Ensino Integrado de Fundamentos de Programação e de Teste de Software

Camila Kozlowski Della Corte[1]*, **Ellen Francine Barbosa**[1] , **José Carlos Maldonado**[1]

[1]Instituto de Ciências Matemáticas e de Computação
Universidade de São Paulo
Av. do Trabalhador São-Carlense, 400, Centro
Caixa Postal 668 – 13560-970 São Carlos, SP

camila@icmc.usp.br, francine@icmc.usp.br, jcmaldon@icmc.usp.br

Resumo. *A importância do ensino de teste para o desenvolvimento de software é amplamente reconhecida, mas somente uma parte muito pequena dos currículos dos cursos de computação é alocada a esse tipo de ensino. O ensino de teste deve começar o mais cedo possível para que uma cultura adequada de teste de software seja criada. Uma maneira de alcançar isso é ensinar teste de software juntamente com fundamentos de programação em disciplinas introdutórias dos cursos de computação. Neste artigo propõe-se a integração do ensino de teste de software com o ensino de fundamentos de programação. Para isso, pretende-se elaborar material didático integrado de fundamentos de programação e de teste de software e ainda desenvolver um ambiente, baseado na Web para submissão e avaliação automática de trabalhos práticos baseados em atividades de teste de software para auxiliar os processos de ensino e aprendizagem.*

Abstract. *The importance of teaching software testing is widely recognized, but only a small portion of the computer science (CS) curriculum is allocated to it. The teaching of software testing must begin as earlier as possible so an adequate culture of software testing be created. One way to achieve this is to teach software testing in conjunction with programming foundations in introductory CS disciplines. In this paper is proposed to integrate the teaching of software testing along with the teaching of programming foundations. In this direction, we intend to elaborate integrated learning materials on programming foundations and software testing and to develop a Web-based environment for the submission and automatic evaluation of practical testing activities to aid the teaching and learning processes.*

1. Introdução

Segundo Edwards [14], a atividade de programação está sendo ensinada baseada na abordagem da tentativa e erro [14]. Na maioria das vezes, a ênfase do ensino de programação encontra-se na sintaxe de alguma linguagem de programação. Isso acaba levando os alunos a não pensarem realmente na forma como os problemas devem ser resolvidos com o desenvolvimento de algoritmos e acabam concentrando seu aprendizado nos detalhes técnicos das linguagens de programação que são ensinadas [3].

*Apoio Financeiro da Fapesp, processo 03/04567-3.

A prática de tentativa e erro geralmente ocorre porque a programação, em grande parte dos currículos de graduação, tem seu ensino voltado para o desenvolvimento de aplicações e codificação. Habilidades de compreensão e análise não são ensinadas aos alunos. Essas habilidades devem ser pré-requisitos para o ensino de programação. Os estudantes devem desenvolver essas habilidades para conseguir solucionar melhor os problemas, ler e compreender o código fonte, prever como a seqüência de comandos do programa irá se comportar, e, ainda, prever como mudanças no código poderão resultar na mudança de comportamento do programa [8].

Esse cenário acaba levando os alunos a terem visões erradas da atividade de programação [14]. Uma forma de tentar ajudar a diminuir essas visões erradas que os estudantes possuem sobre programação é ensinar a atividade de teste conjuntamente com fundamentos de programação.

Outras mudanças estão acontecendo no sentido de tentar melhorar o ensino de fundamentos de programação, dentre elas, a introdução de orientação a objetos nessas disciplinas. Muitas universidades estão mudando seus currículos para a adoção de orientação a objetos como o paradigma a ser ensinado nas disciplinas introdutórias dos cursos de computação. Acredita-se que a orientação a objetos é um paradigma mais intuitivo e traz mais benefícios aos alunos que o paradigma procedimental [11]. No entanto, as visões enganadas que os estudantes possuem em relação à programação continuam existindo, independentemente do paradigma a ser utilizado. Nesse sentido, a atividade de teste pode ajudar no desenvolvimento das habilidades de compreensão e análise nos estudantes pois para que a mesma seja realizada é necessário que os alunos conheçam o comportamento dos seus programas [14].

Dentro dessa perspectiva, a atividade de teste pode ser abordada no escopo de disciplinas fundamentais de computação. Tal abordagem, explorada por Kernighan & Pike em “The Practice of Programming” [22], consiste na introdução de conceitos e práticas de teste concomitantemente com o ensino de conceitos básicos e técnicas de programação logo nos primeiros períodos letivos da formação do aluno.

A atividade de teste de software desempenha um papel importante na garantia da qualidade de software, na medida que consiste em uma atividade que visa a revelar a presença de erros [16, 27]. Segundo Harrold [16], a atividade de teste é uma das mais onerosas do processo de desenvolvimento, chegando a consumir 50 % dos custos. Por outro lado, o conjunto de informações obtido durante os testes é de fundamental importância para as atividades de depuração, manutenção e estimativa de confiabilidade [16].

A qualidade da atividade de teste depende da qualidade da atividade de desenvolvimento, como por exemplo, de inspeção, do estilo da solução/programação, entre outros fatores. Dessa forma, o ensino de programação está intrisicamente ligado ao ensino de teste de software. No entanto, a atividade de teste de software também é pouco explorada nos cursos de graduação, existindo uma forte necessidade da sua inclusão nos currículos dos cursos de graduação [9].

Historicamente, nos currículos dos cursos de graduação em computação, pouca ênfase tem sido fornecida ao ensino de teste de software em comparação com outras atividades do processo de desenvolvimento de software, como por exemplo análise e projeto de sistemas. Este fenômeno acontece porque muitas pessoas acreditam que

análise e projeto são processos “produtivos”, ao contrário de teste de software que seria um processo “destrutivo” no ciclo de vida do software [10]. Particularmente no Brasil, embora existam mais de 600 cursos de Computação/Informática no país, poucos deles possuem uma disciplina especificamente voltada ao ensino dos conceitos relacionados à atividade de teste.

Este artigo apresenta uma proposta para fornecer subsídios para que o ensino de fundamentos de programação seja integrado com teste de software. A proposta é criar material didático integrado de fundamentos de programação e de teste de software e, ainda, construir um ambiente de apoio para a submissão e avaliação automática de trabalhos práticos dos alunos, auxiliando os processos de ensino e aprendizagem. Será utilizada a orientação a objetos, com o uso da linguagem Java. Assim, pretende-se criar uma cultura de teste de software nos alunos e ao mesmo tempo contribuir para práticas de programação mais sistemáticas.

Na Seção 2, questões e problemas relativos ao ensino de programação são apresentadas. Uma visão da atividade de teste de software, bem como a importância dessa atividade em disciplinas introdutórias de programação, é apresentada na Seção 3. Na Seção 4, atividades essenciais para que o ensino de fundamentos de programação e de teste de software sejam integradas são abordadas. A proposta deste artigo é relatada na Seção 4, na qual são discutidas questões relativas à elaboração de material didático e ao ambiente de submissão e avaliação de trabalhos práticos a ser desenvolvido. Um exemplo ilustrando o funcionamento do ambiente a ser desenvolvido é mostrado na Seção 5. Na Seção 6, são apresentadas as conclusões.

2. Ensino de Programação

A abordagem mais tradicional no ensino de disciplinas introdutórias dos cursos de computação é fornecer uma visão geral sobre Ciência da Computação incluindo hardware, software, história da computação, ética, mercado de trabalho na indústria e Internet. Posteriormente, fundamentos de programação são enfatizados, utilizando alguma linguagem de programação de propósito geral, preferencialmente uma linguagem procedimental, como Pascal ou C [17].

Segundo Alphonce, o ensino de introdução à programação possui alguns problemas [3]:

- Os cursos de programação freqüentemente têm seu foco na sintaxe e em características específicas de uma linguagem de programação. Isso leva os estudantes a se concentrarem nesses detalhes ao invés de desenvolver habilidade nos algoritmos. Os alunos, ao se aprofundarem especificamente na programação, acabam vendo a mesma como uma “luta contra o compilador”.
- Para ser acessível aos estudantes novatos, os cursos de programação introdutórios freqüentemente simplificam demais o processo de programação, dando pouca atenção à análise, projeto e testes, com o intuito de simplificar o processo de codificação.

A mudança principal que está ocorrendo no ensino introdutório dos cursos de ciência da computação é tentar fazer com que o mesmo seja interessante e relevante para os estudantes [17]. Nessa perspectiva, uma abordagem investigada é a introdução do paradigma da orientação a objetos em disciplinas introdutórias.

Adams [1] acredita que o ensino de programação orientada a objetos em nível introdutório pode trazer benefícios para os alunos, uma vez que os estudantes aprendem a projetar software que seja mais fácil de codificar, depurar, acarretando em menores custos de manutenção sobre o seu tempo de vida. Para Alphonce [3], a orientação a objetos deve ser ensinada nas disciplinas introdutórias porque os alunos obtêm benefícios nas disciplinas posteriores já que, com o ensino de orientação a objetos, os estudantes aprendem mais facilmente como descrever, projetar e modelar sistemas com uma complexidade maior.

Embora o ensino de orientação a objetos (OO) especialmente utilizando a linguagem Java, esteja se tornando cada vez mais comum, os educadores não têm muita experiência em ensinar orientação a objetos para alunos iniciantes. Ferramentas e material didático de suporte, bem como a experiência dos professores estão em um estágio bastante inicial se comparado ao ensino da programação procedimental [23].

Segundo Cooper [11], o melhor caminho para ajudar os alunos a compreenderem a orientação a objetos é utilizar uma abordagem de ensino em que os conceitos de objetos são ensinados antes de qualquer coisa *(object first)*. Nessa abordagem, as noções de objetos e herança são introduzidos em primeiro lugar. Depois as estruturas de controles tradicionais são ensinadas, mas sempre dentro do contexto de OO.

3. Ensino de Teste de Software

O processo de desenvolvimento de software envolve uma série de atividades nas quais, apesar das técnicas, métodos e ferramentas empregadas, erros no produto ainda podem ocorrer. A atividade de teste consiste em uma análise dinâmica do produto, ou seja, na execução do produto de software com o objetivo de verificar a presença de defeitos e aumentar a confiança de que o mesmo esteja correto, representando a última revisão da especificação, projeto e codificação. Portanto, o teste bem sucedido é aquele que consegue determinar casos de teste para os quais o programa que está sendo testado falhe [16, 24, 27].

O teste de produtos de software envolve basicamente quatro etapas: planejamento de testes, projeto de casos de teste, execução e avaliação dos resultados dos testes. Essas atividades devem ser desenvolvidas ao longo do próprio processo de desenvolvimento de software e, em geral, concretizam-se em três fases. A primeiras delas é o teste de unidade, na qual cada módulo do software é testado individualmente, buscando revelar erros de lógica e fornecer evidências de que o módulo funciona adequadamente. A próxima fase, o teste de integração, é uma técnica sistemática para integrar os módulos componentes da estrutura do software, visando a identificação de erros de interface entre tais módulos. Finalizando, o teste de sistema verifica se todos os elementos do sistema combinam-se adequadamente e se a função/desempenho global do mesmo é atingida na perspectiva dos requisitos do usuário [24, 27].

Para que a atividade de teste possa ser conduzida de forma sistemática e teoricamente fundamentada, faz-se necessária a aplicação de critérios que indiquem como testar o software, quando parar os testes e que, se possível, forneçam uma medida objetiva do nível de confiança e de qualidade alcançados com os testes realizados [13].

Em geral, os critérios de teste são estabelecidos a partir de três técnicas: funcional, estrutural e baseada em erros. Tais técnicas diferenciam-se pela origem da informação

utilizada na avaliação e construção dos conjuntos de casos de teste. Na técnica funcional, os requisitos de teste são estabelecidos a partir da especificação funcional do software; na técnica estrutural, derivam-se os requisitos de teste a partir dos aspectos de implementação do software; na técnica baseada em erros, os elementos requeridos para caracterizar a atividade de teste são baseados em erros comuns que podem ocorrer durante o processo de desenvolvimento de software [24].

Outro aspecto fundamental associado ao teste de software refere-se ao desenvolvimento de ferramentas que automatizem a aplicação das técnicas e critérios de teste. De fato, sem a utilização de ferramentas automatizadas como mecanismos de apoio, a atividade de teste tende a ser extremamente trabalhosa, propensa a erros e limitada a programas muito simples [18]. Além de contribuírem para a qualidade e a produtividade da atividade de teste, ferramentas de teste também são de fundamental relevância no apoio ao ensino de teste de software, pois permitem que os alunos possam realizar na prática os conceitos aprendidos.

A atividade de teste de software é uma das mais importantes na garantia da qualidade e da confiabilidade do software que está sendo desenvolvido. Desta forma, um profissional qualificado em Engenharia de Software deve ter domínio em teste de software. Com o aumento da demanda por profissionais desse tipo no mercado, é fundamental que os alunos comecem a aprender teste o mais cedo possível [5].

Tradicionalmente, teste vem sendo ensinado de acordo com a abordagem clássica de desenvolvimento de software, o modelo cascata [27], na qual as fases de construção do sistema são descritas de uma forma linear — análise, projeto, codificação, teste e manutenção — e os testes acontecem somente no final do processo de desenvolvimento. Geralmente, conceitos mais práticos de teste só são apresentados em cursos avançados e específicos de Engenharia de Software.

No entanto, experiências modernas sugerem que o teste deve ser ensinado durante todas as fases do desenvolvimento de software, e com isso garantir a qualidade do sistema e reduzir custos de desenvolvimento. Um exemplo disso é a metodologia ágil de desenvolvimento de software *eXtreme Programming* [7]. A *eXtreme Programming* é uma metodologia relativamente recente que possui em suas premissas básicas a idéia de "*test first, code later*", ou seja, o projeto de casos de teste deve ser realizado antes mesmo do início da implementação [7].

De acordo com Patterson [26], um fator de fundamental importância para o ensino de teste começar o mais cedo possível é o fato de que um bom testador deve ter habilidades e conhecimentos que só são desenvolvidos na prática. Assim, quanto mais cedo os alunos aprenderem teste, mais tempo eles terão para praticá-los e, com isso, podem se tornar melhores testadores e também desenvolvedores. Desta forma, o teste deve ser ensinado no início das disciplinas introdutórias dos cursos de computação e ser reforçado nas disciplinas subseqüentes [15].

Outras experiências também vêm sendo conduzidas no sentido de ensinar teste o mais cedo possível. Aqueles alunos que aprendem teste mais cedo podem se tornam melhores desenvolvedores, pois o teste força a integração e a aplicação das habilidades do desenvolvedor de software de análise, projeto e implementação [20]. Contudo, uma parte muito pequena dos currículos dos cursos de computação é alocada ao ensino de teste de software, sobretudo concomitantemente com o ensino de programação [28].

4. Atividades Essenciais para Integrar o Ensino de Fundamentos de Programação e de Teste de Software

Ensinar teste de software conjuntamente com fundamentos de programação pode melhorar a formação de profissionais. Acredita-se que os benefícios advindos dessa atividade são relevantes para a formação acadêmica do aluno. Como uma parte muito pequena dos currículos dos cursos de computação é alocada ao ensino do teste de software, pretende-se fornecer subsídios para que esse tipo de ensino se torne natural. Com isso, espera-se criar uma cultura de teste nos alunos logo que eles entram em um curso de computação e que a mesma se mantenha durante a vida profissional desses alunos tanto no meio acadêmico como no industrial. Ao mesmo tempo, contribuir para práticas de programação mais sistemáticas.

Para que a integração do ensino de fundamentos de programação e de teste de software se concretize, duas etapas foram estabelecidas. Na primeira etapa, do ponto de vista didático, problemas relativos ao ensino tanto da atividade de programação quanto da atividade de teste foram identificados. Do ponto de vista prático, foram investigados e selecionados ambientes de programação que forneçam apoio para a integração de fundamentos de programação e de teste de software. Esses ambientes foram avaliados do ponto de vista didático, procurando aqueles que fossem mais adequados para o ensino tanto de programação quanto de teste de software. Após a identificação dos requisitos descritos acima, passou-se para a segunda etapa, na qual propõe-se o desenvolvimento de material didático, e também a construção de um ambiente de submissão e avaliação de trabalhos práticos dos alunos, denominado PROGTEST. As atividades necessárias para a integração do ensino de fundamentos de programação e de teste de software são discutidas a seguir:

1. Identificação dos problemas relacionados à atividade de programação;
2. Identificação dos problemas relacionados à atividade de teste de software;
3. Investigação e seleção de ambientes de programação;
4. Elaboração de material didático integrado;
5. Construção do ambiente PROGTEST.

4.1. Problemas no ensino de programação

Os estudantes de cursos introdutórios de computação possuem visões erradas a respeito da atividade de programação, independentemente do paradigma que é ensinado. Segundo Edwards [14], os alunos possuem as seguintes visões erradas sobre as atividades de programação [14]: *Uma vez que um compilador aceita o código sem reclamar, todos os erros já foram removidos*; *Uma vez que o código produz a saída esperada com um ou dois valores de teste, o trabalho já está pronto*; *O código sempre parece estar correto. Se ele está produzindo valores errados, não faz sentido, ou então o código não foi compreendido*; *Uma vez que o código fornece uma resposta correta para um simples dado do professor, a tarefa está terminada*;

Essas visões erradas que os alunos possuem da atividade de programação, muitas vezes existem pela forma que a programação é ensinada atualmente, em geral baseados na sintaxe de uma linguagem de programação. Os alunos acabam se preocupando mais com detalhes técnicos de uma linguagem de programação do que com os algoritmos em si. Nessa perspectiva, a inserção de atividades de teste de software juntamente com a

programação pode fazer com que os alunos se preocupem mais no desenvolvimento e no entendimento dos seus algoritmos, já que os mesmos terão que ser testados. Logo, a atividade de teste também pode ajudar a acabar com essas visões erradas que os estudantes possuem sobre a atividade de programação.

4.2. Problemas no ensino de teste de software

O ensino de teste de software em níveis introdutórios não é uma tarefa trivial. Existem vários problemas relacionados ao ensino de teste abordados por Patterson [26]:

- Os alunos vêem teste como uma atividade "chata", pois não é criativa;
- Testes gastam muito tempo para serem realizados e os alunos não gostam de passar muito tempo realizando-os;
- O ensino de teste freqüentemente envolve a escrita de longos planos de teste. Os planos de teste são usualmente mais longos que os programas que estão sendo testados. Os alunos acham que a escrita desses planos leva a atrasos nos trabalhos que eles têm a fazer;
- É muito difícil motivar os estudantes a realizar um bom teste. Eles não conseguem enxergar os benefícios da abordagem formal de teste ensinada, uma vez que seus programas são pequenos e simples;
- Ferramentas que dão suporte ao ensino de testes em nível introdutório são raras.

Segundo Edwards [14], existem algumas barreiras na adoção de práticas de teste de software para alunos que estão se iniciando em programação [14]:

1. Teste de software requer experiência em programação e muitas vezes, estudantes novatos não possuem essa experiência;
2. Professores não têm tido tempo (em termos de horas-aula) de ensinar um novo tópico, como teste de software, em cursos já sobrecarregados;
3. Professores gastam muito tempo corrigindo os trabalhos dos alunos, assim eles não estariam disponíveis para avaliar também os casos de teste;
4. Para aprender teste de software os estudantes necessitam de um freqüente e concreto *feedback* de como melhorar sua performance durante o desenvolvimento do seu programa. Recursos para um rápido e eficaz *feedback* durante a escrita de um programa não estão disponíveis na maioria dos cursos;
5. Os alunos devem valorizar tudo o que eles aprendem durante as atividades de programação. O estudante deve enxergar qualquer trabalho extra como uma ajuda no desenvolvimento dos seus programas, e não como um obstáculo imposto pelo professor.

Apesar das limitações observadas, Barriocanal [6] realizou um experimento no sentido de determinar se as práticas de teste ensinadas inicialmente realmente melhoram a qualidade do código implementado pelos alunos e se também ajudam no processo de aprendizado dos alunos. Investigou-se ainda se os alunos gostam ou não, na prática, de realizar testes em seus programas. De acordo com o resultado desse experimento, chegou-se à conclusão que existe uma parcela pequena de estudantes que não estão dispostos a praticar o teste. Grande parte dos alunos acharam a idéia válida e apesar de reconhecerem que a atividade de teste é um pouco maçante, eles também concordaram que o teste trouxe benefícios, tanto na melhoria da qualidade dos programas construídos como na assimilação dos conceitos de programação ensinados no curso.

Outros trabalhos estão sendo realizados no sentido de se construir ambientes que dêem suporte ao ensino do teste. Isso está acontecendo tanto na direção da construção de ambientes de programação que facilitem a construção de casos de teste apoiando a realização da atividade de teste de unidade, como 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.

4.3. Seleção de Ambientes de Programação Pedagógicos

Os ambientes integrados de desenvolvimento profissionais são projetados para o desenvolvimento de grandes sistemas. Freqüentemente, tais ambientes contêm muitas características avançadas, como por exemplo, o projeto gráfico de interfaces. Embora a grande quantidade de características avançadas nesses ambientes seja útil para quem já sabe programar, a mesma pode ser inapropriada para usuários que estão aprendendo [26].

Nessa perspectiva, tem-se investigado a construção de ambientes integrados de desenvolvimento pedagógicos com interfaces intuitivas que sejam fáceis para o aprendizado de estudantes novatos. Além disso, esses ambientes também têm procurado fornecer apoio para a atividade de teste de unidade, facilitando a realização deste por parte dos alunos. Entre os ambientes pedagógicos encontrados, destacam-se o DrJava [2] e o BlueJ [26]. O DrJava é um ambiente de programação pedagógico para Java que possui uma interface que permite a interação do aluno com o código que ele está construindo e também fornece suporte à atividade de teste pois é integrado ao JUnit[1]. O BlueJ é um ambiente de programação pedagógico para Java desenvolvido especificamente para facilitar o ensino de programação orientada a objetos pois permite a interação direta dos alunos com as classes criando objetos e realizando a chamada a seus métodos graficamente. O BlueJ também é integrado ao JUnit e possui um mecanismo que permite a criação interativa de casos de teste no JUnit fornecendo suporte à atividade de teste.

Esses ambientes apóiam o ensino de OO e facilitam a construção de casos de teste no JUnit podendo ser utilizados pelos alunos para auxiliar o ensino integrado de fundamentos de programação e de teste de software. Entretanto, ressalta-se que esses ambientes não são integrados com ferramentas de teste de análise de cobertura. Eles apenas facilitam a construção de casos de teste no JUnit.

4.4. Material Didático

Material didático de teste de software e de fundamentos de programação no paradigma OO será elaborado para apoiar o ensino integrado de programação e de teste de software. A elaboração de material didático e sua posterior evolução não é uma tarefa trivial em virtude da complexidade e da diversidade dos fatores envolvidos. A simples utilização de recursos computacionais não garante necessariamente o sucesso e a eficácia da aprendizagem. A qualidade do material didático utilizado - o qual deve ser elaborado levando-se em consideração aspectos tais como o domínio de conhecimento, o perfil de aprendizado dos alunos, os objetivos do curso e as estratégias pedagógicas adotadas - é um fator determinante nesse sentido. Nessa perspectiva, é de fundamental importância para o

[1] JUnit é um framework de código aberto, utilizado para a realização de testes de unidade de programas Java [25]

ensino integrado de fundamentos de programação e de teste de software que material didático de qualidade seja desenvolvido. Neste trabalho serão utilizados o processo padrão para elaboração/evolução de material didático e os mecanismos de modelagem de conteúdos educacionais investigados por Barbosa [4] para a elaboração deste material didático.

4.5. PROGTEST: Ambiente de Submissão e Avaliação de Trabalhos Práticos

Um dos fatores críticos para o sucesso do ensino integrado de teste de software e de fundamentos de programação é promover um *feedback* apropriado e avaliar a performance dos estudantes. Neste caso, haverá uma dupla carga de trabalho, uma vez que, tanto os casos de teste quanto o código do programa terão que ser avaliados. Uma solução para esse problema crítico é o desenvolvimento e utilização de ambientes automatizados para a avaliação de trabalhos práticos, o que poderá reduzir a sobrecarga que existe sobre os professores [14].

Além disso, o uso de um ambiente de avaliação automática de trabalhos práticos traz benefícios adicionais em termos de consistência, eficácia e eficiência. Todo programa submetido é analisado no mesmo nível de eficiência e os resultados da avaliação são baseados nos mesmos padrões. A avaliação tem que ser realizada independentemente do professor ou de qualquer outro efeito externo. Um benefício desses ambientes é o fornecimento de relatórios, após a avaliação, de cada programa dos estudantes. Assim, os alunos podem saber como está a sua performance [19].

Alguns ambientes para submissão e avaliação automática de trabalhos práticos foram propostas no sentido de facilitar o ensino integrado de teste e de fundamentos de programação, como [15, 19, 21, 29]. Esses ambientes recebem os programas desenvolvidos pelos alunos (alguns deles também aceitam casos de teste) e realizam comparações do resultado do programa feito pelo aluno com o resultado esperado. Desta forma, ocorre um julgamento se o programa do aluno está correto ou não.

Nesse sentido, a fim de apoiar o ensino de fundamentos de programação e de teste de software, propõe-se a construção de um ambiente de submissão e avaliação automática de trabalhos práticos, denominado PROGTEST. Este ambiente deverá avaliar tanto os programas quanto os casos de teste fornecidos pelos alunos. Assim, serão analisadas tanto a qualidade do programa quanto a atividade de teste realizada pelos estudantes. Para isso, ferramentas de teste devem ser integradas ao ambiente para permitir que seja realizado uma análise de cobertura baseada em critérios de teste. Desta forma, o enfoque do ambiente PROGTEST está na performance da atividade de teste realizada pelos alunos, ao contrário da maioria dos ambientes para submissão e avaliação automática de trabalhos práticos [15, 19, 21, 29] existentes cujo enfoque está na saída dos programas produzidos pelos alunos.

Na Figura 1 são ilustradas as principais características associadas ao ambiente. Basicamente, dados um programa P_{Al_i} (fornecido pelo aluno) e seu respectivo conjunto de casos de teste T_{Al_i} (gerado com base no critério de teste C_K previamente estabelecido - T_{Al_i} é $C_{K-adequado}$), o ambiente, integrado às ferramentas de teste pertinentes, deve ser capaz de:

1. executar o programa P_{Al_i} com os conjuntos de casos de teste fornecidos pelo professor (T_{Inst});

2. utilizar o conjunto de casos de teste T_{Al_i} para testar o programa “oráculo” P_{Inst}, fornecido pelo professor;
3. comparar o comportamento de P_{Inst} executado com o conjunto de casos de testes T_{Inst} em relação a P_{Al_i} executado com o conjunto de casos de testes T_{Inst}; e
4. comparar o comportamento de P_{Inst} executado com o conjunto de casos de testes T_{Al_i} em relação a P_{Al_i} executado com o conjunto de casos de testes T_{Al_i}.

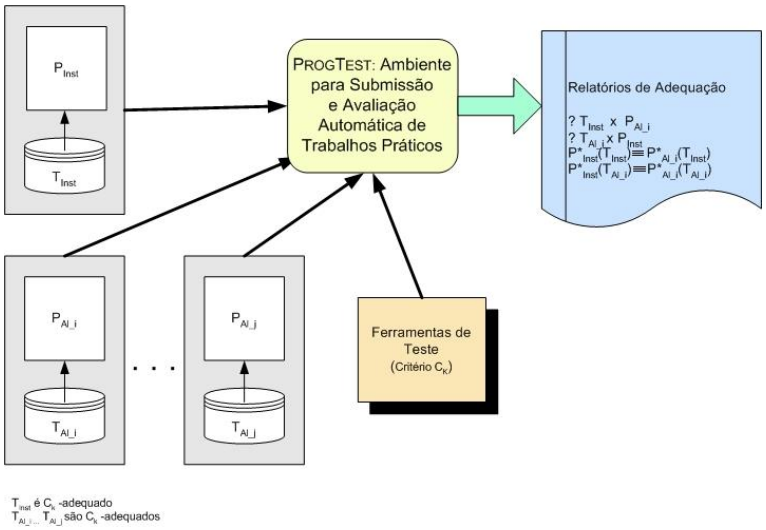

Figura 1: PROGTEST– Ambiente para Submissão e Avaliação Automática de Trabalhos Práticos

A partir das execuções realizadas, o sistema deve ser capaz de decidir pela aceitação ou não do programa e/ou dos casos de teste fornecidos pelo aluno, sugerindo eventualmente uma “nota” aos trabalhos submetidos, tendo como base os índices de cobertura dos conjuntos de casos de teste utilizados. Ferramentas de teste servirão como suporte à aplicação dos critérios de teste pré-estabelecidos e à avaliação da cobertura dos conjuntos de casos de teste, obtida a partir da execução dos programas fornecidos. Ressalta-se ainda que o ambiente será baseado na Web, na concepção de software livre.

Como ferramenta de teste a ser integrada no ambiente, optou-se pela ferramenta de teste JaBUTi (*Java Bytecode Understanding and Testing*) [30]. Trata-se de um ambiente completo para o entendimento e teste de programas e componentes Java desenvolvido no ICMC/USP. A JaBUTi fornece ao testador diferentes critérios de teste estruturais para a análise de cobertura, um conjunto de métricas para avaliar a complexidade das classes que compõem do programa/componente, e implementa ainda algumas heurísticas de particionamento de programas que visam a auxiliar a localização de falhas [30]. Na Figura 2 está ilustrado um programa que ainda não foi testado na ferramenta de teste JaBUTi e as diferentes cores estabelecidas para este programa antes da realização da análise de cobertura. Geralmente, como existe um grande número de requisitos de teste para serem cobertos, a ferramenta JaBUTi utiliza cores diferentes - diferentes pesos que são associados aos requisitos de teste de cada critério - dando dicas ao testador para facilitar a geração de casos de teste que cubram um maior número de requisitos de teste em menor tempo [30].

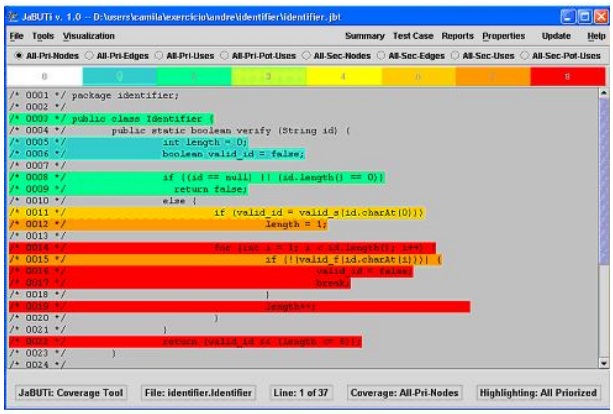

Figura 2: Ferramenta de teste – JaBUTi

5. Exemplo

A seguir, um exemplo será utilizado para mostrar como o ambiente PROGTEST funcionará. Alguns estudantes implementaram, em Java, um programa simples e juntamente aplicaram as técnicas de teste de software funcional e estrutural. Os estudantes tinham que desenvolver um conjunto de casos de testes funcionais e analisar a cobertura obtida com a massa de testes que eles desenvolveram. Se o conjunto de casos de teste não fosse adequado, casos de teste deviam ser adicionados de acordo com os critérios estruturais. O objetivo era atingir a cobertura total de 100% durante a realização da atividade de teste. Para realização da análise de cobertura, os alunos utilizaram a ferramenta de teste JaBUTi. Os casos de teste foram desenvolvidos utilizando o JUnit [25]. Os alunos tiveram que entregar o código fonte do programa (arquivo .java) e o arquivo .class e o conjunto de todos os casos de teste desenvolvidos.

Após a entrega por parte dos alunos do código fonte dos programas e do conjunto de casos de teste desenvolvidos, foi realizada uma análise simulando as principais características que se espera desse ambiente, como foi descrito na Seção 4.5. Para realização dessa análise existia uma implementação e um conjunto de casos de teste do professor que foi utilizado como base para a comparação com os programas dos alunos.

Utilizando a ferramenta de teste JaBUTi, o programa de cada aluno foi executado com seus próprios casos de teste e com os casos de teste do professor, sendo que os dados referentes à cobertura atingida foram armazenados. A realização dessa atividade permitiu analisar a qualidade do código escrito pelo aluno. A meta era atingir 100% de cobertura com os casos de teste do professor. O fato da cobertura de 100% não ser atingida, pode indicar que o código contenha caminhos não-executáveis [24]. A existência de caminhos não-executáveis em um programa pode ser um indicativo de que o aluno possui práticas de programação que não são de boa qualidade. Assim, um dos parâmetros que foi utilizado na comparação comportamental dos programas produzidos pelos alunos foi a quantidade de caminhos não-executáveis, ou seja, quanto menos caminhos executáveis existirem maior a qualidade do código implementado.

Posteriormente, os casos de testes dos alunos foram aplicados no programa do professor. A realização dessa atividade permitiu que a qualidade do conjunto de casos de teste construído pelos alunos fosse analisada. O indicativo de qualidade da atividade de

teste realizada pelo aluno foi a cobertura obtida no programa do professor.

Os programas submetidos pelos alunos foram aceitos se o índice de cobertura obtido pelos seus programas com os casos de teste do professor e dos seus casos de teste no programa do professor fossem superiores a 80% no critério de teste estabelecido pelo professor.

Analisando os dados obtidos, percebeu-se que a cobertura obtida pelos alunos com seus próprios casos de teste quase chegou a 100%. Alguns alunos conseguiram atingir 100% de cobertura em algum critério, mas nenhum atingiu 100% em todos os critérios de teste cobertos pela JaBUTi. O fato de os alunos não terem conseguido atingir o máximo de cobertura pode indicar que eles não conseguiram realizar a atividade de teste direito, ou porque não entenderam os conceitos ensinados ou o seu próprio código.

Com relação à execução dos programas dos alunos com os casos de teste do professor, a maioria dos alunos conseguiram atingir 100% de cobertura em algum critério de teste, mas não em todos. Analisando o código dos alunos, percebeu-se que todos eles tinham caminhos não-executáveis, alguns com mais e outros com menos, o que indica que talvez as práticas de programação utilizadas pelos alunos não sejam de boa qualidade.

Na análise do programa do professor com os casos de teste dos alunos os resultados obtidos não foram tão bons quanto os outros. A maioria dos alunos não conseguiram obter a cobertura de 100% em todos os critérios. E alguns deles obtiveram índices de cobertura abaixo de 75%. Este fato pode indicar que a qualidade da massa de testes produzida pelos alunos não está boa, podendo ser em virtude de uma falta de entendimento dos conceitos ensinados ou da realização da atividade de teste de forma inadequada.

Este exemplo é meramente ilustrativo e serve para mostrar o quanto a automatização é necessária na avaliação dos trabalhos dos alunos, devendo o mesmo ser reproduzido quando o ambiente estiver pronto para que dados mais concretos sejam obtidos. A definição dos parâmetros que serão utilizados na comparação do comportamento dos programas não é uma tarefa trivial. A utilização dos índices de cobertura atingidos, dos caminhos não-executáveis é apenas uma tentativa inicial para simular as características do ambiente de submissão e avaliação de trabalhos práticos. Estudos estão sendo realizados no trabalho de mestrado de Corte [12] para definir os parâmetros que serão utilizados na comparação dos programas dentro do ambiente.

6. Conclusões

Neste artigo discutiu-se a relevância do ensino conjunto de fundamentos de programação e de teste de software. Do ponto de vista da programação, a atividade de teste pode contribuir para a melhorar a capacidade de compreensão e análise dos alunos. Do ponto de vista da atividade de teste, se a mesma for ensinada juntamente com fundamentos de programação, pode ser criada uma cultura de teste nos alunos, tornando-se uma prática do dia-a-dia de um profissional de computação.

Nesse sentido, algumas atividades essenciais foram realizadas: identificação dos problemas inerentes tanto ao ensino de programação como ao de teste de software; investigação e seleção de ambientes de apoio à atividade de programação e à atividade de teste. O exemplo demonstrado neste artigo é meramente ilustrativo, devendo ser

reproduzido quando o ambiente estiver pronto para a obtenção de dados mais concretos. Atualmente, as atividades referentes à elaboração de material didático integrado de fundamentos de programação e de teste de software e a construção do ambiente PROGTEST estão sendo realizadas.

Referências

[1] J. Adams and J. Frens. Object centered design for Java: Teaching OOD in CS1. In *34th SIGCSE Technical Symposium on Computer Science Education (SIGCSE'03)*, volume 35, pages 273–277, Reno, Nevada, USA, 2003.

[2] E. Allen, R. Cartwright, and B. Stoler. DrJava: A lightweight pedagogic environment for Java. In *33rd ACM SIGCSE Technical Symposium on Computer Science Education (SIGCSE'02)*, volume 34, pages 137–141, Cincinnati, Kentucky, 2002.

[3] C. Alphonce and P. Ventura. Object orientation in CS1-CS2 by design. In *7th Annual Conference on Innovation and Technology in Computer Science Education (ITiCSE'02)*, volume 34, pages 70–74, Aarhus, Denmark, 2002.

[4] E. F. Barbosa. *Uma Contribuição ao Processo de Desenvolvimento e Modelagem de Módulos Educacionais*. Tese de doutoramento, ICMC/USP, São Carlos, SP, Maio, 2004.

[5] E. F. Barbosa, J. C. Maldonado, R. Leblanc, and M. Guzdial. Introducing testing practices into objects and design course. In *16th Conference on Software Engineering Education and Training (CSEE&T'03)*, pages 279–286, Madrid, Spain, 2003.

[6] E. G. Barriocanal, M. A. S. Urbán, I. A. Cuevas, and P. D. Pérez. An experience in integrating automated unit testing practices in an introductory programming course. In *ACM SIGCSE Bulletim*, volume 34, pages 125–128, 2002.

[7] K. Beck. *Extreme Programming Explained: Embrace Change*. Addison Wesley, 1999.

[8] D. Buck and D. J. Stucki. Design early considered harmful: graduated exposure to complexity and structure based on levels of cognitive development. In *31 st SIGCSE Technical Symposium Computer Science Education*, pages 16–20, 2000.

[9] T. Y. Chen. Experience with teaching black-box testing in a computer science/software engineering curriculum. *IEEE Transactions on Education*, 47(01):42–50, 2004.

[10] T. Y. Chen and P. L. Poon. Teaching black box testing. In *IEEE International Conference Software Engineering: Education and Practice*, pages 324–329, 1998.

[11] S. Cooper, W. Dann, and R. Pausch. Teaching objects-first in introductory computer science. In *34th SIGCSE Technical Symposium on Computer Science Education (SIGCSE'03)*, volume 35, pages 191–195, Reno, Navada, USA, 2003.

[12] C. K. Della Corte. Ensino integrado de teste de software e fundamentos de programação. Exame de qualificação, ICMC/USP, São Carlos - SP, 2004. Mestrado (em curso).

[13] R. A. Demillo. Mutation analysis as a tool for software quality assurance. In *4th IEEE International Computer Software and Applications Conference (COMPSAC'80)*, pages 390–393, Los Alamitos, CA,, 1980.

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

[15] M. H. Goldwasser. A gimmick to integrate software testing throughout the curriculum. In *33rd SIGCSE Technical Symposium on Computer Science Education (SIGCSE'02)*, volume 34, pages 271–275, Cincinnati, Kentucky,, 2002.

[16] M. J. Harrold. Testing: A roadmap. In *22nd International Conference on Software Engineering (ICSE'00), Future of Software Engineering Track*, pages 61–72, Limerick, Irlanda, 2000.

[17] T. J. Hickey. Scheme-based web programming as a basis for a CS0 curriculum. In *35th SIGCSE Technical Symposium on Computer Science Education*, pages 353–357, Norfolk, Virginia, USA, 2004.

[18] J. R. Horgan and A. P. Mathur. Assessing testing tools in research and education. *IEEE Software*, 09(03):61–69, 1992.

[19] J. Isong. Developing an automated program checker. *The Journal of Computing in Small Colleges (JCSC)*, 16(03):218–224, 2001.

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

[21] E. L. Jones. Grading student programs - a software testing approach. In *4th Annual Consortium on Small Colleges Southeastern Conference*, volume 16, pages 185–192, Salem, Virginia, 2001.

[22] B. W. Kernighan and R. Pike. *The Practice of Programming*. Addison Wesley, 1999.

[23] M. Kolling and J. Rosenberg. Guidelines for teaching object orientation with java. In *6th Annual Conference on Innovation and Technology in Computer Science Education (ITiCSE'01)*, volume 33, pages 33–36, Canterbury, UK, 2001.

[24] J. C. Maldonado. *Critérios Potenciais Usos: Uma Contribuição ao Teste Estrutural de Software*. Tese de doutoramento, DCA/FEEC/UNICAMP, Campinas, SP, 1991.

[25] Inc Object Mentor. JUnit, testing resources for extreme programming. Disponível em http://www.junit.org/index.htm, Último acesso em 20/06/2003.

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

[27] R. S. Pressman. *Software Engineering - A Practitioner's Approach*. McGraw-Hill, 5th edition, 2001.

[28] T. Shepard, M. Lamb, and D. Kelly. More testing should be taught. *Communications of the ACM*, 44(06):103–108, 2001.

[29] A. Trivedi, D. C. Kar, and H. Patterson-McNeill. Automatic assignment management and peer evaluation. *The Journal of Computing in Small Colleges (JCSC)*, 18(04):30–37, 2003.

[30] A. M. R. Vincenzi. *Orientação a Objetos: Definição e Análise de Recursos de Teste e Validação*. Tese de doutoramento, ICMC/USP, São Carlos, SP, Maio, 2004.