considerando somente o uso de critérios estruturais no teste de programas em C; e (2) a segunda considerando tanto critérios estruturais como baseados em erros.

3.1. Validação 1: Critérios Estruturais

A Tabela 4 mostra as coberturas e notas obtidas na Validação 1 para as implementações com o algoritmo `HeapSort`. Na `Implementação 2`, por exemplo, nenhum defeito

foi inserido no programa e no conjunto de teste. Como esperado, a cobertura obtida em cada execução foi de 100% e a nota 10.0 foi sugerida pela PROGTEST.

Tabela 4. Validação 1: Coberturas e Notas das Implementações com `HeapSort`

Impl.	Programa	Conj. de Teste	$P_{Inst} - T_{Inst}$	$P_{St} - T_{St}$	$P_{Inst} - T_{St}$	$P_{St} - T_{Inst}$	Nota
2	Sem Defeito	Sem Defeito	100.00%	100%	100%	100%	10.0
9	Sem Defeito	Com Defeito	100.00%	95%	95%	100%	9.67
16	Sem Defeito	Menor Qualidade	100.00%	100%	100%	100%	10.0
23	Com Defeito	Sem Defeito	100.00%	60%	100%	60%	7.33
30	Com Defeito	Com Defeito	100.00%	60%	95%	60%	7.17
37	Com Defeito	Menor Qualidade	100.00%	50%	100%	60%	7.0

Por outro lado, na `Implementação 9`, um defeito foi inserido no conjunto de teste. Nas execuções em que o conjunto de teste com defeito é considerado ($P_{St} - T_{St}$ e $P_{Inst} - T_{St}$), uma cobertura de 95% foi obtida, refletindo o defeito inserido no conjunto de teste. A nota atribuída pela PROGTEST à `Implementação 9` foi de 9.67.

Já a menor qualidade do conjunto de teste da `Implementação 16` não foi detectada quando executado contra o programa oráculo ($P_{Inst} - T_{St}$). Como será discutido na Seção 3.2, a introdução de critérios de teste mais fortes, como Análise de Mutantes [DeMillo et al. 1978], é fundamental para avaliar de forma mais rigorosa o conjunto de teste dos alunos.

Por fim, as implementações de 23, 30 e 37 possuem defeitos no programa. Observando como exemplo as coberturas obtidas pela `Implementação 23` (Tabela 5), nas execuções em que o programa com defeito é considerado ($P_{St} - T_{St}$ e $P_{St} - T_{Inst}$) apenas 20% dos casos de teste não falham, ressaltando assim o defeito do programa.

Figura 5. Validação 1: Coberturas Obtidas pela `Implementação 23`

	$P_{Inst} - T_{Inst}$	$P_{St} - T_{St}$	$P_{Inst} - T_{St}$	$P_{St} - T_{Inst}$
Casos de Teste Não Falhos	100%	20%	100%	20%
Cobertura de Código (Todos-Nós e Todas-Arestas)	100%	100%	100%	100%
Cobertura Total	**100%**	**60%**	**100%**	**60%**

3.2. Validação 2: Critérios Estruturais e Baseados em Erros

Na Validação 2 foram considerados tanto critérios e ferramentas de teste estrutural como critérios e ferramentas de teste baseado em erros. A ideia foi verificar o impacto do critério Análise de Mutantes [DeMillo et al. 1978] na avaliação realizada pela PROGTEST.

A título de ilustração, a Tabela 5 mostra as características e notas obtidas para as implementações com o algoritmo `HeapSort`. As notas variam adequadamente de acordo com o problema inserido nas implementações, analogamente aos resultados obtidos na Validação 1. No entanto, é possível observar que mesmo a `Implementação 2`, em que nenhum defeito foi inserido, não atingiu notas iguais a 10.0.

Tabela 5. Validação 2: Coberturas e Notas das Implementações com `HeapSort`

Impl.	Programa	Conj. de Teste	$P_{Inst} - T_{Inst}$	$P_{St_i} - T_{St_i}$	$P_{Inst} - T_{St_i}$	$P_{St_i} - T_{Inst}$	Nota
2	Sem Defeito	Sem Defeito	94.09%	93.09%	94.09%	93.09%	9.54
9	Sem Defeito	Com Defeito	94.09%	89.75%	90.76%	93.09%	9.31
16	Sem Defeito	Menor Qualidade	94.09%	90.2%	85.56%	93.09%	9.14
23	Com Defeito	Sem Defeito	94.09%	63.12%	94.09%	63.12%	7.54
30	Com Defeito	Com Defeito	94.09%	63.53%	90.76%	63.12%	7.44
37	Com Defeito	Menor Qualidade	94.09%	46.65%	85.56%	63.12%	6.69

Isso se deve ao fato de que, para aplicar o critério Análise de Mutantes, versões do programa com defeitos (mutantes) precisam ser geradas. Em seguida, é verificado se o conjunto de teste em questão consegue identificar os defeitos presentes nos mutantes. No entanto, alguns programas mutantes são equivalentes ao programa original, ou seja, embora sejam diferentes, sempre geram saídas iguais as do programa original. Como consequência disso, nenhum caso de teste é capaz de identificar o defeito no programa mutante.

A Tabela 6 mostra as coberturas obtidas pela `Implementação 2`. Observa-se que os conjuntos de teste conseguem identificar os defeitos de 82.27% dos programas mutantes gerados para o programa oráculo (execuções $P_{Inst} - T_{Inst}$ e $P_{Inst} - T_{St}$) e de 79.27% dos programas mutantes gerados para o `HeapSort` da `Implementação 2` (execuções $P_{St} - T_{St}$ e $P_{St} - T_{Inst}$). Os conjuntos de teste não conseguem atingir 100% porque os programas mutantes, cujos defeitos não foram identificados, são mutantes equivalentes.

Figura 6. Validação 2: Coberturas Obtidas pela `Implementação 2`

	$P_{Inst} - T_{Inst}$	$P_{St} - T_{St}$	$P_{Inst} - T_{St}$	$P_{St} - T_{Inst}$
Casos de Teste Não Falhos	100%	100%	100%	100%
Cobertura de Código (Todos-Nós e Todas-Arestas)	100%	100%	100%	100%
Programas Mutantes com Defeitos Identificados	82.27%	79.27%	82.27%	79.27%
Cobertura Total	**94.09%**	**93.09%**	**94.09%**	**93.09%**

Apesar dessa limitação, uma das vantagens observadas em utilizar o critério Análise de Mutantes foi a avaliação mais rigorosa do conjunto de teste dos alunos. Considerando como exemplo a `Implementação 16`, na Validação 1, 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 o critério Análise de Mutantes, a menor qualidade do conjunto de teste foi identificada. A cobertura $P_{Inst} - T_{St}$, que avalia a qualidade do conjunto de teste, foi reduzida para 85.56%, inferior à cobertura de 94.09%, calculada para a `Implementação 1`, que contém um conjunto de teste com maior qualidade.

4. Conclusão e Trabalhos Futuros

Neste artigo foi discutida a evolução do ambiente PROGTEST a fim de possibilitar: (1) o apoio a programas escritos em linguagem C; e (2) a condução de testes por meio da Análise de Mutantes [DeMillo et al. 1978]. Para isso, novas ferramentas foram integradas a PROGTEST. Além disso, validações foram realizadas visando verificar se a PROGTEST, com as novas ferramentas integradas, era capaz de avaliar os trabalhos dos alunos adequadamente. Em síntese, a PROGTEST foi capaz de identificar os problemas nos trabalhos dos alunos. Além disso, foi observado que a utilização de critérios de teste mais "fortes", como o critério Análise de Mutantes [DeMillo et al. 1978] foi fundamental para avaliar os conjuntos de teste dos alunos de forma mais efetiva.

Como trabalhos futuros pretende-se investigar, em especial, mecanismos que mitiguem os efeitos dos mutantes equivalentes. Além disso, experimentos vêm sendo planejados e executados em cenários reais de ensino e aprendizagem, envolvendo o uso do ambiente PROGTEST por alunos em disciplinas introdutórias de programação.

Referências

Beck, K. and Gamma, E. (2010). JUnit Cookbook. `http://junit.sourceforge.net/doc/cookbook/cookbook.htm`. Último acesso em: 19/03/2012.

CUnit Project (2012). CUnit - A Unit testing Framework for C. `http://cunit.sourceforge.net/`. Último acesso em: 23/07/2012.

Delamaro, M. E. and Maldonado, J. C. (2001). Proteum/IM 2.0: An integrated mutation testing environment. In Wong, W. E., editor, *Mutation testing for the new century*, pages 91–101. Kluwer Academic Publishers, Norwell, MA, USA.

DeMillo, R. A., Lipton, R. J., and Sayward, F. G. (1978). Hints on test data selection: help for the practicing programmer. *Computer*, 11(4):34–41.

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

Edwards, S. H. (2004). Using software testing to move students from trial-and-error to reflection-in-action. *SIGCSE Bulletin*, 36(1):26–30.

Eler, M. M., Endo, A. T., Masiero, P. C., Delamaro, M. E., Maldonado, J. C., Vincenzi, A. M. R., Chaim, M. L., and Beder, D. M. (2009). JaBUTiService: a web service for structural testing of Java programs. In *Proceedings of the 2009 33rd Annual IEEE Software Engineering Workshop*, SEW '09, pages 69–76, Skövde, Sweden.

GCC Team (2012). Gcov - Using the GNU Compiler Collection (GCC). `http://gcc.gnu.org/onlinedocs/gcc/Gcov.html`. Último acesso em: 03/08/2012.

Irvine, S. A., Pavlinic, T., Trigg, L., Cleary, J. G., Inglis, S., and Utting, M. (2007). Jumble Java Byte Code to measure the effectiveness of unit tests. In *Proceedings of the Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION*, TAICPART-MUTATION '07, pages 169–175, Cumberland Lodge, Windsor, UK.

Janzen, D. S. and Saiedian, H. (2006). Test-driven learning: intrinsic integration of testing into the CS/SE curriculum. *SIGCSE Bulletin*, 38(1):254–258.

Lahtinen, E., Ala-Mutka, K., and Järvinen, H.-M. (2005). A study of the difficulties of novice programmers. *SIGCSE Bulletin*, 37(3):14–18.

Souza, D. M., Maldonado, J. C., and Barbosa, E. F. (2011a). ProgTest: An environment for the submission and evaluation of programming assignments based on testing activities. In *Proceedings of the 2011 24th IEEE-CS Conference on Software Engineering Education and Training*, CSEET '11, pages 1–10, Honolulu, HI, USA.

Souza, D. M., Maldonado, J. C., and Barbosa, E. F. (2011b). ProgTest: Apoio automatizado ao ensino integrado de programação e teste de software. In *Anais do XXII Simpósio Brasileiro de Informática na Educação - XVII Workshop de Informática na Educação*, SBIE-WIE '11, pages 1893–1897, São Paulo, SP, Brazil.

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