Ensino integrado de fundamentos de
programação e teste de software

Camila Kozlowski Della Corte

SERVIÇO DE PÓS-GRADUAÇÃO DO ICMC-USP

Data de Depósito: 4 de maio de 2006

Assinatura: _____

Ensino integrado de fundamentos de programação e teste de software

Camila Kozłowski Della Corte

Orientador: *Prof. Dr. José Carlos Maldonado*

Dissertação apresentada ao Instituto de Ciências Matemáticas e de Computação — ICMC/USP como parte dos requisitos para obtenção do título de Mestre em Ciências de Computação e Matemática Computacional.

USP - São Carlos
Maio/2006

*“Comece fazendo o que é necessário,
depois o que é possível,
e de repente você estará fazendo o impossível.”
(São Francisco de Assis)*

Agradecimentos

Agradeço a Deus por todos os momentos de força e esperança.

À minha família, em especial à minha mãe Mathilde e ao meu pai Marcantônio, por todo amor, carinho, apoio, incentivo incondicionais e compreensão pela distância necessária para a realização do mestrado. Aos meus pais obrigado por tudo, inclusive pelo apoio financeiro!

À minha irmã, Marina pelo amor e carinho sempre presentes mesmo com a distância.

Ao Claudio pelo amor, carinho, apoio e incentivo durante todo o período de realização deste trabalho.

Ao meu orientador, Prof. Dr. José Carlos Maldonado pela orientação, confiança e incentivo durante esse trabalho. Sua experiência e sugestões foram essenciais para a realização deste trabalho.

À Profa. Dra. Ellen Francine Barbosa pelo apoio, colaboração, parceria nos trabalhos, e sobretudo, pela amizade conquistada nesses anos.

Ao Prof. Dr. Auri M. R. Vincenzi e ao Prof. Dr. Adenilso Simão pela ajuda e sugestões a este trabalho.

A todos que são ou foram LABES, pelas horas de descontração, alegria e incentivo. Tentando nomear (por ordem alfabética), mesmo que arriscando esquecer alguém, aos amigos: Adenilso, Andrea, André Luís, André Rocha, Antonielly (Tonton), Alessandra, Auri, Bira, Darley, Débora, Ellen, Elisa, Erika, Fabiano, Luciana, Marcella, Marcelo Eler, Marco, Maris, Otávio, Paula, Reginaldo, Rogério, Sandro (KLB), Stanley, Simone Domingues, Tati e Valter. E a todos os amigos dos demais laboratórios!

Ao meu amigo Bira (Birinha) pela amizade, parceria durante o mestrado e também por estar sempre pronto a me auxiliar em quaisquer momentos. À Erika, sempre prestativa, pelas palavras de apoio, incentivo e amizade sempre presentes. À Simone Domingues pela amizade conquistada nesse último ano de mestrado e também pelo apoio com a interface gráfica da ferramenta. Ao Fabiano pelo apoio em vários momentos desse trabalho, por estar sempre pronto a ajudar e por sua amizade.

Aos meus amigos de Goiânia, em especial ao André Constantino, que sempre me incentivou a fazer mestrado.

A todos funcionários, professores e alunos que participaram e colaboraram de alguma maneira na realização deste trabalho.

À FAPESP, pelo apoio financeiro.

Resumo

O ensino das disciplinas de fundamentos de programação e teste de software não é trivial. Várias iniciativas têm sido investigadas a fim de amenizar os problemas associados a essa atividade. Uma dessas iniciativas é o ensino conjunto de conceitos básicos de programação e de teste de software em disciplinas introdutórias dos cursos de Ciências da Computação. Este trabalho insere-se nessa perspectiva, tendo como principal objetivo fornecer subsídios para o ensino integrado de fundamentos de programação OO e teste de software.

Para atingir esse objetivo, foi desenvolvido um módulo educacional integrado de fundamentos de programação OO e teste de software, utilizando uma abordagem integrada para modelagem de conteúdos educacionais. Ênfase é dada na modelagem dos aspectos conceituais, instrucionais e didáticos dos domínios de conhecimento envolvidos. Além disso, também foi desenvolvido um ambiente, denominado PROGTEST, baseado na *Web* e na concepção de software livre, para submissão e avaliação de trabalhos práticos dos alunos. O ambiente foi integrado com ferramentas de teste de modo que a avaliação dos trabalhos seja realizada com base em atividades de teste. Os programas e os resultados são avaliados com base em padrões de consistência, eficácia e eficiência uniformes. Os professores também podem ter uma redução na sua carga de trabalho, uma vez que o sistema avalia automaticamente tanto os casos de teste quanto o código do programa.

Abstract

The teaching of fundamentals of programming and software testing is not trivial. Many attempts have been investigated in order to reduce the problems associated to this activity. One of these initiatives is the teaching of basic concepts and techniques of programming in conjunction with software testing in introductory courses of the Computer Science. In this sense, this work aims at providing mechanisms to the integrated teaching of fundamentals of OO programming and software testing.

To achieve this goal, we developed an integrated educational module of fundamentals of OO programming and software testing, using an integrated approach to model the educational contents. We focus on conceptual, instructional and didactic modeling of the knowledge domains. Furthermore, an open source Web-based environment, named ProgTest, was also developed for the submission and evaluation of practical assignments to the students. The environment was integrated with testing tools, so the evaluation of the assignments is performed based on testing activities. The assignments and the results are evaluated based on the uniform standards in terms of consistence, efficacy and efficiency. The professors can have their workload reduced with this environment, once the system automatically evaluates both the code of the assignment as well as its test cases.

Sumário

1	Introdução	1
1.1	Contexto e Motivação	1
1.2	Objetivos	3
1.3	Organização do Trabalho	4
2	Revisão Bibliográfica	5
2.1	Considerações Iniciais	5
2.2	Orientação a Objeto	6
2.2.1	Ensino de Programação OO	7
2.3	Teste de Software	9
2.3.1	Problemáticas do Teste de Software OO	11
2.3.2	Fases do Teste de Software OO	12
2.3.3	Técnicas e Critérios de Teste de Software OO	13
2.3.3.1	Técnica funcional	15
2.3.3.2	Técnica estrutural	16
2.3.3.3	Técnica baseada em erros	19
2.3.4	Ferramentas de Teste	21
2.3.4.1	JaBUTi	23
2.3.5	Ensino de Teste de Software	24
2.4	Ambientes Educacionais	27
2.4.1	Ambientes de Apoio ao Ensino de Teste de Software	29
2.4.1.1	Ambientes de Programação Pedagógicos	29
2.4.1.2	Ambientes de Submissão e Avaliação de Trabalhos	30
2.5	Mecanismos para o Desenvolvimento de Módulos Educacionais	31
2.5.1	Processo Padrão para Elaboração de Módulos Educacionais	32
2.5.2	Modelagem de Conteúdos Educacionais	34
2.5.2.1	Modelo Conceitual	35
2.5.2.2	Modelo Instrucional	36
2.5.2.3	Modelo Didático	37
2.6	Considerações Finais	38

3	Módulo Educacional Integrado de Orientação a Objetos e Teste de Software	41
3.1	Considerações Iniciais	41
3.2	Processo Padrão Instanciado para Elaboração de Módulos Educacionais	42
3.3	Evolução da Abordagem <i>AIM-CID</i>	42
3.4	Módulo Educacional Integrado: Modelagem dos Conteúdos	43
3.4.1	Modelo Conceitual	44
3.4.1.1	Modelo Conceitual: Programação OO	44
3.4.1.2	Modelo Conceitual: Teste de Software	46
3.4.1.3	Modelo Conceitual Integrado	48
3.4.2	Modelo Instrucional	49
3.4.3	Modelo Didático	55
3.5	Módulo Educacional Integrado: Aspectos Gerais da Implementação	58
3.6	Considerações Finais	61
4	ProgTest: Ambiente de Submissão e Avaliação de Trabalhos Práticos	63
4.1	Considerações Iniciais	63
4.2	Motivação para o Ambiente PROGTEST	64
4.3	Abordagem para Avaliação de Programas	64
4.4	Arquitetura do Ambiente PROGTEST	70
4.5	Aspectos de Modelagem	73
4.6	Aspectos de Implementação	78
4.7	Aspectos Operacionais	80
4.8	Considerações Finais	88
5	Conclusão	91
5.1	Considerações Gerais	91
5.2	Contribuições	93
5.3	Trabalhos Futuros	94
	Referência Bibliográfica	96
A	Scripts utilizados na integração da JaBUTi com o ambiente PROGTEST	107

Lista de Figuras

2.1	Relacionamento entre teste de unidade, de integração e de sistema: programas procedimentais e OO (Vincenzi, 2004)	14
2.2	Estrutura Geral do Processo Padrão para Módulos Educacionais (Barbosa, 2004)	33
2.3	AIM-CID: Abordagem Integrada para Modelagem de Conteúdos Educacionais (Barbosa, 2004)	35
3.1	Modelo Conceitual de Orientação a Objetos	45
3.2	Modelo Conceitual de Teste de Software	47
3.3	Modelo Conceitual Integrado de Programação e Teste de Software	48
3.4	Transparência sobre Conceitos Básicos de programação OO	50
3.5	Modelo Instrucional Integrado: Conceitos Básicos	51
3.6	Transparências referentes ao conceito de Objeto	52
3.7	Transparências referentes ao Exemplo e Exercício sobre Terminologia de Teste	53
3.8	Modelo Instrucional Integrado: Comandos Essenciais	54
3.9	Transparências referentes ao Critério Todos-Nós	55
3.10	Modelo Didático Integrado: Conceitos Básicos	57
3.11	Transparências representando o estados <i>DD</i> do Modelo Didático	58
3.12	Módulo Orientação a Objetos e Teste de Software	60
3.13	Cenário de Navegação através dos módulos de Programação e Teste de Software	61
4.1	PROGTEST– Ambiente para Submissão e Avaliação Automática de Trabalhos Práticos	65
4.2	Arquitetura <i>Model-View-Controller</i> MVC do ambiente PROGTEST	71
4.3	Camada de Modelo do ambiente PROGTEST	71
4.4	Parte do Mapa de Navegação - Visão Professor	74
4.5	Parte do Mapa de Navegação - Visão Aluno	75
4.6	Diagrama de Seqüência - Avaliação de Trabalhos	76
4.7	Modelo Relacional	77
4.8	Diagrama de Classes	77
4.9	PROGTEST - Tela Inicial	81
4.10	PROGTEST - Cadastro de Usuários	81
4.11	PROGTEST - Lista de Turma Professor	82
4.12	PROGTEST - Lista de Trabalho Professor	83
4.13	PROGTEST - Cadastro de Trabalhos	84
4.14	PROGTEST - Quadro Geral Notas da Turma	85

4.15	PROGTEST - Histórico e Modificação de Notas do Aluno	85
4.16	PROGTEST - Lista de Trabalhos Aluno	86
4.17	PROGTEST - Histórico do Aluno	87
4.18	PROGTEST - Submissão de Trabalhos	87
4.19	PROGTEST - Relatório de Avaliação dos Trabalhos	88

Lista de Tabelas

2.1	Relação entre as fases de teste e o teste de software OO considerando o método como a menor unidade (Vincenzi, 2004)	13
2.2	- Relação entre as fases de teste e o teste de software OO considerando a classe como a menor unidade (Vincenzi, 2004)	14
4.1	Relatório do JUnit (Teste Funcional)	67
4.2	Relatório de Cobertura obtido com a execução na JaBUTi (Teste Estrutural) . .	67
4.3	Valores Ajustados da Análise de Cobertura	69
4.4	Relatório de Cobertura Geral (Execuções Trocadas)	69
4.5	Esforço da implementação da camada de modelo do ambiente PROGTST . . .	78

Introdução

1.1 Contexto e Motivação

Neste trabalho discute-se a relevância do ensino conjunto de fundamentos de programação e de teste de software. Tanto a programação como a atividade de teste de software não são disciplinas triviais de serem ensinadas. Considerando o ensino da disciplina de fundamentos de programação, 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). Atualmente, a atividade de programação está sendo ensinada com base na abordagem da tentativa e erro (Edwards, 2004). 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 (Alphonse e Ventura, 2002).

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, as quais devem ser pré-requisitos para o ensino de programação, não são, em geral, ensinadas aos alunos. Os estudantes devem desenvolver essas habilidades de modo a conseguir solucionar melhor os problemas, ler e compreender o código fonte, prever como a sequência de comandos do programa irá se comportar e, ainda, prever como mudanças no código poderão resultar na mudança de comportamento do programa (Buck e Stucki, 2000).

Esse cenário acaba gerando problemas no ensino da atividade de programação. Assim, uma das principais mudanças que vêm ocorrendo nos cursos introdutórios de programação refere-se à introdução de orientação a objetos (OO) nessas disciplinas a fim de tornar as mesmas interessantes e relevantes para os alunos. Muitas universidades estão mudando seus currículos no sentido de adotar 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 (Cooper et al., 2003). No entanto, é importante observar que as visões erradas que os estudantes possuem em relação à programação continuam existindo, independentemente do paradigma utilizado.

Uma das iniciativas que têm sido investigadas para tentar amenizar os problemas existentes no ensino de programação refere-se à introdução dos conceitos de teste de software em conjunto com fundamentos de programação em cursos introdutórios de Ciências da Computação (Barbosa et al., 2003a; Barriocanal et al., 2002; Edwards, 2004; Patterson et al., 2003). A introdução da atividade de teste pode ajudar no desenvolvimento das habilidades de compreensão e análise nos estudantes, já que para que a mesma seja realizada é necessário que os alunos conheçam o comportamento dos seus programas (Edwards, 2004).

Teste de software é o processo de execução de um programa com a intenção de encontrar erros (Myers et al., 2004). No contexto da Engenharia de Software, teste é uma das atividades mais importantes para garantir a qualidade e a confiabilidade do software que está em desenvolvimento. Por outro lado, trata-se de um tópico difícil para aprender ou ensinar sem o suporte apropriado. A atividade de teste de software é pouco explorada nos cursos de graduação, existindo a necessidade da sua inclusão nos currículos dos cursos de graduação (Chen, 2004).

Tradicionalmente, a atividade de teste tem sido ensinada no final do processo de aprendizagem dos cursos de Ciências da Computação. Além disso, somente uma pequena parte do currículo do curso de Ciências da Computação é alocado para a atividade de teste em comparação a outras atividades do processo de desenvolvimento de software (Shepard et al., 2001). 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 (Chen e Poon, 1998).

Por outro lado, experiências recentes têm sugerido que a atividade de teste poderia ser ensinada o mais cedo possível aos alunos. A introdução o quanto antes de conceitos e técnicas de teste pode: (1) melhorar o raciocínio sobre os programas (e suas respectivas soluções), o que leva a uma melhor qualidade dos produtos desenvolvidos; e (2) induzir e facilitar o uso de técnicas de teste através do processo de desenvolvimento de software, levando com isso a uma melhoria na qualidade do processo, em contraste com as práticas atuais.

No entanto, ensinar teste de software em níveis introdutórios não é uma tarefa trivial. Existem vários problemas relacionados que vão desde a dificuldade de motivar os alunos a realizarem uma boa atividade de teste até a falta de ferramentas de suporte ao ensino de teste em nível

introdutório (Patterson et al., 2003). É importante destacar que um ambiente que avalie automaticamente os programas e os casos de teste dos alunos pode ajudar a diminuir o trabalho que os professores têm ao ensinar teste, uma vez que, eles não terão que avaliar também os casos de teste dos alunos. Entretanto, iniciativas mostram que apesar desses problemas, os alunos podem obter benefícios aprendendo teste em disciplinas introdutórias de programação (Barriocanal et al., 2002).

Dentro dessa perspectiva, visto que a atividade de teste requer que os alunos conheçam o comportamento dos seus programas, ela pode ser explorada para ajudar aos alunos a compreenderem os conceitos abstratos de programação e desenvolver as habilidades esperadas (Edwards, 2004). Além disso, uma vez que o teste força a integração e aplicação de teorias e habilidades de análise, projeto e implementação, os alunos que aprendem teste mais cedo podem se tornar melhores testadores e desenvolvedores (Barbosa et al., 2003a; Jones, 2001a; Patterson et al., 2003).

De fato, o ensino conjunto de fundamentos de programação e de teste de software pode contribuir tanto para a melhoria do ensino de programação quanto para o ensino de teste de teste. No caso da programação, a atividade de teste pode melhorar a capacidade de compreensão e análise dos alunos. E no caso da atividade de teste de software pode-se tentar criar uma “cultura de teste de software” adequada entre os alunos (e desenvolvedores).

É importante ressaltar, ainda, que a integração do ensino de teste e de programação pode conduzir a uma reforma nos currículos de computação, motivando a introdução o mais cedo possível de aspectos de qualidade no processo de desenvolvimento de software.

1.2 Objetivos

A idéia principal desse trabalho de mestrado é fornecer mecanismos para que o ensino de fundamentos de programação seja integrado com teste de software. Desta forma, os objetivos deste trabalho podem ser caracterizados seguindo duas linhas de atuação:

- Desenvolver um módulo educacional integrado de programação OO e teste de software. Para o desenvolvimento desse material é utilizado o processo padrão para desenvolvimento de módulos educacionais estabelecido por Barbosa (2004), dando-se ênfase à evolução e aplicação da abordagem *AIM-CID*.
- Construir um ambiente de apoio, baseado na *Web* e na concepção de software livre, para a submissão e avaliação automática de trabalhos práticos dos alunos baseados em atividades de teste de software, auxiliando os processos de ensino e aprendizagem. Um ambiente automático de submissão e avaliação de trabalhos pode reduzir a sobrecarga de trabalho existente sobre os professores, permitir a avaliação de todos os alunos no mesmo nível e ainda possibilitar que o aluno tenha conhecimento sobre seu desempenho por meio dos relatórios gerados.

1.3 Organização do Trabalho

Essa dissertação está dividida em cinco capítulos. Neste capítulo foram apresentados o contexto em que se insere o trabalho, as motivações para o seu desenvolvimento e os seus objetivos. No Capítulo 2 é apresentada uma revisão bibliográfica das principais pesquisas relacionadas à condução deste trabalho — OO e seu ensino em fundamentos de programação, teste de software e como ele vem sendo ensinado, ambientes que dão suporte ao ensino de teste, e módulos educacionais e mecanismos de apoio ao seu desenvolvimento.

No Capítulo 3 mostra-se a evolução da abordagem *AIM-CID* proposta neste trabalho para que o módulo educacional integrado de programação OO e teste de software fosse construído. Este capítulo também discute como esse módulo integrado foi desenvolvido. Para a construção deste módulo foram utilizados os mecanismos para desenvolvimento de módulos educacionais propostos por Barbosa (2004), com ênfase na atividade de modelagem de conteúdos utilizando a abordagem *AIM-CID* e sua evolução. No Capítulo 4 é apresentado o ambiente *PROGTEST*, 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 de programação e teste de software. Aspectos de sua modelagem, arquitetura, implementação e operacionais são discutidos nesse capítulo. O Capítulo 5 apresenta as conclusões deste trabalho, sendo sintetizadas suas principais contribuições e trabalhos futuros. No Apêndice A são listados três *scripts* desenvolvidos para executar a ferramenta de teste *JaBUTi* por linha de comando permitindo sua integração com o ambiente *PROGTEST*.

Revisão Bibliográfica

2.1 Considerações Iniciais

Conforme mencionado anteriormente, a proposta deste trabalho é fornecer subsídios para que o ensino de teste de software seja integrado ao de fundamentos de programação. Para isso, foram desenvolvidos um módulo educacional integrado de fundamentos programação e teste de software e um ambiente para submissão e avaliação automática de trabalhos práticos, auxiliando o processo de ensino e aprendizagem. Com base nas tendências atuais do ensino de programação escolheu-se a Orientação a Objetos (OO) como o paradigma no qual foi desenvolvido o módulo educacional. Dentro dessa perspectiva, neste capítulo é apresentada uma revisão bibliográfica das principais pesquisas relacionadas à condução deste trabalho — OO e seu ensino em fundamentos de programação, teste de software e como ele vem sendo ensinado, ambientes que dão suporte ao ensino de teste e módulos educacionais e mecanismos de apoio ao seu desenvolvimento.

Na Seção 2.2 são apresentados a conceituação do paradigma orientado a objetos, os problemas e as principais tendências no ensino de programação. Na Seção 2.3 é apresentada uma visão geral da atividade de teste de software. Além disso, são relatados os principais problemas do ensino de teste de software e com ele tem sido conduzido. Na Seção 2.4 é mostrado um levantamento de ambientes educacionais de uma forma geral e também de ambientes que fornecem suporte ao ensino de teste de software, entre eles ambientes de programação pedagógicos e ambientes de submissão de trabalhos práticos. Por fim, é relatada na Seção 2.5 uma visão geral a respeito da modelagem de conteúdos educacionais.

2.2 Orientação a Objeto

A Orientação a Objeto começou a se consolidar a partir de meados da década de 80. A programação Orientada a Objeto (POO) iniciou-se com uma linguagem de programação que apoiou seus conceitos, a linguagem Smalltalk, criada em 1983.

A POO proporciona a abstração de dados, na qual dados e procedimentos passam a fazer parte de um só elemento básico com significado próprio, o objeto. Assim, a POO pode ser um processo mais natural que a programação estruturada pois a programação ocorre de uma maneira que reflete o mundo real. Desta forma, a POO tende a produzir software que é mais compreensível, organizado e mais fácil de manter, modificar e depurar (Sebesta, 2000).

A POO ¹ introduz uma abordagem na qual o programador visualiza seu programa em execução como uma coleção de objetos cooperantes que se comunicam por meio de troca de mensagens. Cada um dos objetos é uma instância de uma classe e todas as classes formam uma hierarquia de classes unidas via relacionamento de herança.

Uma **classe** é uma entidade estática que engloba **atributos** (ou dados) e **métodos** (ou funções membro) que representam operações que podem ser realizadas sobre os dados. Um **objeto** é uma instância de uma classe criado em tempo de execução. Cada objeto tem uma cópia dos dados existentes na classe e encapsula **estado** e **comportamento**.

Ao contrário da idéia procedimental de entrada e saída para procedimentos, a programação OO se preocupa com os objetos e seus relacionamentos. Os objetos interagem entre si e são ativados por meio de mensagens. Uma **mensagem** é uma solicitação para que um objeto execute um de seus métodos. O método solicitado pode alterar o estado interno do objeto e/ou enviar mensagens a outros objetos. Ao encerrar sua execução ele retorna o controle, e possivelmente algum valor, ao objeto que enviou a mensagem solicitando a operação.

A capacidade que um objeto tem de impedir que outros objetos tenham acesso aos seus dados é denominada de **encapsulamento**. O encapsulamento é uma técnica empregada para garantir o ocultamento de informações na qual a interface e a implementação de uma classe são separadas sintaticamente. Com isso, somente os métodos pertencentes a um objeto podem ter acesso aos dados encapsulados.

Novas classes podem ser definidas em função de classes já existentes. Tal relacionamento entre classes é obtido por meio de **herança**. Com a utilização de herança, as classes são inseridas em uma hierarquia de especialização, de modo que uma classe mais especializada herda todas as propriedades da(s) classe(s) mais genérica(s) (classe(s) que está(ão) um nível acima na hierarquia). À classe mais genérica dá-se o nome de **superclasse**, classe pai ou classe-base e à classe mais especializada dá-se o nome de **subclasse**, classe filha ou classe derivada. Com o mecanismo de herança, uma subclasse pode estender ou restringir as características herdadas da superclasse.

¹Os conceitos de POO descritos nessa seção foram parcialmente extraídos de (Vincenzi, 2004)

O termo **polimorfismo** representa a qualidade ou estado de um objeto ser capaz de assumir diferentes formas. Quando aplicado a linguagens de programação indica que uma mesma construção de linguagem pode assumir diferentes tipos ou manipular objetos de diferentes tipos. Por exemplo, o operador “+” nas linguagens C++ e Java pode ser utilizado para fazer a adição de dois valores inteiros ou ponto-flutuante, bem como para concatenar duas *Strings*.

Uma característica fortemente relacionada com herança e polimorfismo é o **acoplamento dinâmico** que permite que novos métodos sejam carregados e ligados à aplicação em tempo de execução.

Atualmente, com o aumento da importância e da utilização da POO, algumas iniciativas têm ocorrido no sentido de investigar sua introdução o mais cedo possível na grade curricular dos cursos de Ciências da Computação. A seguir é mostrada uma breve descrição dessas iniciativas.

2.2.1 Ensino de Programação OO

A abordagem mais tradicional no ensino de disciplinas introdutórias dos cursos de computação é fornecer uma visão geral sobre Ciências 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 (Hickey, 2004).

Segundo Alphonse e Ventura (2002), o ensino de introdução à programação possui alguns problemas:

- Os cursos de programação frequentemente 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 habilidades na construção dos algoritmos propriamente ditos. 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 frequentemente 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 Ciências da Computação é tentar fazer com que o mesmo seja interessante e relevante para os estudantes (Hickey, 2004). Nessa perspectiva, uma abordagem investigada é a introdução do paradigma OO em disciplinas introdutórias. Nesse sentido, vários trabalhos vêm sendo desenvolvidos com relação ao ensino de programação orientada a objetos (Adams e Frens, 2003; Alphonse e Ventura, 2002; Campbell e Bolker, 2002; Cooper et al., 2003; Daly e Horgan, 2004; Irimia, 2001;

Knudsen e Madsen, 1996; Kolling e Rosenberg, 2001; Neubauer e Strong, 2002; Rajaravivarma e Pevac, 2003; Richards, 2003; Roberts e Picard, 1998; Satratzemi et al., 2003).

Em recentes levantamentos nos EUA sobre a primeira linguagem de programação adotada nos cursos de graduação em computação, percebe-se a predominância das linguagens OO como C++ e Java. A *Reid First Course Language List* (Reid, 2004), contém uma pesquisa realizada em 2004 em mais de 200 universidades americanas e mostra que aproximadamente 75 % dessas universidades utilizam uma linguagem OO como primeira linguagem — C++ e Java são empregadas aproximadamente em 31 % e 44 % dos casos, respectivamente.

Pesquisas também mostram que os estudantes que aprendem primeiro a resolver problemas em um caminho procedural têm a tendência de adotar a programação OO como uma extensão da abordagem procedimental, apresentando dificuldade na assimilação dos seus conceitos. Na orientação a objetos, os alunos têm que compreender como um objeto é criado e como as mensagens entre eles são passadas ao invés da simples chamada de função da programação tradicional (Campbell e Bolker, 2002; Kamin et al., 2002; Rajaravivarma e Pevac, 2003; Savitch, 2001).

Adams e Frens (2003) acreditam 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 menores custos de manutenção sobre o seu tempo de vida. Para Alphonse e Ventura (2002), a orientação a objetos pode ser explorada nas disciplinas introdutórias de modo que os alunos obtenham 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 OO esteja se tornando cada vez mais comum, os educadores, em geral, 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 (Kolling e Rosenberg, 2001).

Segundo Cooper et al. (2003), 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 controle tradicionais são ensinadas, mas sempre dentro do contexto de OO.

Apesar das vantagens relatadas com a introdução da POO como primeira linguagem de programação, os problemas relacionados à visão errada que os alunos possuem sobre a atividade de programação continuam existindo, pois essas visões errôneas independem do paradigma que é ensinado (Edwards, 2004). A seguir são descritas essas visões erradas dos alunos:

- 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;

Tais visões muitas vezes são decorrentes da forma pela qual a programação é ensinada, em geral, com base 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 com o desenvolvimento e com o entendimento dos seus algoritmos, já que os mesmos terão que ser testados. Logo, a atividade de teste ensinada conjuntamente com fundamentos de programação pode ajudar a acabar com essas visões erradas que os estudantes possuem sobre a atividade de programação, além de contribuir para a consolidação de uma cultura de qualidade de software.

2.3 Teste de Software

Conforme mencionado anteriormente, um dos objetivos deste trabalho é o desenvolvimento de material didático de fundamentos básicos de programação OO e teste de software. Nessa perspectiva, é relatada nesta seção uma visão geral sobre a atividade de teste de software e como essa atividade tem sido ensinada atualmente nos cursos de graduação.

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 no produto e aumentar a confiança de que o mesmo esteja correto, representando a última revisão da especificação, projeto e codificação (Harrold, 2000; Maldonado, 1991; Pressman, 2005; Zhu, 1996).

Segundo Myers et al. (2004), o principal objetivo do teste de software é revelar a presença de erros ou defeitos no produto. Portanto, o teste bem sucedido é aquele que consegue determinar casos de teste para os quais o programa que está sendo testado falhe.

O teste de produtos de software envolve basicamente quatro etapas (Maldonado, 1991; Pressman, 2005): 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 primeira 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 (Pressman, 2005).

Um dos pontos críticos da atividade de teste é o projeto e/ou avaliação dos casos de teste a serem utilizados. O programa, em princípio, deveria ser exercitado com todos os valores do domínio de entrada. Sabe-se, entretanto, que o teste exaustivo é impraticável por razões de custo e tempo. Dessa forma, por questões de produtividade, o objetivo é utilizarem-se casos de teste que tenham alta probabilidade de revelar a presença da maioria dos erros existentes com o mínimo de tempo e esforço. Assim, 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 (Demillo, 1980).

Em geral, os critérios de teste são estabelecidos a partir de três técnicas: funcional, estrutural e baseada em erros. Além dessas, do ponto de vista comportamental, existe também a técnica de teste baseada em máquinas de estados finitos (MEF). 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; e na **técnica baseada em máquinas de estados finitos** (MEF) onde uma máquina de estados finitos é utilizada para modelar o comportamento do sistema (Maldonado, 1991).

É importante destacar que os critérios de teste podem ser utilizados para auxiliar tanto na geração de conjuntos de casos de teste quanto na avaliação da adequação desses conjuntos. Além disso, ressalta-se que as técnicas de teste devem ser vistas como complementares e a questão que se coloca está em como utilizá-las de forma que as vantagens de cada uma sejam melhor exploradas em uma estratégia que leve a um teste de boa qualidade, ou seja, eficaz em revelar a presença de erros e de baixo custo (Maldonado, 1991).

Outro aspecto fundamental associado ao teste de software refere-se ao desenvolvimento de ferramentas que automatizem a aplicação das técnicas e critérios de teste.

Conceitos e mecanismos de apoio ao teste de software também vêm sendo investigados em outros contextos, como a orientação a objetos, introduzindo com isso novos desafios e considerações no teste de produtos. As problemáticas do teste de programas orientados a objetos, bem como suas técnicas, critérios e ferramentas são relatadas nas seções seguintes.

2.3.1 Problemáticas do Teste de Software OO

O desenvolvimento orientado a objetos fornece uma forma abstrata de resolução de problemas que não está presente no desenvolvimento tradicional. A orientação a objetos também pode levar a uma melhor arquitetura de um sistema que está sendo desenvolvido e forçar uma disciplina no estilo de programação. No entanto, não é possível evitar os erros cometidos pelos programadores, pela falta de entendimento dos requisitos e, conseqüentemente, garantir a produção correta de programas. Desta forma, programas orientados a objetos também devem ser testados (Barbey e Strohmeier, 1994; Binder, 1999; Perry e Kaiser, 1990).

As características típicas do paradigma de programação orientada a objetos, como exemplo, encapsulamento, herança e polimorfismo, embora tragam benefícios para o projeto e codificação, também acarretam problemas quanto à maneira com que os requisitos e os critérios de teste devem ser estabelecidos e aplicados. Os principais problemas que essas características podem causar na atividade de teste são relatados a seguir (Barbey e Strohmeier, 1994; Binder, 1999; Perry e Kaiser, 1990; Vincenzi, 2004):

- **Encapsulamento:** O encapsulamento é um mecanismo de controle de acesso que determina a visibilidade de atributos e métodos dentro de uma classe. O teste requer um relatório completo do estado concreto e abstrato de um objeto, bem como a possibilidade de alterar esse estado facilmente. O encapsulamento torna o teste mais difícil porque existe uma falta de visibilidade dos estados — encapsula o comportamento e os estados de um objeto — e também existe uma dificuldade na inicialização dos itens de dados e na chamada de métodos. Apesar de o encapsulamento não contribuir diretamente na introdução de falhas, ele pode ser um obstáculo na atividade de teste por reduzir a capacidade de controle e de observação.
- **Herança:** A herança é o relacionamento entre classes que permite a definição de uma nova classe (derivada) baseada na definição de uma classe existente (base). As classes derivadas podem ser redefinidas pela modificação ou remoção de operações herdadas, ou pela adição de novas propriedades. Também permite a reusabilidade por meio do compartilhamento de características presentes em uma classe já definida anteriormente. A herança pode contribuir com a introdução dos seguintes erros: enfraquecimento do encapsulamento, criação de um risco de erro similar ao uso de variáveis globais em programas procedimentais, inconsistências e comportamentos incorretos. Além disso, grandes hierarquias de herança podem dificultar a compreensão pelo aumento da chance de ocorrer erros e pela redução da testabilidade das classes. A utilização de herança pode levar a uma falsa conclusão de que subclasses que herdaram características de superclasses não precisam ser testadas, reduzindo assim o esforço com os testes. No entanto, a herança define um novo contexto para os métodos, de maneira que um método que foi testado em uma super-classe necessita ser retestado quando é reusado em uma sub-classe. Mesmo

se o método for herdado de uma super-classe sem qualquer modificação, ele deve ser retestado no contexto da sub-classe.

- **Polimorfismo:** O polimorfismo é a capacidade de se poder fazer referência a mais de um tipo de objeto usando o mesmo nome ou variável. Permite que, em tempo de execução, associações entre objetos diferentes sejam realizadas e facilita a produção de código elegante e extensível. Entretanto, o polimorfismo pode trazer indecidibilidade ao programa que está sendo testado, pois é impossível, quando é invocada uma operação com nome polimórfico, prever em tempo de execução se é a implementação original ou a redefinida que está sendo testada. O polimorfismo pode conduzir também a conversões erradas de tipos, levando a uma dificuldade na detecção de erros.

Existem ainda outros tipos de problemas que impactam a realização do teste de programas orientados a objetos, como garantir o comportamento correto na execução de objetos concorrentes, e a executar mensagens em alguma sequência.

2.3.2 Fases do Teste de Software OO

A atividade de teste é dividida em fases, tanto na abordagem de desenvolvimento de software procedimental quanto na orientada a objetos, sendo desenvolvida de forma incremental. Conforme comentado anteriormente, existem basicamente três fases de teste: teste de unidade, teste de integração, teste de sistema (Maldonado e Fabbri, 2001).

O **teste de unidade** tem por objetivo explorar a menor unidade do projeto, procurando identificar erros de lógica e de implementação em cada módulo separadamente. Segundo o padrão IEEE 610.12-1990, uma unidade é um componente de software que não pode ser subdividido, podendo ser uma sub-rotina ou um procedimento correspondendo à menor unidade funcional de um programa (Maldonado e Fabbri, 2001).

Para a realização de testes de unidade é necessária a implementação de *drivers* e *stubs*. O *driver* é responsável por pegar dados de teste fornecidos pelo testador, passar esses dados na forma de parâmetros para a unidade e coletar os resultados relevantes produzidos e apresentá-los para o testador. Um *stub* é uma unidade que substitui, na hora do teste, uma unidade que está subordinada ao módulo a ser testado e que ainda não foram implementados e/ou testados (Vincenzi, 2004).

O **teste de integração** tem por objetivo descobrir erros associados às interfaces entre os módulos quando esses são integrados para construir a estrutura do software que foi estabelecida na fase de projeto. E o **teste de sistema** tem como principal objetivo identificar erros de funções e características de desempenho que não estejam de acordo com a especificação (Pressman, 2001).

No contexto de software orientado a objetos algumas variações são identificadas nessas fases descritas acima. No caso do teste de unidade, a menor unidade a ser testada é um método, sendo

que a classe da qual o método pertence pode ser vista como o *driver* do método. Assim, o teste de unidade para software orientado a objetos é conhecido como teste **intra-método** (Vincenzi, 2004).

Considerando uma única classe é possível pensar em teste de integração. No que diz respeito ao teste de integração, têm-se três tipos de teste para programas OO: teste **inter-método**, teste **intra-classe** e teste **inter-classe**. No teste **inter-método** deve ser testada a integração entre os métodos de uma mesma classe que interagem entre si. No teste **intra-classe** são testadas interações entre métodos públicos fazendo chamada a esses métodos em diferentes seqüências. O objetivo é identificarem-se possíveis seqüências de ativação de métodos inválidos que levem o objeto a um estado inconsistente. No teste **inter-classe** o mesmo conceito de invocação de métodos públicos em diferentes seqüências é utilizado, entretanto, esses métodos públicos não necessitam estar na mesma classe (Vincenzi, 2004).

O teste de sistema é conduzido após os testes citados acima serem realizados e o sistema ser integrado como um todo. Esse teste é baseado em critérios funcionais, e por isso não apresenta diferenças fundamentais entre o teste procedimental e OO.

Segundo Vincenzi (2004), alguns autores consideram a classe como a menor unidade no paradigma OO. Nessa direção, o teste de unidade poderia envolver o teste intra-método, inter-método e intra-classe e, o teste de integração corresponderia ao teste inter-classe. Nas Tabelas 2.1 e 2.2, adaptadas de Vincenzi (2004) é mostrada uma relação entre a menor unidade a ser considerada no teste, método ou classe respectivamente, e os tipos de teste de software orientados a objetos que podem ser aplicados a cada uma das fases de teste.

A Figura 2.1 ilustra as três fases de teste mencionadas acima bem como os componentes utilizados em cada uma das fases tanto para programa procedimentais como para programas OO.

Tabela 2.1: Relação entre as fases de teste e o teste de software OO considerando o método como a menor unidade (Vincenzi, 2004)

FASES	TESTE DE SOFTWARE ORIENTADO A OBJETOS
<i>Unidade</i>	<i>Intra-método</i>
<i>Integração</i>	<i>Inter-método, Intra-classe e Inter-classe</i>
<i>Sistema</i>	<i>Toda a aplicação</i>

2.3.3 Técnicas e Critérios de Teste de Software OO

As técnicas e critérios de teste fornecem uma abordagem sistemática e teoricamente fundamentada para se conduzir e avaliar a qualidade do teste de software. Conforme mencionado anteriormente, essas técnicas diferenciam-se pela origem da informação utilizada na avaliação e construção dos conjuntos de casos de teste. Dentre elas, encontram-se as técnicas de teste

Figura 2.1: Relacionamento entre teste de unidade, de integração e de sistema: programas procedimentais e OO (Vincenzi, 2004)

Tabela 2.2: - Relação entre as fases de teste e o teste de software OO considerando a classe como a menor unidade (Vincenzi, 2004)

FASES	TESTE DE SOFTWARE ORIENTADO A OBJETOS
<i>Unidade</i>	<i>Intra-método, Inter-método e Intra-classe</i>
<i>Integração</i>	<i>Inter-classe</i>
<i>Sistema</i>	<i>Toda a aplicação</i>

funcional, estrutural, baseada em erros e baseada em máquinas de estados finitos (Maldonado, 1991).

Cada uma das técnicas de teste citadas acima possui uma série de critérios de teste. Um critério de teste serve para selecionar e avaliar casos de teste com o intuito de revelar a presença de defeitos, ou estabelecer um nível elevado de confiança na correção do produto quando erros não são revelados (Maldonado e Fabbri, 2001).

Apesar da maioria dos critérios existentes serem para o teste de programas procedimentais, já existem iniciativas de estender esses critérios para o teste de software orientado a objetos. Nas sub-seções seguintes são apresentadas as técnicas funcional, estrutural, e baseada em erros,

bem como seus respectivos critérios, que são explicados dentro do contexto de orientação a objetos

2.3.3.1 Técnica funcional

A técnica funcional (teste baseado em especificação) aborda o software do ponto de vista macroscópico sem se preocupar com detalhes de implementação utilizando a especificação de requisitos do programa para derivar os casos de teste. Assim sendo, 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. O teste funcional também é conhecido como caixa preta pelo fato de tratar o software como uma caixa da qual o conteúdo é desconhecido e só é possível visualizar o lado externo (Maldonado e Fabbri, 2001; Myers et al., 2004; Pressman, 2005).

A realização do teste funcional envolve basicamente duas etapas: a identificação das funções que o programa deve realizar e a criação de casos de teste adequados para verificar se as mesmas estão funcionando de forma correta (Pressman, 2005).

Alguns problemas são encontrados com a utilização da técnica funcional (Vincenzi, 2004):

- Dificuldade em quantificar a atividade de teste, pois não se pode garantir que as partes essenciais ou críticas do programa serão executadas.
- O teste está sujeito às inconsistências decorrentes da especificação. Como geralmente, a especificação é feita de forma descritiva e informal, os requisitos derivados da especificação também são, de certa forma, descritivos e informais, dificultando a automatização dos critérios funcionais.

Como a técnica funcional precisa somente da especificação de requisitos, não necessitando da implementação para ser aplicada, todos os critérios funcionais podem ser aplicados a programa orientados a objetos (Vincenzi, 2004). Os critérios particionamento em classes de equivalência, análise do valor limite e grafo causa-efeito são usados também em programas procedimentais (Pressman, 2005), mas o critério método partição-categoria foi desenvolvido para software OO (Offutt e Irvine, 1995). Exemplos de critérios de teste funcional são mostrados a seguir.

- **Particionamento em Classes de Equivalência:** O particionamento em classes de equivalência divide o domínio de entrada de um programa em classes de equivalência válidas e inválidas que provavelmente exercitarão uma função de software específica, levando em consideração as condições de entrada dos dados identificados na especificação. A partir daí, os casos de teste são derivados, baseando-se na idéia que se um caso de teste de determinada classe revela um erro, todos os demais elementos daquela mesma classe

também revelam o mesmo erro (Barbosa et al., 2000; Myers et al., 2004; Pressman, 2005; Vincenzi, 2004). O uso do particionamento em classes de equivalência permite avaliar os requisitos mais sistematicamente. Permite também a minimização do número de casos de teste, uma vez que se seleciona apenas um caso de teste para cada classe, pois se leva em consideração que todos os elementos de uma classe devem se comportar de maneira equivalente (Barbosa et al., 2000; Maldonado e Fabbri, 2001).

- **Análise do Valor Limite:** A análise do valor limite é um complemento ao critério particionamento em classes de equivalência e verifica a capacidade de um programa em manipular dados nos limites das condições de entrada. Este critério se difere do particionamento em classes de equivalência porque ao invés de se selecionar qualquer elemento de uma determinada classe de equivalência, os casos de teste são construídos com os valores limite das classes. Os valores limite são utilizados, pois são nestes pontos que se concentram o maior número de erros. Além disso, ocorre também o particionamento da saída do programa em classes e casos de teste que produzam resultados nos limites dessas classes de saída devem ser criados (Barbosa et al., 2000; Myers et al., 2004; Vincenzi, 2004).
- **Grafo Causa-Efeito:** Como os critérios mostrados acima não exploram as combinações das condições de entrada, o grafo causa-efeito estabelece requisitos de teste que são baseados nas possíveis combinações das condições de entrada. Para a aplicação desse critério, devem primeiramente ser levantados as causas (condições de entrada distintas) e os efeitos (possíveis condições de saída) de um programa. Com essas informações, é construído em seguida um grafo relacionando as causas e os efeitos identificados. Posteriormente, esse grafo é convertido em uma tabela de decisão e, a partir dela, os casos de teste são derivados (Barbosa et al., 2000; Myers et al., 2004; Vincenzi, 2004).
- **Método de Partição-Categoria:** Offut (Offutt e Irvine, 1995) propôs um critério de teste funcional no teste de programas OO chamado de Método de Partição-Categoria. Esse método oferece um procedimento que é utilizado pelo testador para produzir casos de teste a partir da especificação. O trabalho do testador é definir categorias que representem as principais características do domínio de entrada da função que está sendo testada e particionar cada categoria em classes de equivalência de entradas chamadas *choices*. Por definição, as *choices* dentro de uma categoria devem ser disjuntas e, quando unidas, devem cobrir todo o domínio de entrada de uma determinada categoria.

2.3.3.2 Técnica estrutural

A técnica estrutural, também conhecida como caixa-branca, permite examinar a estrutura interna do programa, verificando os detalhes do código e solicitando a execução de partes ou de componentes elementares do programa (Myers et al., 2004). O teste estrutural enquadra-se na

classificação de teste baseado em programa, pois se baseia na estrutura interna da implementação para a aplicação dos casos de teste.

Em geral, a maioria dos critérios dessa técnica utilizam uma representação de programa conhecido como grafo de fluxo de controle ou de grafo de programa. Um grafo de fluxo de controle é um grafo orientado que possui um único nó de entrada e um único nó de saída, onde cada aresta representa um possível desvio de um bloco ao outro. A partir desse grafo podem ser escolhidos os componentes que devem ser executados, caracterizando assim o teste estrutural (Barbosa et al., 2000).

Os critérios da técnica estrutural geralmente são classificados em critérios baseados em fluxo de controle, baseados em fluxo de dados e baseados em complexidade. Os critérios baseados em fluxo de controle usam apenas características baseadas no controle de execução do programa, como comandos ou desvios, para determinar quais estruturas são necessárias. Os mais conhecidos são os critérios (Pressman, 2005):

- **Todos-Nós:** Exige que durante a execução, cada comando do programa seja executado pelo menos uma vez, ou seja, deve passar por cada vértice do grafo de fluxo de controle ao menos uma vez.
- **Todos-Arcos:** Requer que cada desvio de fluxo de controle do programa, ou seja, cada aresta do grafo seja executada pelo menos uma vez
- **Todos-Caminhos:** Geralmente é impraticável e exige que todos os caminhos possíveis do programa sejam executados.

Os critérios baseados em fluxo de dados associam ao grafo de fluxo de controle informações sobre o fluxo de dados do programa a fim de determinar os requisitos de teste. Os critérios baseados em fluxo de dados exploram as associações entre pontos do programa onde ocorre a definição de uma variável e pontos onde ocorre uma referência ou uso dessa variável. Exemplos desses critérios são os critérios de Rapps e Weyuker (1982, 1985) e os Critérios Potenciais-Usos (Maldonado, 1991).

De acordo com os critérios de Rapps e Weyuker (1985), para derivar os requisitos de teste exigidos é necessário adicionar ao grafo de programa informações sobre fluxo de dados, caracterizando o Grafo Def-Uso (Def-Use Graph) definido por Rapps e Weyuker (1982, 1985). Nesse grafo são exploradas as associações entre a definição e o uso das variáveis determinando os caminhos a serem exercitados. Quando a variável é usada em uma computação, diz-se que seu uso é computacional (c-uso); quando usada em uma condição, seu uso é predicativo (p-uso). Os critérios básicos que fazem parte da família de critérios definidos por Rapps e Weyuker (1985) são:

- **Todas-definições:** Requer que cada definição de variável seja exercitada pelo menos uma vez.

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

Os critérios Potenciais-Usos, por sua vez, requerem associações independentemente da ocorrência explícita de uma referência (um uso) a uma definição de variável, ou seja, requerem que caminhos livres de definição a partir da definição de uma determinada variável sejam executados, independentemente de ocorrer um uso dessa variável nesse caminho (Maldonado, 1991).

Os critérios de teste estrutural apresentados acima são utilizados no teste de programas procedimentais. Esses critérios também são utilizados no teste intra-método de software orientado a objetos. Outros trabalhos vêm sendo realizados para complementar os critérios estruturais de programas OO. Uma síntese desses trabalhos é mostrada a seguir ².

Harrold et al. (1994) estenderam o teste de fluxo de dados para o teste de classes. Os autores comentam que os critérios de fluxo de dados destinados ao teste de programas procedimentais podem ser utilizados tanto para o teste de métodos individuais quanto para o teste de métodos que interagem entre si dentro de uma mesma classe. Entretanto, esses critérios não consideram interações de fluxo de dados quando os usuários de uma classe invocam uma seqüência de métodos em uma ordem arbitrária.

Para resolver esse problema, os autores apresentam uma abordagem que permite testar diferentes tipos de interações de fluxo de dados entre classes. A abordagem proposta usa as técnicas tradicionais de fluxo de dados para testar os métodos individuais e as interações entre os métodos dentro de uma mesma classe. Para testar os métodos que são acessíveis fora da classe e que podem ser utilizados por outras classes, uma nova representação, denominada grafo de fluxo de controle de classe, foi desenvolvida. Para gerar-se o grafo de fluxo de controle de classe (CCFG), primeiramente constrói-se o grafo de chamada de classe que é um grafo dirigido no qual os nós representam os métodos e as arestas representam mensagens entre os métodos. A partir desse grafo, novos requisitos de teste inter-método, intra-classe e inter-classe podem ser obtidos (Harrold et al., 1994).

Buy et al. (2000) desenvolveram uma técnica para automatizar o teste de classes, baseada na identificação de seqüências de invocação de métodos que são utilizadas para exercitar a classe que está sendo testada. Essa técnica usa na análise de fluxo de dados, o grafo de fluxo de controle de classes para calcular os pares def-uso de cada instância das variáveis de uma classe.

Chen e Kao (1999) desenvolveram uma estratégia de teste baseada no fluxo de objetos para ajudar na geração de casos de teste. Para isso, dois critérios foram propostos a fim de ajudar no rastreamento do comportamento dinâmico de programas orientados a objetos. Esses critérios

²A síntese desses trabalhos foi parcialmente extraída de Vincenzi (2004)

são *all-bindings* e *all-du-pairs*. O critério *all-binding* é usado para garantir a cobertura baseada no exercício de todos os possíveis tipos de objetos que existem devido à herança e ao polimorfismo. O critério *all-du-pairs* é utilizado para monitorar o comportamento dos objetos através do rastreamento de onde um objeto é definido e usado.

Uma extensão do teste de fluxo de dados para classes de Harrold et al. (1994) foi investigada por Souter et al. (1999). O foco desse trabalho foi como computar pares def-uso inter-classes em classes que interagem com muitos métodos e que possuem interações complexas.

Souter e Pollock (2003) estabeleceram o conceito de associações definição-uso contextuais (contextual def-use associations). Segundo este conceito, deve existir um contexto específico para que determinada associação definição-uso seja considerada coberta. Com isso, as associações de fluxo de dados obtidas a partir dos critérios descritos acima seriam classificadas com sendo livres de contexto (context-free) uma vez que as mesmas podem ser cobertas sem considerar um contexto específico. Segundo as autoras, o teste baseado em associações definição-uso contextuais pode melhorar a cobertura dos testes uma vez que múltiplas associações definição-uso contextuais únicas podem ser geradas para uma mesma associação definição-uso livre de contexto

Vincenzi (Vincenzi, 2004) definiu oito critérios de teste destinados ao teste intra-método de programas OO. Quatro desses critérios são baseados em análise de fluxo de controle e quatro baseados em análise de fluxo de dados. Os requisitos de teste são separados em dois conjuntos disjuntos: (1) os que podem se cobertos durante a execução normal do programa, denominados independentes de exceção: Todos-Nós-Independentes-de-Exceção (Todos-Nós-ei), Todos-Arcos-Independentes-de-Exceção (Todos-Arcos-ei), Todos-Usos-Independentes-de-Exceção (Todos-Usos-ei) e Todos-Potenciais-Usos-Independentes-de-Exceção (Todos-Pot-Usos-ei); e (2) os que para serem cobertos exigem, obrigatoriamente, que uma exceção tenha sido gerada, denominados dependentes de exceção: Todos-Nós-Dependentes-de-Exceção (Todos-Nós-ed), Todos-Arcos-Dependentes-de-Exceção (Todos-Arcos-ed), Todos-Usos-Dependentes-de-Exceção (Todos-Usos-ed) e Todos-Potenciais-Usos-Dependentes-de-Exceção (Todos-Pot-Usos-ed).

2.3.3.3 Técnica baseada em erros

A técnica de teste baseada em erros utiliza informações sobre os tipos de erros mais frequentes no processo de desenvolvimento de software para derivar os requisitos de teste. A ênfase da técnica está nos erros que o programador ou projetista pode cometer durante o desenvolvimento e nas abordagens que podem ser usadas para detectar a sua ocorrência (Barbosa et al., 2000). Basicamente existem dois tipos de critérios baseados em erros: semeadura de erros e análise de mutantes.

No critério semeadura de erros alguns erros são inseridos artificialmente no programa. Depois da realização do teste, verificam-se entre os erros encontrados quais são naturais e quais

são artificiais. Após isso, usando estimativas de probabilidade, o número de erros naturais ainda existentes no programa pode ser calculado (Budd, 1981).

A idéia básica da análise de mutantes baseia-se na hipótese do programador competente, que assume que programadores experientes escrevem programas corretos ou muito próximos do correto (Demillo et al., 1978). Considerando a validade desta hipótese, pode-se afirmar que erros são introduzidos no programa por meio de pequenos desvios sintáticos que, embora não causem erros sintáticos, alteram sua semântica e, conseqüentemente, o programa pode apresentar um comportamento incorreto. Para revelar a presença de tais erros, a análise de mutantes identifica os desvios sintáticos mais comuns e, por meio da aplicação de pequenas transformações sobre o programa em teste, encoraja o testador a construir casos de testes que mostrem que tais transformações levam a um programa incorreto (Agrawal et al., 1989).

Outra hipótese explorada na aplicação do critério análise de mutantes é o efeito de acoplamento (Demillo et al., 1978), que assume que erros complexos estão relacionados a erros simples. Assim, espera-se, e alguns estudos empíricos já confirmaram essa hipótese (Acree et al., 1979; Budd et al., 1980), que conjuntos de casos de teste capazes de revelar a presença de erros simples são também capazes de revelar a presença de erros complexos.

Partindo-se da hipótese do programador competente e do efeito de acoplamento, a análise de mutantes utiliza um conjunto de programas que são ligeiramente modificados (mutantes) através de operadores de mutação, a partir de um determinado programa P para avaliar o quanto um conjunto de casos de teste T é adequado para o teste de P. O objetivo é determinar um conjunto de casos de teste que consiga revelar, através da execução de P, as diferenças de comportamento entre o programa P e seus mutantes. Um operador de mutação consiste nas regras que definem as alterações que devem ser aplicadas no programa original P.

O teste baseado em erros, no que diz respeito ao teste de programas OO, está sendo utilizado para testar aspectos referentes à concorrência, comunicação entre processos e teste de programas Java em nível de unidade e integração. Em 1999, Kim et al. (1999a,b) utilizaram uma técnica denominada Hazop (*Hazzard and Operability Studies*) para definir um conjunto de operadores de mutação para o teste de programas Java. Esse conjunto de operadores de mutação de classes não explora todas as características de um programa orientado a objetos.

Bieman et al. (2001) definiram um conjunto de operadores de mutação específico para o teste de determinadas classes da Java API. Offutt et al. (2002) propuseram um conjunto de operadores de mutação de classe para Java que estendem os operadores propostos por Kim et al. (1999a,b).

Um conjunto de operadores de mutação visando ao teste de interfaces de comunicação entre componentes distribuídos foram desenvolvidos por Ghosh e Mathur (2001). Delamaro et al. (2001b) definiram um conjunto de operadores de mutação que modelam erros típicos cometidos em programas concorrentes implementados em Java. Ainda, Vincenzi (2004) definiu três conjuntos distintos de operadores de mutação os quais podem ser utilizados no teste intra-método, inter-método e inter-classe, considerando as linguagens Java e C++.

2.3.4 Ferramentas de Teste

As ferramentas de teste automatizam 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 (Horgan e Mathur, 1992). Além de contribuir para a qualidade e produtividade da atividade de teste, a disponibilidade de ferramentas de teste permite a transferência tecnológica entre universidade e indústria, fator indispensável para a produção de software de alta qualidade. Além disso, a existência de ferramentas auxilia pesquisadores e alunos de Engenharia de Software a adquirirem os conceitos básicos e experiência na comparação, seleção e estabelecimento de estratégias de teste. Nesse sentido, tais ferramentas desempenham um papel importante no processo de ensino, aprendizado e treinamento dos conceitos de teste, contribuindo para a formação e capacitação de pessoal (Barbosa, 2004).

Desta forma, pretende-se ao se ensinar teste de software com fundamentos de programação fazer com que os alunos exercitem na prática os conceitos de teste aprendidos utilizando ferramentas de teste pertinentes. As ferramentas serão utilizadas antes da submissão dos trabalhos ao ambiente e servirão de apoio na construção dos casos de teste pelos alunos.

Nesse sentido, como este trabalho trata de teste de software para programas OO, foi realizada uma pesquisa sobre as ferramentas comerciais existentes de teste de programas OO, principalmente para programas escritos em C++ e Java que vêm sendo desenvolvidas. Domingues (2002) apresenta uma descrição dessas ferramentas de teste comerciais para programas OO, entre as quais destacam-se: *C++ Test*, *Jprobe Suite*, *Jtest*, *Panorama C/C++ e Panorama for Java*, *PureCoverage*, *TCAT C/C++ e TCAT Java*, *xSuds Toolsuite*. A seguir é apresentada uma descrição extraída de Domingues (2002), de algumas dessas ferramentas:

- **C++ Test:** Desenvolvida pela *Parasoft Corporation* é uma ferramenta de teste de unidade para códigos C/C++ que executa teste funcional, teste estrutural e teste de regressão. Essa ferramenta permite aos desenvolvedores testar suas classes imediatamente após elas terem sido escritas e compiladas através da automatização da criação de driver e de quaisquer *stubs* necessários, no qual o testador pode personalizar os seus respectivos valores de retorno, ou ainda entrar com seus próprios *stubs*.
- **Jprobe Suite:** Produzida pela *Gramatech*, é um conjunto de três ferramentas composto por: *JProbe Profiler and Memory Debugger*, que ajuda a eliminar gargalos de execução causados por algoritmos ineficientes em códigos Java e aponta as causas de perdas de memória nessas aplicações rastreando quais objetos seguram referências para outros; *JProbe Threadalyzer*, que monitora interações entre threads e avisa o testador quando essa interação representar perigo. Essa ferramenta identifica potenciais perigos de concorrências e deadlocks; *JProbe Coverage*, que localiza códigos não testados e mede quanto do có-

digo está sendo exercitado, permitindo ao testador estimar qual a confiança dos testes executados.

- **Jtest:** Também desenvolvida pela *Parasoft Corporation*, é uma ferramenta de teste de classes para códigos Java que realiza análise estática, teste funcional, teste estrutural e teste de regressão. Essa ferramenta pode executar todos esses tipos de teste em uma simples classe ou em um conjunto de classes.
- **Panorama C/C++ e Panorama for Java:** Desenvolvido pela *International Software Automation* é um conjunto de cinco ferramentas composto por: 1) OO-Test; 2) OO-SQA; 3) OO-Analyser; 4) OO-Browser; e 5) OO-Diagrammer. Essas ferramentas são úteis no teste de software, na garantia de qualidade e na reengenharia e ainda suportam as etapas do projeto de software, codificação e documentação. OO-Test executa a análise de cobertura de arcos, análise das frequências dos arcos executados, análise da eficiência dos casos de teste e minimização do conjunto de casos de teste.
- **PureCoverage:** Construída pela *Rational Software Corporation*, é uma ferramenta de análise de cobertura para códigos C++ e Java que aponta as áreas do código que foram ou não exercitadas durante os testes. Ela expõe o código não testado em todas as partes da aplicação. Essa ferramenta apresenta diversas opções de apresentação que auxiliam o testador na compreensão de quais partes do código foram ou não testadas através da cobertura de código em arquivos, módulos e linhas de código, permitindo ainda ao testador escolher o nível de cobertura de código por módulos e então focalizar as partes da aplicação que mais lhe interessam.
- **TCAT C/C++ e TCAT Java:** É uma ferramenta que fornece medidas para determinar o quanto de código que já foi testado, desenvolvida pela *Software Research*. Esta ferramenta gera relatórios gráficos e textuais que auxiliam na identificação de quais partes do programa foram exercitadas pelos casos de teste.
- **xSuds Toolsuite:** A *xSuds Toolsuite da Telcordia Technologies* é um conjunto de sete ferramentas para o entendimento, a análise e o teste de software para códigos C/C++ composto por: xATAC, que mede o quanto à aplicação está sendo exercitada pelo conjunto de casos de teste, identificando códigos que não estão suficientemente testados e determinando a cobertura da aplicação com os casos de teste; xRegress, que ajuda o testador a identificar um subconjunto representativo de casos de teste que devem ser novamente executados para revalidar as modificações nas aplicações; xVue, que localiza códigos do programa associado a determinadas características; xSlice, que automatiza e auxilia desenvolvedores na realização da atividade de depuração através da focalização de partes relevantes do código; xProf, que ajuda desenvolvedores a melhorarem a execução de seus códigos apontando somente as partes relevantes que necessitam ser analisadas

e possivelmente reorganizadas e/ou novamente escritas, para que a execução global da aplicação seja melhorada; xFind, que localiza regiões interessantes para a realização de testes na aplicação e exporta para as outras ferramentas; xDiff, que mostra diferença entre arquivos, tais como linhas modificadas, adicionadas ou removidas o que pode ser útil para comparar versões de código, documentos, dados e resultados.

Como uma das contribuições de grupo de Engenharia de Software do ICMC/USP no contexto de orientação a objetos, foi desenvolvida a ferramenta JaBUTi (Java Bytecode Understanding and Testing) (Vincenzi, 2004). A ferramenta JaBUTi é explicada detalhadamente na Seção 2.3.4.1, pois a mesma foi utilizada neste trabalho. Além da JaBUTi, outras ferramentas de teste, em diferentes contextos, foram ou estão sendo desenvolvidas pelo Grupo de Engenharia de Software do ICMC/USP. Dentre elas podem-se destacar:

- **Proteum (*Program Testing Using Mutants*)**: apóia a aplicação do critério Análise de Mutantes, em nível de unidade, no teste de programas C (Delamaro e Maldonado, 1996). Existe uma série de outras ferramentas baseadas na **Proteum** como a **Proteum/IM** que apóia a aplicação do critério Mutação de Interface (Delamaro et al., 2001a) no teste de programas C, em nível de integração. As outras extensões da **Proteum** são utilizadas para teste de especificação.
- **Poke-Tool (Potential Uses Criteria Tool for Program Testing)**: desenvolvida na FEEC/UNICAMP em colaboração com o ICMC/USP, apóia a aplicação dos critérios Potenciais-Usos e também outros critérios estruturais como Todos-Nós e Todos-Arcos (Chaim, 1991).

2.3.4.1 JaBUTi

A ferramenta JaBUTi (*Java Bytecode Understanding and Testing*), foi desenvolvida no ICMC/USP em cooperação com pesquisadores da UNIVEM e da *University of Texas at Dallas*, e é um ambiente completo para o teste de programas e componentes Java. 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 o programa/componente, e implementa ainda algumas heurísticas de particionamento de programas que visam a auxiliar a localização de falhas (Vincenzi, 2004).

Uma das principais vantagens associadas à JaBUTi está no fato da ferramenta não requerer código-fonte na condução da atividade de teste, pois todas as informações são coletadas do Java bytecode. Esta característica permite utilizar a ferramenta não só no teste de aplicações Java, mas também no teste de componentes baseados em Java, que, em geral são disponibilizadas sem código fonte (Vincenzi et al., 2003)

A ferramenta JaBUTi implementa quatro critérios de teste intra-método baseados em fluxo de controle (Todos-Nós-ei, Todos-Nós-ed, Todos-Arcos-ei, Todos-Arcos-ed) e quatro critérios

de teste intra-método baseados em fluxo de dados (Todos-Usos-ei, Todos-Usos-ed, Todos-Potenciais-Usos-ei, Todos-Potenciais-Usos-ed) (Vincenzi, 2004). O par Todos-Nós-ei e Todos-Nós-ed, o Todos-Arcos-ei e Todos-Arcos-ed, bem como o par Todos-Usos-ei e Todos-Usos-ed correspondem, respectivamente, aos critérios tradicionais Todos-Nós, Todos-Arcos e Todos-Usos. O par Todos-Potenciais-Usos-ei, Todos-Potenciais-Usos-ed corresponde ao critério Todos-Potenciais-Usos (Maldonado, 1991). Essa divisão dos critérios tradicionais em dois tipos diferentes ocorreu para representar o mecanismo de exceção de Java, representado nos critérios secundários (Vincenzi et al., 2003).

As atividades principais executadas pela JaBUTi para executar a análise de cobertura são: instrumentar arquivos de classe, coletar informação de cobertura durante a execução do programa, e determinar como as classes serão testadas, dado um determinado critério de teste. Relatórios dos diferentes tipos de teste são gerados para ajudar o testador a decidir quando parar os testes ou quais partes ainda não foram suficientemente testadas. Os relatórios são gerados de acordo com cada projeto, cada classe, cada método, cada caso de teste (Vincenzi, 2004; Vincenzi et al., 2003).

Os casos de teste que a ferramenta JaBUTi aceita estão no formato do JUnit (Massol e Husted, 2005). O JUnit é um framework de teste que vem sendo bastante utilizado e que viabiliza a documentação e execução automática de casos de teste. O framework JUnit é de código aberto e pode ser utilizado para escrever e executar de forma automática um conjunto de teste fornecendo relatórios sobre quais casos de teste não se comportaram de acordo com o que foi especificado. A idéia básica é implementar algumas classes específicas que armazenam informação sobre os dados de entrada e a respectiva saída esperada para cada caso de teste. Após a execução de um caso de teste, a saída obtida é comparada com a saída esperada e quaisquer discrepâncias são reportadas. O principal problema do JUnit é que ele não fornece informações a respeito da cobertura obtida pelos casos de teste. O JUnit pode ser utilizado mesmo que somente o bytecode e a especificação do programa estejam disponíveis (Massol e Husted, 2005).

2.3.5 Ensino de Teste de Software

No contexto de engenharia de software, teste é uma das mais importantes atividades na garantia da qualidade e da confiabilidade do software que está sendo desenvolvido mas, ao mesmo tempo, é um tópico difícil para ensinar ou aprender sem suporte apropriado. Um profissional qualificado em engenharia de software deve ter domínio em teste de software. Com o aumento da demanda por profissionais qualificados em engenharia de software, o desenvolvimento das habilidades de teste nos estudantes pode ter um papel fundamental na melhora da qualificação desses profissionais. Desta forma, uma iniciativa interessante é ensinar a atividade de teste o mais cedo possível aos estudantes (Barbosa et al., 2003a).

Tradicionalmente, teste tem sido ensinado de acordo com a abordagem clássica de desenvolvimento de software, o modelo cascata (Pressman, 2005), 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. Portanto, a atividade de teste vem sendo ensinada somente no final do processo de aprendizagem dos cursos de Ciências da Computação. Além disso, somente uma pequena porção dos currículos dos cursos de Ciências da Computação é alocada para a atividade de teste de software em comparação com outras atividades do processo de desenvolvimento de software (Shepard et al., 2001).

Por outro lado, experiências recentes sugerem que a atividade de teste deve ser ensinada o mais cedo possível no processo de aprendizado. Dessa forma, a aquisição de habilidades o mais cedo possível de conceitos e técnicas de teste pode: (1) melhorar o raciocínio sobre os programas (e sua solução), conduzindo para uma melhor qualidade dos produtos; e (2) induzir e facilitar o uso de teste por todo o processo de desenvolvimento de software, conduzindo a uma melhor qualidade do processo, em contraste com as práticas correntes. Desta forma, a inclusão da atividade de teste de software o quanto antes pode ajudar tanto a aumentar a qualidade do produto final produzido bem como melhorar a qualidade do processo utilizado no seu desenvolvimento. No caso do aumento da qualidade do processo utilizado, a atividade de teste ensinada o mais cedo possível pode melhorar tanto as abordagens tradicionais de desenvolvimento, como o modelo cascata, onde o teste acontece somente no final do processo de desenvolvimento, como em experiências modernas que sugerem que o teste deve ser introduzido durante todas as fases do desenvolvimento de software, garantindo com isso a qualidade do sistema e reduzindo os custos de desenvolvimento. Um exemplo disso é a metodologia ágil de desenvolvimento de software *eXtreme Programming* (Beck, 1999). 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 (Beck, 1999).

De acordo com Patterson et al. (2003), 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 tornar-se 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 (Goldwasser, 2002).

Outras experiências também vêm sendo conduzidas no sentido de ensinar teste o mais cedo possível. Os alunos que aprendem teste mais cedo podem tornarem-se melhores desenvolvedores, pois o teste força a integração e a aplicação das habilidades do desenvolvedor de software de análise, projeto e implementação (Jones, 2001a). Contudo, como já mencionado anteriormente, 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 (Shepard et al., 2001).

Apesar disso, 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 et al. (2003):

- 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, usualmente mais longos que os próprios programas 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.

Além disso, segundo Edwards (2004), existem algumas barreiras na adoção de práticas de teste de software para alunos que estão se iniciando em programação:

1. Teste de software requer experiência em programação e muitas vezes, estudantes novatos não possuem essa experiência;
2. Professores não dispõem de tempo hábil (em termos de horas-aula) para ensinar um novo tópico, como teste de software, em cursos já sobrecarregados;
3. Professores gastam muito tempo corrigindo os trabalhos dos alunos, o que dificulta sua disponibilidade 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 seu desempenho 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 et al. (2002) realizaram 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 de 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. Nesse sentido, vários outros trabalhos também vêm sendo realizados no sentido de se construir ambientes que dêem suporte ao ensino do teste. Na Seção 2.4 é realizada uma breve descrição de ambientes educacionais em geral e também daqueles que fornecem suporte ao ensino de teste de software.

2.4 Ambientes Educacionais

Ambientes educacionais constituem ferramentas de apoio ao processo de ensino e aprendizado. Essas ferramentas fornecem suporte à disponibilização e apresentação de material, acompanhamento didático, apoio ao trabalho colaborativo e à captura de material multimídia em salas de aula convencionais. Esses ambientes podem ser usados como ferramentas de apoio ao ensino integrado de programação e teste de software. A seguir são descritos alguns ambientes educacionais existentes.

- **WebCT:** O *WebCT (Web Course Tools)* (Goldberg et al., 1996), desenvolvido na *University of British Columbia*, consiste em uma iniciativa internacional utilizada com sucesso em programas de ensino, aprendizado e treinamento em várias universidades de diferentes países. Basicamente, o ambiente fornece um conjunto de ferramentas que facilitam a criação de cursos baseados no ambiente WWW, podendo ainda ser utilizado na disponibilização de material como ferramenta complementar de um curso já existente. A principal característica associada ao WebCT está na possibilidade de se estabelecer um ambiente de ensino e aprendizado integrado, contendo uma série de ferramentas educacionais tais como sistema de conferência, chat, correio eletrônico, acompanhamento do aluno, suporte para projetos colaborativos, auto-avaliação, questionários, distribuição e controle de notas, glossário, controle de acesso, calendário do curso, geração automática de índices e pesquisa, entre outras.
- **AulaNet:** O AulaNet (Lucena et al., 1998) é uma iniciativa nacional, tendo sido desenvolvido no Laboratório de Engenharia de Software (LES) do Departamento de Informática da Pontifícia Universidade Católica do Rio de Janeiro (PUC/RJ). É um ambiente de aprendizado cooperativo baseado na *Web*, com suporte para administração, criação, manutenção e assistência de cursos à distância. Os cursos criados no ambiente enfatizam a cooperação entre alunos e entre aluno e professor, sendo apoiados por uma variedade de tecnologias disponíveis na Internet.

- **CoWeb:** A *CoWeb(Collaborative Website)* (Guzdial et al., 2002) é uma ferramenta de edição colaborativa baseada na *Web*, desenvolvida no *Georgia Institute of Technology*. A *CoWeb* fornece um espaço colaborativo no qual os alunos podem trabalhar juntos, ajudando-se mutuamente na articulação do conhecimento para a realização das tarefas e projetos propostos. Ainda, a ferramenta pode ser utilizada como um ambiente de apoio a revisões, sugestões e críticas no qual os alunos discutem os trabalhos e projetos uns dos outros. Segundo Guzdial (Guzdial et al., 2002), o desenvolvimento e o uso da *CoWeb* no contexto educacional estão baseados na idéia de que novas oportunidades de aprendizado são criadas quando os alunos são motivados a articular seu conhecimento por meio da elaboração de documentos multimídia, quando revisam seus trabalhos e quando recebem comentários e críticas sobre aquilo que fizeram. A *CoWeb* fornece recursos para transferência do conhecimento, podendo ser utilizada como mecanismo de apoio aos processos de disponibilização e manutenção de material didático.
- **Panda:** O Panda (IME-USP, 2006) é um sistema online de acompanhamento didático desenvolvido no Instituto de Matemática e Estatística da Universidade de São Paulo (IME/USP). Seu principal objetivo é agilizar e aperfeiçoar o contato entre aprendizes e professores disponibilizando uma série de funcionalidades que podem ser utilizadas pelos envolvidos no processo de aprendizagem (alunos e professores). Submissão automática de trabalhos práticos, disponibilização de notas de aulas, notícias e links, publicação de datas (provas e trabalhos) e, posteriormente, das notas são algumas das funcionalidades presentes no Panda. Além disso, mecanismos de comunicação, como por exemplo, fórum e envio de mensagens públicas e privadas também estão disponíveis. O sistema é intuitivo e de fácil utilização permitindo que usuários que não tenham contato frequente e nem experiência com o uso da Internet não encontram dificuldades em utilizá-lo.
- **CoL:** O ambiente CoL (Cursos on-Line) (LARC - USP, 2006) é um sistema para criação e utilização de ambientes educacionais pela Internet, desenvolvido no Laboratório de Arquitetura e Redes de Computadores da Escola Politécnica (LARC/USP). O ambiente hospeda cursos, oferecendo o suporte necessário para a organização de conteúdo, trabalhos, testes, FAQ automático, mensagens para a turma, entre outros recursos. Possui ferramentas interativas, como chat e lista de discussão. Além disto, o docente pode acompanhar a evolução do aluno por meio de testes objetivos e de estatísticas de acesso, pois o sistema registra informações a respeito das páginas acessadas.

Esses ambientes, conforme mencionado anteriormente, fornecem suporte à disponibilização e apresentação de material, acompanhamento didático, apoio ao trabalho colaborativo, à captura de material multimídia em salas de aula convencionais e ao gerenciamento de cursos *online*. Desta forma, é interessante observar que tais ambientes não fornecem apoio ao ensino de teste de software e nem possuem integração com ferramentas de teste. A seguir, são mostrados alguns ambientes que apóiam o ensino de teste, concomitantemente ao ensino de programação.

2.4.1 Ambientes de Apoio ao Ensino de Teste de Software

Seguindo a tendência da inclusão da atividade de teste de software o mais cedo possível nos cursos de Ciência da Computação, trabalhos vêm sendo conduzidos a fim de fornecer suporte ao ensino de teste de software. Nesse sentido, 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. A outra é a 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. Nas Seções 2.4.1.1 e 2.4.1.2 esses ambientes são sucintamente descritos.

2.4.1.1 Ambientes de Programação Pedagógicos

Os ambientes profissionais integrados de desenvolvimento são projetados para o desenvolvimento de grandes sistemas. Frequentemente, 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 (Patterson et al., 2003).

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 (Allen et al., 2002) e o BlueJ (Patterson et al., 2003):

- **DrJava:** DrJava (Allen et al., 2002) 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. Um ponto importante do DrJava é o suporte que o ambiente fornece à atividade de teste. A interface interativa pode ser utilizada para a realização de teste *ad hoc* e, além disso, o DrJava é integrado ao JUnit, que é um framework de código aberto utilizado para a realização de testes de unidade de programas Java (Massol e Husted, 2005), facilitando a realização de testes de unidade por parte dos alunos iniciantes.
- **BlueJ:** BlueJ (Patterson et al., 2003) é um ambiente de programação pedagógico para Java, desenvolvido especificamente para facilitar o ensino de programação orientada a objetos. O ambiente permite a interação direta dos alunos com as classes criando objetos e realizando a chamada a seus métodos graficamente. O BlueJ também fornece suporte à atividade de teste de software pois é integrado ao JUnit, possuindo um mecanismo que permite a criação interativa de casos de teste no JUnit. Permite, ainda, a realização de testes *ad hoc*.

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. Eles apenas facilitam a construção de casos de teste no JUnit. Conforme mencionado na Seção 2.3.4, as ferramentas de teste automatizam a aplicação das técnicas e critérios de teste e são fundamentais no suporte ao ensino de teste de software pois permitem que os alunos possam realizar na prática os conceitos aprendidos. Nesse sentido, é importante que os ambientes que forneçam apoio ao ensino integrado de programação e teste de software sejam integrados com ferramentas de teste.

2.4.1.2 Ambientes de Submissão e Avaliação de Trabalhos

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 o desempenho 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 (Edwards, 2004).

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, o aluno pode saber como está a seu desempenho (Isong, 2001).

Isong (2001) propõe um sistema para automatizar a avaliação de programas. A principal característica desse sistema é avaliar a precisão dos programas dos alunos. O professor é responsável por criar a especificação do programa e a partir dela os casos de teste que serão utilizados para avaliar os programas dos alunos. Assim, o sistema executa os programas dos alunos utilizando os casos de teste desenvolvidos pelo professor e compara com os resultados esperados.

O ambiente proposto por Jones (2001b), o PGSE, é um sistema baseado em UNIX para submissão e avaliação automática de programas. O PGSE armazena o programa submetido, compila e executa o programa. O professor tem que manualmente inserir entradas para o programa e o sistema executa os programas com essas entradas definidas e compara com os resultados esperados gerando relatórios de acordo com o resultado encontrado. Esse sistema não permite que os alunos submetam novamente seus programas após a primeira submissão não sendo possível que os alunos corrijam os erros que o sistema encontrou nos seus programas.

O sistema ASSYST (Jackson e Usher, 1997) é utilizado para avaliar automaticamente programas e ajuda a analisar além do comportamento correto dos programas outras questões relativas a programação de software como documentação, estilo e projeto. O sistema também requer que os alunos submetam um conjunto de casos de teste juntamente com a implementação dos seus programas. O processo de avaliação está baseado na execução dos programas utilizando os casos de teste dos alunos como base para comparação.

A abordagem proposta por Goldwasser (2002) permite que os alunos submetam tanto a implementação do programa como um conjunto de casos de teste. O sistema possui uma implementação de referência e um conjunto de casos de teste. Desta forma, a avaliação é baseada na porcentagem de casos de teste que executam corretamente no código do aluno e na porcentagem de defeitos que os casos de teste dos alunos encontram na implementação de referência existente no sistema.

Conforme descrito acima, existem alguns ambientes para submissão e avaliação automática de trabalhos práticos que podem facilitar o ensino integrado de teste e de fundamentos de programação, como (Goldwasser, 2002; Isong, 2001; Jackson e Usher, 1997; Jones, 2001b; Korhonen et al., 2002; Saikkonen et al., 2001). 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. Esses ambientes, no entanto, não utilizam nenhuma ferramenta de teste para avaliar os programas e os casos de testes dos alunos. Eles apenas comparam a saída produzida pelos programas com saída esperada sendo que nenhum critério de teste é utilizado.

Nesse sentido, a fim de apoiar o ensino de fundamentos de programação e de teste de software, este trabalho propõe também a construção de um ambiente de submissão e avaliação automática de trabalhos práticos o qual deve avaliar tanto os programas quanto os casos de teste fornecidos pelos alunos. Assim, pretende-se analisar tanto a qualidade do programa quanto da atividade de teste realizada pelos estudantes. Para isso, ferramentas de teste que automatizem a aplicação de critérios de teste devem ser integradas ao ambiente para permitir que seja realizada uma análise de cobertura baseada em critérios de teste.

2.5 Mecanismos para o Desenvolvimento de Módulos Educacionais

Conforme mencionado anteriormente, um dos objetivos deste trabalho é o desenvolvimento de um módulo educacional integrado de fundamentos de programação OO e teste de software. Para a construção desse módulo educacional integrado foram utilizados os mecanismos de suporte para o desenvolvimento de módulos educacionais propostos no trabalho de doutorado de Barbosa (2004). Esses mecanismos correspondem a um processo padrão para a elaboração de

módulos educacionais e mecanismos de suporte para modelagem dos conteúdos educacionais associados. A seguir é realizada uma breve descrição do que são módulos educacionais e os mecanismos para seu desenvolvimento presentes em (Barbosa, 2004).

Módulos Educacionais são unidades concisas de estudo, compostas basicamente por conteúdos teóricos integrados a atividades práticas e avaliações, cuja disponibilização aos aprendizes é apoiada por recursos tecnológicos e computacionais (Barbosa et al., 2003b). O módulo educacional é composto pelo material didático associado, envolvendo os conteúdos referentes ao domínio de conhecimento que se deseja ensinar, e pelos ambientes educacionais e mecanismos de apoio, representando a infra-estrutura necessária à utilização do módulo.

O material didático é constituído por conteúdos teóricos e práticos. Conteúdos teóricos podem ser vistos como livros, artigos, informações e referências na *Web*, anotações de aula, audio e vídeo associados, entre outros. Conteúdos práticos são caracterizados em função das atividades e avaliações conduzidas e de seus produtos resultantes (documentos, códigos-fonte, programas executáveis, discussões, experimentos). Ferramentas específicas pertinentes ao domínio de conhecimento bem como os resultados obtidos a partir de sua utilização também são considerados conteúdos práticos (Barbosa, 2004).

2.5.1 Processo Padrão para Elaboração de Módulos Educacionais

Um dos mecanismos para o desenvolvimento de módulos educacionais propostos por Barbosa (2004) é o processo padrão para elaboração/evolução de módulos educacionais que consiste no estabelecimento de diretrizes para o desenvolvimento sistemático de módulos educacionais. Esse processo é baseado na norma ISO/IEC 12207 (ISO, 1995) ao contexto de produção de módulos educacionais, ao qual foram incorporados aspectos de modelagem de conteúdos educacionais, práticas estabelecidas pelas teorias de projeto instrucional e questões relacionadas ao trabalho distribuído e cooperativo (Barbosa, 2004).

O desenvolvimento de módulos educacionais envolve, além de aspectos técnicos e organizacionais, características específicas associadas ao aprendizado. Quanto ao aprendizado, deve-se levar em consideração a aplicação de teorias e princípios educacionais, como paradigmas instrucionais, estratégias de motivação e estilos de aprendizado, e a seleção e utilização de ferramentas apropriadas ao ensino e aprendizagem (Barbosa et al., 2003b). Além disso, o desenvolvimento de módulos educacionais possui similaridade com o desenvolvimento de software, pois requer o estabelecimento e aplicação de práticas sistemáticas e disciplinadas que garantam a produtividade do processo de desenvolvimento e a qualidade dos produtos gerados. A norma internacional ISO/IEC 12207 (ISO, 1995) insere-se nessa perspectiva, visando a definir e a padronizar os processos e as atividades básicas que compõem o processo de desenvolvimento de um software e por isso foi escolhida para ser adaptada ao contexto educacional (Barbosa, 2004).

Na norma ISO/IEC 12207, os processos do ciclo de vida são agrupados em três classes: fundamentais, de apoio e organizacionais. No estabelecimento do processo padrão para módulos educacionais foi mantida a mesma estrutura de classes de processos, mas internamente adaptações foram necessárias para adequar a mesma ao contexto educacional. Atividades referentes ao processo padrão para equipes geograficamente dispersas (Maidantchik, 1999) também foram incorporadas ao processo.

A estrutura geral do processo padrão para a elaboração de módulos educacionais é mostrada na Figura 2.2. Os processos de aquisição, fornecimento, desenvolvimento, operação, manutenção, documentação, gerência de configuração, garantia de qualidade, verificação, validação, revisões, auditoria, resolução de problemas, gerência, melhoria, infra-estrutura e treinamento foram extraídos da norma ISO/IEC 12207 e adaptados ao contexto de ensino e treinamento. Os processos de definição, planejamento, publicação de resultados, capacitação, controle de artefatos, comunicação e coordenação foram extraídos e adaptados do processo padrão para equipes geograficamente dispersas. Os processos de disponibilização, gerência do conhecimento, copy-right e licença foram incluídos por Barbosa (2004) e são específicos ao contexto de módulos educacionais.

Figura 2.2: Estrutura Geral do Processo Padrão para Módulos Educacionais (Barbosa, 2004)

2.5.2 Modelagem de Conteúdos Educacionais

Dentro do contexto de desenvolvimento de módulos educacionais, Barbosa (2004) também investigou a atividade de modelagem de conteúdos educacionais. A modelagem de material didático de qualidade, uma atividade que faz parte do processo padrão, capaz de motivar os aprendizes e contribuir efetivamente em seu processo de construção do conhecimento, representa um aspecto essencial no contexto de desenvolvimento de módulos educacionais. Assim, a elaboração de material didático de qualidade representa um aspecto essencial no desenvolvimento de módulos educacionais, sendo uma tarefa complexa e que requer que uma série de fatores, tanto pedagógicos como de desenvolvimento, sejam considerados. No entanto, apesar da sua relevância, existem poucas abordagens para modelar conteúdos educacionais. Além disso, essas iniciativas possuem diferentes perspectivas, que são apropriadas para um determinado cenário de aprendizado mas inadequado para outros (Barbosa, 2004).

Motivado por esse cenário, Barbosa (2004) propôs um conjunto preliminar de requisitos de modelagem (Barbosa et al., 2002): (1) taxonomia de conceitos; (2) composição de conceitos; (3) relacionamentos específicos; (4) decomposição hierárquica; (5) diferenciação entre categorias do conhecimento; (6) contextos de aprendizado; (7) ordem pedagógica; (8) mecanismos de história; e (9) propagação de eventos. Também foram identificadas algumas perspectivas de modelagem, a fim de caracterizar os modelos para a representação dos conteúdos educacionais (Barbosa, 2004; Barbosa et al., 2002). A seguir são descritas essas perspectivas de modelagem identificadas (Barbosa, 2004):

Perspectiva Conceitual: Trata especificamente da modelagem dos conceitos relevantes do domínio de conhecimento e da forma com que tais conceitos se inter-relacionam nesse domínio.

Perspectiva Instrucional: Aborda a modelagem de informações adicionais e elementos complementares sobre o domínio de conhecimento.

Perspectiva Didática: É responsável por associar os objetos anteriormente modelados, estabelecendo uma sequência de apresentação entre eles.

Dessa forma, Barbosa (2004) estabeleceu a conexão entre as perspectivas e os requisitos e definiu os modelos para modelar conteúdos educacionais e sua integração em uma abordagem integrada para modelagem de conteúdos — a abordagem *AIM-CID*. A Figura 2.3 mostra a abordagem *AIM-CID* que consiste na reunião, em uma única proposta, das perspectivas de modelagem conceitual, instrucional e didática (Barbosa, 2004). Essa abordagem constitui o outro mecanismo para o desenvolvimento de módulos educacionais.

Figura 2.3: AIM-CID: Abordagem Integrada para Modelagem de Conteúdos Educacionais (Barbosa, 2004)

2.5.2.1 Modelo Conceitual

Em linhas gerais, o modelo conceitual consiste de uma descrição de alto-nível do domínio de conhecimento, na qual são representados os conceitos associados e seus inter-relacionamentos (Barbosa, 2004; Barbosa et al., 2002). Os elementos do domínio do conhecimento são caracterizados essencialmente em termos de seus conceitos. Os relacionamentos entre os conceitos podem ser divididos em duas classes:

Relacionamentos Estruturais: Representam uma categoria genérica de relacionamentos, independentes de domínio, podendo ainda ser pré-definidos de acordo com o modelo conceitual utilizado. Permite estabelecer a taxonomia e a composição dos conceitos.

Relacionamentos Específicos: Representam as relações específicas, na qual as interpretações dependem do domínio do conhecimento.

A taxonomia dos conceitos refere-se à representação hierárquica do domínio do conhecimento, onde conceitos gerais devem ser apresentados antes dos conceitos específicos. A idéia da composição de conceitos indica que um conceito pode ser definido e explicado em termos de outros, podendo ser usado como um mecanismo para facilitar a compreensão do domínio do conhecimento (Barbosa, 2004; Barbosa et al., 2002).

Após a definição dos conceitos e dos relacionamentos, os mesmos devem ser organizados e representados de forma estruturada no modelo para facilitar a compreensão e o entendimento do domínio. Para a construção do modelo conceitual, optou-se pela utilização da técnica de Mapas Conceituais (Novak, 1990). Essa técnica foi escolhida por ser adequada para a representação de

conceitos e seus relacionamentos e para a estruturação hierárquica do domínio de conhecimento; é fácil de utilizar, sendo uma técnica bastante intuitiva, mesmo para usuários não familiarizados; é fundamentada em princípios educacionais, possuindo grande aceitação por especialistas e profissionais da área de Educação (Barbosa, 2004).

Para a construção do modelo conceitual, algumas restrições/extensões foram incorporadas à técnica de Mapas Conceituais (Barbosa, 2004):

- Apenas conceitos devem ser representados no mapa conceitual. Essa restrição é justificada pelo fato de que exemplos (além de outras informações associadas) serão tratados no modelo instrucional.
- Além de relacionamentos genéricos, já estabelecidos pela própria definição da técnica, primitivas para a representação de taxonomia e composição de conceitos também devem ser incorporadas.

2.5.2.2 Modelo Instrucional

Para uma melhor compreensão e assimilação de um dado domínio do conhecimento, uma diversidade de informações pode ser associada aos conceitos relatados, em diferentes caminhos. Além dos conceitos, itens de informação e elementos instrucionais devem ser considerados como parte do domínio do conhecimento. O modelo instrucional é responsável pela definição de informações adicionais relativas aos conceitos previamente identificados. Dessa forma, é no modelo instrucional que os itens de informação e elementos instrucionais são modelados. Por itens de informação compreende-se outros elementos significativos do domínio do conhecimento que devem ser representados e incorporados ao conteúdo educacional. Para o estabelecimento dos itens de informação é considerada a teoria de Merrill (Merrill, 1983) que especifica os itens de informação em conceitos, fatos, procedimentos e princípios. Por elementos instrucionais entende-se: descrições, exemplos, exercícios, sugestões de estudo e quaisquer outros mecanismos que possam ser associados aos conceitos do domínio de conhecimento (Barbosa, 2004; Barbosa et al., 2002).

Três tipos de elementos instrucionais são identificados: explanatório, exploratório, e de avaliação. Os elementos explanatórios são usados para explicar, ilustrar, motivar e complementar um dado tópico, e para isso, são utilizados exemplos, explicações, referências, entre outros. Os elementos exploratórios permitem que os alunos naveguem através do domínio do conhecimento e pratiquem os conceitos relacionados que eles supostamente aprenderam. Problemas com sugestões e dicas para resolução podem ser usados como elementos exploratórios. Elementos de avaliação avaliam o desempenho e o comportamento do aprendiz. Estes elementos podem ser de avaliação diagnóstica, formativa ou somativa (Barbosa, 2004; Barbosa et al., 2002).

Na construção do modelo instrucional, optou-se pelo uso simplificado do modelo HMBS (Turine, 1998; Turine et al., 1997) considerando, nesse nível somente os aspectos de decom-

posição hierárquica fornecidos pelo mesmo. O modelo HMBS consiste em um modelo para a especificação da estrutura e da semântica navegacional de hiperdocumentos, utilizando a técnica *statechart* como modelo formal (Barbosa, 2004).

O modelo HMBS foi estendido para permitir a representação de diferentes categorias de conhecimento, a fim de garantir sua completa adequação aos aspectos considerados na perspectiva instrucional (Barbosa, 2004). A utilização do HMBS pode ser visto como uma evolução do modelo conceitual baseado em Mapas Conceituais para o modelo instrucional. De fato, os mecanismos de decomposição e visões hierárquicas podem ser utilizados para complementar, no nível instrucional, a idéia de estruturação hierárquica explorada no nível conceitual (Barbosa, 2004).

O modelo HMBS, desvinculado da representação de transições, eventos e mecanismos de história, e estendido para a representação de diferentes categorias de conhecimento, foi denominado *HMBS/Instrucional* (Barbosa, 2004).

2.5.2.3 Modelo Didático

Finalmente, no modelo didático os elementos instrucionais são relacionados aos objetos conceituais e a precedência entre esses relacionamentos é definida. Diferentes maneiras de utilização e disponibilização do conteúdo educacional, estabelecidas segundo os objetivos do curso, a estratégia pedagógica do professor e o perfil de cada aluno, podem ser representadas por meio de modelos didáticos (Barbosa, 2004; Barbosa et al., 2002).

Basicamente, os modelos didáticos são responsáveis pelo estabelecimento de relações de precedência e relacionamentos didáticos, definindo seqüências de apresentação entre os objetos caracterizados no modelo instrucional (itens de informação e elementos instrucionais). Por relacionamentos didáticos entendem-se relações tais como complementa, exemplifica, ilustra, motiva, exercita e avalia, entre outras. Estruturas de acesso, tais como índices, roteiros e visitas guiadas também podem ser utilizadas (Barbosa, 2004).

No nível didático, também optou-se pela utilização do modelo HMBS representando, além dos aspectos já tratados no *HMBS/Instrucional*, elementos associados à navegação e à especificação de aspectos comportamentais do hiperdocumento educacional (Barbosa, 2004).

O comportamento associado às possíveis formas de navegação do usuário (aprendiz ou instrutor) pelo hiperdocumento constitui um aspecto essencial sob a perspectiva didática da modelagem de conteúdos educacionais. Por exemplo, com base em informações comportamentais é possível personalizar e controlar o caminho a ser percorrido pelo aprendiz em tempo de execução (Barbosa, 2004).

O modelo permite ainda a incorporação de outros aspectos relevantes tanto sob a perspectiva didática (história, propagação de eventos, definição de contextos de aprendizado) como sob a perspectiva genérica de desenvolvimento (validação do hiperdocumento) (Barbosa, 2004).

A principal extensão proposta ao HMBS para sua utilização no nível didático refere-se ao estabelecimento de uma semântica de navegação livre, em que seja possível representar simultaneamente todas as seqüências de apresentação entre conceitos e informações pertinentes associadas. Em outras palavras, a idéia é permitir que o modelo seja suficientemente flexível na definição e representação das seqüências de apresentação (navegação) do conteúdo. Para representar as semânticas de navegação livres foi incorporada ao modelo HMBS a idéia de estados *DD* (*Dynamically Defined - Definido Dinamicamente*) (Barbosa, 2004). Mais detalhes encontram-se em Barbosa (2004).

No modelo didático têm-se três tipos de especificações sobre os aspectos de navegação: aberta, parcialmente aberta e fechada. Na especificação aberta estão representadas todas as possíveis seqüências de apresentação entre os objetos modelados. Na especificação parcialmente aberta algumas possibilidades de navegação foram desconsideradas. E por último, na especificação fechada os aspectos de navegação são pré-determinados, ou seja, a ordem de apresentação dos objetos é fixa, devendo ser previamente definida pelo especialista e/ou instrutor responsável pelo módulo. O modelo didático especificado da forma aberta corresponde à base a partir da qual todas as possíveis seqüências de apresentação podem ser derivadas. A decisão sobre qual tipo de especificação utilizar deve ser baseada nos perfis dos aprendizes e do próprio instrutor, bem como nas características esperadas do módulo (Barbosa, 2004).

O modelo HMBS nesse nível é chamado de *HMBS/Didático*. Cabe ressaltar que as extensões propostas para a representação de diferentes categorias de conhecimento, definidas no modelo *HMBS/Instrucional*, são preservadas no modelo. Além disso, a estruturação hierárquica explorada nos níveis conceitual e instrucional também é preservada no *HMBS/Didático*. Garante-se, com isso, a uniformidade da abordagem de desenvolvimento (Barbosa, 2004).

2.6 Considerações Finais

Neste capítulo foi apresentada uma síntese das principais pesquisas que serviram de base para a realização desse trabalho, cujos principais objetivos são: (i) o desenvolvimento de um módulo educacional integrado de fundamentos de programação OO e teste de software; (ii) e a construção de um ambiente para submissão e avaliação de trabalhos práticos dos alunos. Nessa perspectiva, foram abordados os principais problemas encontrados na literatura sobre o ensino de programação e de teste de software bem como o que tem sido realizado para tentar amenizar esses problemas. Uma das tendências no ensino de programação é a inclusão da orientação a objetos em cursos introdutórios. Entretanto, a mudança de paradigma não extingue os problemas relativos ao ensino de programação. Uma alternativa para tentar resolver esses problemas é a inserção da atividade de teste de software no escopo das disciplinas introdutórias de programação.

A proposta deste trabalho é fornecer mecanismos para que o ensino integrado de programação e teste de software se concretize. Para isso, como já mencionado anteriormente, foi desenvolvido um módulo educacional integrado e um ambiente para submissão e avaliação de trabalhos práticos baseados em atividades de teste. Dessa forma, foi realizada uma revisão dos ambientes existentes que forneçam suporte a atividade de teste. Finalmente, foi relatado um processo padrão para a construção de tais módulos com ênfase na atividade de modelagem de conteúdos educacionais.

No próximo capítulo discute-se, dentro do escopo de módulos educacionais, como o módulo integrado de Orientação a Objetos e Teste de Software foi desenvolvido utilizando a abordagem *AIM-CID* (Abordagem Integrada de Modelagem - Conceitual, Instrucional e Didática) descrita neste capítulo. Conforme mencionado anteriormente, o paradigma OO foi escolhido para o desenvolvimento do módulo educacional integrado seguindo as tendências atuais no ensino de programação. No próximo capítulo também é realizada uma análise crítica dos mecanismos propostos por Barbosa (2004) e quais são as extensões que este trabalho de mestrado propõe a esses mecanismos.

Módulo Educacional Integrado de Orientação a Objetos e Teste de Software

3.1 Considerações Iniciais

Conforme mencionado nos capítulos anteriores, um dos objetivos deste trabalho é o desenvolvimento de um módulo educacional integrado de fundamentos de programação OO e teste de software. Esse módulo educacional é um dos mecanismos de suporte ao ensino integrado de fundamentos de programação e teste e foi desenvolvido com a concepção de módulo educacional livre (“material didático livre”). A idéia é gradualmente introduzir os conceitos de teste de software enquanto os conceitos de programação estão sendo ensinados aos estudantes. Este capítulo mostra como o módulo educacional integrado de programação OO e teste de software foi construído. Para a construção do módulo foram utilizados os mecanismos para desenvolvimento de módulos educacionais investigados por Barbosa (2004). Durante o uso desses mecanismos foi proposta uma evolução à abordagem *AIM-CID*. Tal evolução é apresentada e discutida na Seção 3.3. Na Seção 3.4 é descrita a modelagem do módulo educacional integrado usando a abordagem *IMA-CID* para os domínios de OO e teste de software. Uma visão geral da implementação do módulo integrado é mostrada na Seção 3.5.

3.2 Processo Padrão Instanciado para Elaboração de Módulos Educacionais

A atividade de instanciamento consiste na seleção e alocação de métodos, técnicas de desenvolvimento, recursos humanos e tecnológicos para cada uma das atividades de um projeto particular (Maidantchik, 1999). O processo padrão para elaboração de módulos educacionais mencionado no Capítulo 2, foi instanciado no trabalho Barbosa (2004). Esse processo padrão instanciado foi adotado neste trabalho para a elaboração do módulo educacional integrado de programação OO e teste de software.

Na instanciamento do processo padrão para módulos educacionais realizada por Barbosa (2004) é considerada sua utilização em projetos específicos que são conduzidos em uma dada instituição de ensino. Esses projetos envolvem a elaboração de módulos educacionais para serem aplicados presencialmente, tanto em disciplinas oferecidas aos alunos da instituição como em minicursos, tutoriais e palestras ministradas por seus professores em eventos científicos ou em organizações industriais (Barbosa, 2004).

No processo instanciado foram definidos, entre outros, os recursos humanos e as funções desempenhadas ao longo do processo, os métodos e as técnicas de desenvolvimento adotadas, e os recursos tecnológicos e computacionais necessários à realização do projeto. No estabelecimento de métodos e técnicas de desenvolvimento, a abordagem *AIM-CID* foi considerada para ser utilizada na modelagem de conteúdos educacionais.

Com relação aos recursos computacionais necessários à realização do projeto, três categorias básicas de ferramentas foram identificadas: ferramentas de autoria; ferramentas de ensino; e ferramentas administrativas. Com relação às ferramentas de autoria, consideram-se: (1) ferramentas para modelagem de conteúdos educacionais; (2) ferramentas para integração do conhecimento; e (3) ferramentas de editoração. Como ferramentas de ensino, são consideradas: (1) ferramentas de apresentação; (2) ferramentas colaborativas; (3) ferramentas de avaliação; e (4) ferramentas de captura. As ferramentas de apoio ao processo devem ser analisadas em termos de sua adequação à automatização das atividades e tarefas estabelecidas pelo processo padrão e em relação às perspectivas de desenvolvimento de módulos educacionais (Barbosa, 2004).

3.3 Evolução da Abordagem *AIM-CID*

A abordagem *AIM-CID* (Barbosa, 2004), descrita no capítulo anterior, trata dos aspectos de modelagem de um domínio do conhecimento, segundo três perspectivas distintas: conceitual, instrucional e didática. Durante o desenvolvimento do módulo integrado foram identificados dois domínios de conhecimento a serem representados: OO e teste de software. Tendo em vista que a abordagem *AIM-CID* trata a representação de um único domínio, também foi proposta

neste trabalho uma evolução à abordagem *AIM-CID* com a finalidade de incorporar, na fase de modelagem conceitual, a integração de vários domínios distintos de conhecimento.

Inicialmente, para cada domínio de conhecimento distinto é realizado um levantamento dos principais conceitos e é construído o modelo conceitual associado, de acordo com o que a abordagem *AIM-CID* propõe. Em seguida, são estabelecidos os pré-requisitos de cada domínio. Com o estabelecimento desses pré-requisitos são definidos os pontos onde os conceitos podem ser integrados. De fato, são estabelecidos pontos de conexão entre os domínios. A partir da definição dos pontos de conexão, constrói-se o modelo conceitual integrado. A partir do modelo conceitual integrado são desenvolvidos os modelos instrucionais e didáticos, construídos como definido na abordagem *AIM-CID*.

O módulo educacional de programação OO e teste de software foi desenvolvido usando essa evolução da abordagem *AIM-CID*. Primeiramente, foram identificados os conceitos de programação OO e construído seu modelo conceitual. Em seguida, foram identificados os conceitos de teste de software e desenvolvido seu modelo conceitual. Numa segunda fase, foram determinados os pré-requisitos dos conceitos de programação OO e teste de software. Por exemplo, quando se começa a ensinar programação, já podem ser introduzidas noções básicas da atividade de teste aos alunos. Nesse ponto, entretanto, os conceitos relativos à técnica de teste estrutural não devem ser introduzidos pois estes requerem que os alunos já tenham alguma noção de programação como, por exemplo, já tenham visto estruturas de fluxo de controle. Dessa forma, no ponto onde são introduzidos os conceitos de estruturas de fluxo de controle pode-se introduzir também os conceitos pertinentes à técnica de teste estrutural. Para cada conceito de programação foi realizada essa análise de quais conceitos de teste podiam ser introduzidos naquele ponto. Com isso, os pontos de conexão foram estabelecidos e o modelo conceitual integrado foi desenvolvido. Na Seção 3.4, a construção desse módulo é apresentada com detalhes.

3.4 Módulo Educacional Integrado: Modelagem dos Conteúdos

Como discutido no capítulo anterior, a modelagem de conteúdos educacionais possui um papel relevante na elaboração do material didático, auxiliando o autor/desenvolvedor na captura de partes importantes do assunto que se deseja ensinar bem como na estruturação adequada dessas informações, assim como o reúso do material. A elaboração de material didático de qualidade representa um aspecto essencial no desenvolvimento de módulos educacionais, sendo uma tarefa complexa e que requer que uma série de fatores, tanto pedagógicos como de desenvolvimento, sejam considerados. Nesse contexto, Barbosa (2004) definiu uma abordagem integrada de modelagem, a abordagem *AIM-CID*, que reúne perspectivas distintas de modelagem, propondo a construção de um conjunto de modelos associados: modelo conceitual, modelo instrucional e modelo didático, descrito no Capítulo 2.

No desenvolvimento do módulo integrado de programação OO e teste de software realizou-se a modelagem dos conteúdos educacionais usando a abordagem *AIM-CID* com a extensão proposta na seção anterior, resultando na construção de modelos conceituais, instrucionais e didáticos pertinentes ao domínio integrado de programação OO e teste de software. Nas seções seguintes são detalhados como cada um desses modelos foram construídos.

3.4.1 Modelo Conceitual

O modelo conceitual consiste de uma descrição de alto-nível do domínio de conhecimento, na qual estão representados os conceitos associados e seus inter-relacionamentos (Barbosa, 2004).

Inicialmente, para a definição dos conceitos utilizados na modelagem conceitual, foi realizado um levantamento dos materiais didáticos sobre OO e teste de software existentes. Foram considerados materiais de teste de software e programação OO elaborados pelo especialista/-professor, utilizados em disciplinas, minicursos, tutoriais e palestras sobre o assunto; materiais elaborados por outros professores do ICMC-USP, envolvendo aspectos pertinentes às áreas; livros de introdução à Ciência da Computação usando OO; materiais de outras instituições como IME-USP, UNICAMP, entre outras.

A partir do levantamento realizado dos materiais didáticos existentes, foi estruturado o módulo de OO e depois o módulo de teste de software, construindo separadamente cada modelo conceitual associado. Conforme discutido na Seção 3.3, posteriormente foram definidos os pontos de integração entre os conceitos de ambos os domínios e a partir disso foi construído o módulo integrado de OO e teste de software.

3.4.1.1 Modelo Conceitual: Programação OO

A POO introduz uma abordagem na qual o programador visualiza seu programa em execução como uma coleção de objetos cooperantes que se comunicam por meio de troca de mensagens. Cada um dos objetos é uma instância de uma classe e todas as classes formam uma hierarquia de classes unidas via relacionamento de herança (Vincenzi, 2004). Com base na revisão bibliográfica sobre POO, realizada no Capítulo 2, e no levantamento dos materiais didáticos e livros existentes sobre POO, foram extraídos os principais conceitos relacionados.

É importante destacar que esse módulo de programação modelado refere-se a um módulo básico e introdutório de programação OO. Dessa forma, ênfase foi dada à modelagem de conceitos básicos de fundamentos de programação e de OO. Os conceitos de fundamentos de POO foram estruturados como mostra o modelo conceitual da Figura 3.1, da seguinte forma:

- O conceito *Linguagem de Programação* possui os seguintes conceitos relacionados a ele: *Conceitos Básicos*, *Conceitos Essenciais*, *Estruturas de*

Figura 3.1: Modelo Conceitual de Orientação a Objetos

Fluxo de Controle, Arquivos, Tratamento de Exceção e Conceitos Avançados.

- Por Conceitos Básicos entendem-se os conceitos de Objeto, Método e Classe.
- Conceitos Essenciais representam os principais comandos de uma linguagem de programação. Dessa forma, em Comandos Essenciais são abordados os conceitos de Variáveis, Constantes, Operadores, Comandos I/O e Strings.
 - O conceito de Variáveis necessita do conceito de Tipos de Dados Primitivos que podem ser classificados em Inteiro, Real, Caractere. Um dos tipos de Variáveis representada é o Array que pode ser Array Unidimensional e Array Multidimensional.
 - Os Operadores podem ser de vários tipos, entre eles, operadores de Atribuição, Aritméticos, Relacionais, Lógicos, Condicional – Ternário.
- Estruturas de Fluxo de Controle são compostos pelos conceitos de Comandos de Seleção e Comandos de Repetição. As Estruturas de Fluxo de Controle utilizam as Expressões Lógicas que por sua vez, utilizam os Operadores Relacionais e Operadores Lógicos.

- Os Comandos de Seleção são compostos pelos conceitos de If-else e Switch. O conceito de If-else utiliza o conceito de Operadores Condicionais – Ternários.
- Os Comandos de Repetição são compostos pelos conceitos de while, do-while, for.
- O conceito de Tratamento de Exceção é composto pelos comandos de throw e try/catch.
- O conceito Conceitos Avançados é classificado em Herança, Polimorfismo e Encapsulamento.

3.4.1.2 Modelo Conceitual: Teste de Software

A atividade de teste de software tem como principal objetivo revelar a presença de erros ou defeitos no produto. Portanto, o teste bem sucedido é aquele que consegue determinar casos de teste para os quais o programa que está sendo testado falhe (Myers et al., 2004). No trabalho de Barbosa (2004) foi realizado o desenvolvimento da modelagem conceitual de teste de software. Com base nessa modelagem, na revisão bibliográfica sobre teste de software (apresentada no Capítulo 2), e nos levantamentos dos materiais didáticos existentes sobre teste de software foram, determinados os principais conceitos relacionados a esse domínio. Com essas informações, evoluiu-se a modelagem conceitual de teste existente acrescentando novos conceitos e relacionamentos referentes ao teste de software OO. Desta forma, os conceitos relativos à atividade de teste de software foram estruturados como mostra o modelo conceitual da Figura 3.2, da seguinte forma:

- O conceito de Teste de Software foi representado por várias categorias: Etapas de Teste, Fases de Teste e Técnicas de Teste.
- Teste de Software utiliza os conceitos de Conjunto de Casos de Teste, Casos de Teste, Oráculo de Teste, Falha, Erro, Defeito, Engano.
- O conceito de Etapas de Teste é composto por Planejamento, Projeto de casos de teste, Execução e Análise.
- O conceito de Fases de Teste é composto por Teste de Unidade, Teste de Integração e Teste de Sistema.
 - Os conceitos Teste de Unidade, Teste de Integração, Teste de Sistema relacionam-se com o conceito de qual é a menor unidade a ser testada, o método ou a classe. Dependendo de qual é essa menor unidade a ser testada tem-se os seguintes conceitos relacionados com as fases de teste:

Figura 3.2: Modelo Conceitual de Teste de Software

Teste Intra-método, Teste Inter-método, Teste Intra-classe e Teste Inter-classe.

- O conceito de Técnicas de Teste é dividido em Técnica Funcional, Técnica Estrutural, Técnica Baseada em Erros e Técnica Baseada em Estados. A ênfase deste módulo encontra-se nas técnicas funcional e estrutural.
 - O conceito de Técnica Funcional é composto pelos seguintes critérios funcionais: Particionamento em classes de equivalência, Análise do Valor Limite, Grafo Causa-Efeito, Método Partição-Categoria
 - O conceito de Técnica Estrutural é composto pelos seguintes critérios estruturais: Critérios baseados em fluxo de controle e Critérios baseados em fluxo de dados
 - * Os critérios baseados em fluxo de controle são compostos por Critério Todos-Nós, Critério Todos-Arcos, Critério Todos-Caminhos
 - * Os critérios baseados em fluxo de dados são compostos por Critérios Rapps & Weyuker, Critérios Potenciais-Usos

de Controle, as técnicas de teste devem ser reforçadas e os critérios de fluxo de controle Todas-Arestas (Myers et al., 2004) podem ser introduzidos.

Os conceitos relacionados para algum tipo específico de variáveis (como por exemplo arrays) e arquivos devem ser ensinados em conjunção com critérios de teste de fluxo de dados, enfatizando os critérios de Rapps & Weyuker (por exemplo, Todas-Definições, Todos-Usos) (Rapps e Weyuker, 1985). Finalmente, com a introdução de Tratamento de Exceção e Conceitos Avançados, os critérios de fluxo de dados devem ser reforçados e aprofundados. O conjunto de critérios estruturais definidos por Vincenzi (2004) (Todos-Nós-Dependentes-Exceção e Todos-Nós-Independentes-Exceção (Vincenzi, 2004)), o qual explora o uso dos critérios de fluxo de controle e fluxo de dados para teste de programas OO (Java byte-code), é um exemplo que pode ser explorado em etapas avançadas do processo de aprendizado de fundamentos de programação.

Embora o módulo educacional não seja diretamente derivado a partir do modelo conceitual, muitos dos aspectos considerados durante a modelagem conceitual podem ser observados nas transparências implementadas. A Figura 3.4, por exemplo, ilustra a transparência sobre conceitos básicos de programação OO. No *frame* esquerdo está representada a idéia de taxonomia associada aos conceitos básicos de programação OO — Objeto, Classe e Método. Conforme discutido anteriormente, existem pontos de conexão entre os módulos de programação OO e teste de software, os quais representam a integração entre esses dois domínios. Esses pontos de conexão são representados no material didático por meio de links. Por exemplo, na estruturação do modelo conceitual integrado, com a definição dos conceitos básicos de programação OO, as idéias gerais da atividade de teste, a terminologia de teste, as etapas de teste e as fases de teste podem ser introduzidas ao alunos. Dessa forma, os links para teste de software, terminologia, etapas de teste e fases de teste estão representados na transparência de conceitos básicos de OO, na Figura 3.4. Assim, a partir da transparência de conceitos básicos de OO, o instrutor pode também navegar para o módulo de teste de software, cobrindo os conceitos básicos de teste de software, terminologia de teste, etapas de teste e fases de teste.

É importante observar que o instrutor pode optar por parar a exploração do módulo de teste de software e retornar ao módulo de programação. Da mesma forma, que existem pontos de conexão no módulo de programação para o módulo de teste de software, existem pontos de conexão no módulo de teste de software para o módulo de programação. Isso permite a livre navegação entre os módulos fornecendo flexibilidade para o instrutor.

3.4.2 Modelo Instrucional

O modelo instrucional aborda a modelagem de informações adicionais (fatos, princípios, procedimentos) e elementos complementares (exemplos, exercícios, etc.) sobre o domínio. De acordo com a abordagem *IMA-CID*, esses aspectos são modelados segundo o *HMBS/Instrucio-*

Figura 3.4: Transparência sobre Conceitos Básicos de programação OO

nal, enfatizando os aspectos de decomposição hierárquica e abordando, ainda, mecanismos de apoio à diferenciação entre categorias do conhecimento (Barbosa, 2004).

A primeira etapa para a construção do modelo instrucional consiste na identificação de outras informações relevantes que possam ser associadas aos conceitos representados nos modelos conceituais das Figuras 3.1, 3.2, 3.3. De acordo com a abordagem *AIM-CID*, o mecanismo utilizado para a diferenciação das categorias do conhecimento foi baseado na teoria de Merrill (Merrill, 1983), estabelecendo quatro tipos básicos de informações: fato, conceito, princípio e procedimento (Barbosa, 2004).

A segunda etapa da construção do modelo instrucional consiste em identificar os elementos instrucionais a serem associados aos itens de informação. Basicamente, dois tipos de elementos instrucionais são representados no modelo instrucional: (1) explanatórios, na forma de exemplos e informações complementares; e (2) exploratórios, na forma de exercícios.

Seguindo as idéias propostas na evolução da abordagem *AIM-CID*, foi construído o modelo conceitual integrado de programação OO e teste e, a partir do mesmo, foi desenvolvido o modelo instrucional pertinente. Como o modelo conceitual integrado é extenso, a modelagem instrucional foi dividida em vários modelos instrucionais separados, os quais representam partes específicas do modelo conceitual. Dessa forma, os modelos instrucionais, foram divididos da seguinte forma: conceitos básicos, comandos essenciais, estruturas de fluxo de controle, arquivo, tratamento de exceção e

conceitos avançados. Sendo que, juntamente com esses conceitos de programação OO estão representados os conceitos de teste de software que podem ser introduzidos juntamente com os conceitos de programação.

Figura 3.5: Modelo Instrucional Integrado: Conceitos Básicos

Na Figura 3.5 está representado o modelo instrucional referente a conceitos básicos. Nesse modelo foi considerado uma parte do modelo conceitual referente aos Conceitos Básicos de programação OO, que são os conceitos de Objeto, Método e Classe. E os conceitos relacionados de teste de software que podem ser ensinados em conjunto com programação OO. São eles: Conceitos Básicos de Teste, Terminologia, Conjunto de Casos de Teste, Fases de Teste, Etapas de Teste, Técnicas de Teste. Desta forma, a integração dos conceitos de programação OO e teste de software também se reflete no modelo instrucional, uma vez que estão representados no modelo os conceitos de programação e os respectivos conceitos de teste de software que podem ser ensinados simultaneamente. Por exemplo, de acordo com a Figura 3.5, além da definição do conceito de Objeto representada por `Objeto:conceito:texto`, outras informações complementares podem ser associadas ao esse conceito, tais como `Objeto:fato:texto`, `Objeto:conceito:figura`, `Mensagem:conceito:texto`.

Figura 3.6: Transparências referentes ao conceito de Objeto

Na Figura 3.6 são ilustradas três transparências implementadas de acordo com o conceito de Objeto representado no modelo instrucional da Figura 3.5. A Figura 3.6 (a) mostra o item de informação correspondente ao `Objeto:conceito:texto`, `Objeto:fato:texto`. A Figura 3.6 (b) mostra a implementação correspondente ao `Objeto:conceito:figura`. Já a Figura 3.6 (c) mostra a implementação referente ao item

de informação `Mensagem:conceito:texto`. Todas as transparências contém botões de ação indicando que existem mais informações associadas ao conceito de `Objeto`.

No modelo instrucional subentende-se que todas as informações são apresentadas na mesma página/transparência. No entanto, as informações associadas ao conceito de `Objeto` foram apresentadas em três transparências. Essa “quebra” justifica-se em virtude da limitação de espaço imposta às transparências que foram elaboradas no *LaTex*. De fato, essa é uma limitação dependente do formato escolhido para a apresentação do módulo e da própria ferramenta de editoração utilizada. Por exemplo, caso o módulo fosse implementado como um conjunto de páginas HTML, uma única página com barras de rolagem seria suficiente para conter todas as informações relativas ao `Objeto`. Na tentativa de resolver essa limitação, botões de navegação foram inseridos nas transparências com o propósito de indicar que o mesmo item de informação possui mais de uma transparência associada (Barbosa, 2004).

Figura 3.7: Transparências referentes ao Exemplo e Exercício sobre Terminologia de Teste

No modelo instrucional, além de informações complementares, são identificados também os elementos explanatórios, representados na forma de exemplos, e os elementos exploratórios, representados na forma de exercícios. Como exemplo, pode-se citar o exemplo e o exercício sobre Terminologia de Teste que estão representados modelo instrucional da Figura 3.5. Na Figura 3.7 é ilustrado como esses elementos foram implementados em transparências

no módulo integrado de programação OO e teste de software. Na Figura 3.7 (a) é mostrada a transparência principal do conceito Terminologia de Teste; na Figura 3.7 (b) é mostrado um elemento explanatório (no caso, um exemplo) de Terminologia de Teste; e na Figura 3.7 é ilustrado um elemento exploratório (no caso, um exercício) de Terminologia de Teste.

Figura 3.8: Modelo Instrucional Integrado: Comandos Essenciais

A Figura 3.8 ilustra o modelo instrucional referente aos Comandos Essenciais da programação OO e os respectivos conceitos de teste de software que podem ser introduzidos juntamente com esses conceitos de programação OO — neste caso, os conceitos relacionados às Técnicas de Teste, mais especificamente à técnica funcional e à técnica estrutural (ênfatisando os critérios estruturais baseados em fluxo de controle). O conceito Todos-Nós, por exemplo, além da sua definição (Todos-Nós: conceito: texto), tem a informação do conceito Nó: conceito: texto associada a ele.

A implementação do item de informação Todos-Nós do modelo instrucional da Figura 3.8 é ilustrada na Figura 3.9. A transparência mostra a implementação do estado `Nó:conceito:texto` e do estado `Todos-Nós:conceito:texto`. Um ponto importante a ser observado na Figura 3.9 é a integração do módulo educacional com ferramentas específicas do domínio. Como pode-se observar no modelo instrucional da Figura 3.8 está modelado essa integração com a ferramenta de teste JaBUTi. A transparência da Figura 3.9 mostra a implementação dessa integração por meio de um link para ferramenta de teste JaBUTi.

Figura 3.9: Transparências referentes ao Critério Todos-Nós

3.4.3 Modelo Didático

O modelo didático é responsável pelo estabelecimento de relações de precedência e relacionamentos didáticos, definindo sequências de apresentação entre os objetos caracterizados no modelo instrucional. De acordo com a abordagem *AIM-CID*, o modelo didático também é baseado no modelo HMBS representando além dos aspectos já tratados no nível instrucional, elementos associados à navegação e à especificação de aspectos comportamentais do conteúdo educacional (Barbosa, 2004).

Diversos modelos didáticos podem ser derivados a partir de um único modelo instrucional. Esses vários modelos didáticos são construídos a partir do tipo de especificação do modelo didático. De acordo com a abordagem *AIM-CID* tem-se a definição das especificações fechada, parcialmente aberta e aberta. Na especificação fechada, os caminhos de navegação estão todos pré-definidos. Na especificação parcialmente aberta, tem-se alguns caminhos de navegação definidos previamente, enquanto os demais caminhos estão abertos. O instrutor/professor é o

responsável pela definição dos caminhos em aberto. Na especificação aberta todos os caminhos de navegação possíveis estão modelados e cabe ao instrutor/professor escolher qual o melhor caminho de navegação de acordo com o perfil do curso que ele está ministrando.

Segundo a abordagem *AIM-CID*, a decisão sobre qual tipo de especificação utilizar deve ser baseada nos perfis dos aprendizes e do próprio instrutor, bem como nas características esperadas do módulo (Barbosa, 2004). Dessa forma, na implementação do módulo educacional integrado programação OO e teste de software optou-se pela especificação aberta para oferecer maior flexibilidade ao curso. A especificação aberta é de fundamental importância na geração de conteúdos personalizados, cujos tópicos e a ordem em que os mesmos devem ser apresentados é estabelecida em função de fatores tais como tempo de apresentação, objetivos de aprendizado e público-alvo.

A especificação aberta é representada no modelo através dos estados *DD* (*Dynamically Defined* - Definido Dinamicamente) (Barbosa, 2004). Os estados *DD* são utilizados para representar todas as conexões entre os conceitos modelados. Dessa forma, todos caminhos de navegação são modelados e depois de implementadas em transparências o instrutor/professor pode definir qual o caminho que melhor se adapta ao seu contexto.

O modelo didático da Figura 3.10 consiste em uma especificação aberta dos aspectos de navegação, na qual estão representadas todas as possíveis seqüências de apresentação entre os objetos modelados. O modelo didático representado refere-se ao modelo instrucional da Figura 3.5, que trata dos Conceitos Básicos da programação OO. Considere, por exemplo, o estado *DD* Conceitos Básicos. De acordo com as propriedades de um estado *DD*, todos os subestados de Objeto, Classe, Método estão conectados entre si por transições implícitas que são responsáveis por estabelecer a navegação entre os estados. Desse modo, a partir de Objeto, por exemplo, pode-se atingir os outros estados Classe e Método e vice-versa.

A forma como esses estados *DD* foram implementados é mostrada na Figura 3.11. Os estados estão representados por links no *frame* esquerdo das transparências. Na Figura 3.11 (a) estão representados os links para os subestados do estado Linguagem de Programação representados no modelo didático. Esses links existem em todas as transparências, assim pode-se de qualquer ponto da apresentação ir para outros subestados. Na Figura 3.11 (b) está representado o estado *DD* Conceitos Básicos com os links para seus respectivos subestados Objeto, Classe, Método.

É importante destacar que as transparências implementadas foram desenvolvidas manualmente utilizando *LaTex*. No entanto, com a existência de ferramentas de apoio, tais transparências poderiam ter sido geradas automaticamente a partir dos modelos desenvolvidos. Esta é uma das atividades futuras a ser conduzida em decorrência deste trabalho.

Figura 3.10: Modelo Didático Integrado: Conceitos Básicos

Figura 3.11: Transparências representando o estados *DD* do Modelo Didático

3.5 Módulo Educacional Integrado: Aspectos Gerais da Implementação

Conforme discutido no capítulo anterior, módulos educacionais correspondem a unidades de estudo, compostas essencialmente por conteúdos teóricos integrados a atividades práticas e avaliações, cuja disponibilização aos aprendizes é apoiada por meio de recursos tecnológicos e computacionais (Barbosa, 2004).

Baseado nos modelos descritos na Seção 3.4, foi implementado módulo educacional integrado em transparências utilizando o *LaTeX*. O módulo integrado possui aproximadamente 250 transparências e é composto de dois módulos — um de fundamentos de programação OO e outro de teste de software — que foram integrados por meio de pontos de conexão em locais específicos de cada módulo. A determinação dos pontos de conexão entre os conceitos foi discutida na Seção 3.3.

Cada módulo é implementado como um conjunto de slides, documentos-texto, ferramentas educacionais e ferramentas de suporte específicas do domínio que foram integradas. No caso de teste de software, a ferramenta de teste – JaBUTi (*Java Bytecode Understanding and Testing*) (Vincenzi et al., 2003), que apóia análise de cobertura para testar programas e componentes Java – foi incorporada ao módulo. A incorporação de ferramentas educacionais ao módulo, neste caso específico de ferramentas de teste, é importante, pois as mesmas atuam como mecanismos

de apoio à condução das atividades instrucionais propostas, possibilitando a aplicação prática dos conceitos e demais informações teóricas associadas.

A integração entre os diversos componentes do módulo é apresentada na Figura 3.12. Na Figura 3.12 (a) é mostrada a transparência principal do módulo que contém os links para o módulo de programação OO e para o módulo de teste de software. A Figura 3.12 (b) contém a estrutura do módulo de programação OO. A Figura 3.12 (c) contém a estrutura do módulo de teste de software. A Figura 3.12 (d) mostra uma transparência que contém a integração com outros componentes como documentos-texto, ferramentas educacionais, ferramentas específicas do domínio. Por exemplo, nessa transparência tem-se a integração com um documento-texto de teste de software, com a ferramenta educacional *CoWeb* (Guzdial et al., 2002), e também com uma ferramenta específica de domínio, neste caso, a ferramenta de teste *JaBUTi* (Vincenzi et al., 2003).

Como mencionado anteriormente, os módulos de programação OO e teste de software foram integrados em pontos de conexão específicos, determinados durante a modelagem conceitual. Na implementação, essa integração ocorre por meio de links entre os dois módulos. A Figura 3.13 mostra um cenário de navegação que ilustra parte da integração implementada. Iniciando no slide principal do módulo, o usuário pode navegar através da estrutura dos módulos de fundamentos de Programação OO e de Teste de Software. Enquanto os conceitos específicos de programação são apresentados, links para as partes correspondentes em teste de software estão habilitados. Por exemplo, a Figura 3.13 (a) corresponde ao slide principal do módulo; links para o módulo de programação bem como para o de teste de software estão ativados. Considere a navegação através do módulo de programação, mais especificamente passando pelas informações relacionadas às estruturas de fluxo de controle (Figura 3.13(b)). Nesse ponto, pode-se explorar os conceitos relacionados ao teste estrutural, em especial os critérios de teste de fluxo de controle Todos-Arcos (Figura 3.13(c)). Nesse ponto, também é possível executar a ferramenta de teste *JaBUTi*, explorando na prática os conceitos teóricos aprendidos no módulo (Figura 3.13(d)).

A qualquer tempo pode-se parar a exploração do módulo e retornar para os conceitos de programação. De fato, a livre navegação do módulo de programação para o módulo de teste de software (e vice-versa) fornece flexibilidade para o instrutor para adequar os materiais didáticos de acordo com a performance dos estudantes em “tempo de execução”. Essa flexibilidade é possível porque o módulo foi implementado de acordo com especificação aberta, explorada no modelo *HMBS/Didático* da Figura 3.10.

É importante destacar que, caso exista uma ferramenta de apoio à edição, interpretação e execução de modelos didáticos segundo o *HMBS/Didático*, especificações “executáveis” também podem ser derivadas e utilizadas como apoio à geração automática de conteúdos educacionais, personalizados de acordo com interesses pedagógicos, perfil do usuário, características do módulo, entre outros aspectos. Além da geração automática de material, ferramentas de simulação também podem ser utilizadas como apoio à apresentação dos conteúdos desenvolvidos ou,

Slide Inicial do Módulo Integrado

Fundamentos de Programação e Teste de Software

Camila Kozlowski Della Corte
Ellen Francine Barbosa
Orientador: José Carlos Maldonado
camilla, francine, jcmaldonado@cmc.usp.br
Instituto de Ciências Matemáticas e de Computação — ICMC
Universidade de São Paulo — USP

Slide Inicial do Módulo de Programação

Fundamentos de Programação I

- Orientação a Objetos
- Conceitos Básicos
- Comandos Essenciais
- Estruturas de Fluxo de Controle
- Array
- Arquivo
- Tratamento de Exceção

Slide Inicial do Módulo de Teste de Software

Teste de Software

- Teste de Software OO
- Teste Software
- Terminologia
- Etapas de Teste
- Fases de Teste
- Técnicas de Teste

Slide Inicial do Módulo de Teste de Software OO

Teste de Software OO

- Análise dinâmica do produto de software
 - Processo de executar o software de modo controlado, observando seu comportamento em relação aos requisitos especificados
- Processo de executar um programa com a intenção de encontrar erros
 - O teste bem sucedido é aquele que consegue determinar casos de teste que evidenciam falhas no programa sendo analisado

Introdução ao Teste de Software*

Ana Carolina Maldonado
Ellen Francine Barbosa
Ana Marcela Rizzo Vitorini
Universidade de São Paulo — ICMC/USP
(camilla, francine, ana)@icmc.usp.br

Márcio Eduardo Delamaro
delamaro@fundacoe.org.br
Centro Universitário Eurípides de Marília — UNIVEM

Simone da Rocha Senger de Souza
senger@uepg.br
Universidade Estadual de Maringá — UEPG

Maria Ivo
jvo@fca.fee.unicamp.br
Universidade Estadual de Campinas — DCA/FEEC/UNICAMP

Tópicos de Engenharia de Software I (SCES829-3-1)

Prof. José Carlos Maldonado
Horário: Terça-feira (13:30 às 16:30)
09/05/2004

Referências e documentos

- Livro de Texto
- Material Complementar

Figura 3.12: Módulo Orientação a Objetos e Teste de Software

Figura 3.13: Cenário de Navegação através dos módulos de Programação e Teste de Software

ainda, na validação das especificações geradas e dos próprios conteúdos associados (Barbosa, 2004).

Por fim, observa-se que o módulo educacional integrado também vem sendo explorado sob a perspectiva de “material didático livre”. Neste caso, o módulo deverá possuir uma licença de uso e distribuição, sendo permitido a seus usuários realizar alterações nos conteúdos personalizados de modo a adequá-los a seus objetivos e necessidades de aprendizado (Barbosa, 2004).

3.6 Considerações Finais

Neste capítulo foi discutido como o módulo educacional integrado de programação OO e teste de software foi desenvolvido e implementado. Para o desenvolvimento desse material foi utilizado o processo padrão para desenvolvimento de módulos educacionais estabelecido por Barbosa (2004), com ênfase à abordagem *AIM-CID*. Os modelos conceituais foram construídos separadamente para o domínio de conhecimento de programação OO e teste de software e depois integrados num modelo único. Aspectos referentes à evolução da *AIM-CID*, também foram propostos e aplicados na construção do modelo conceitual integrado. Em seguida, foram construídos os modelos instrucionais e didáticos. A partir desses modelos, foram implementadas as transparências, constituindo assim o módulo educacional integrado.

No próximo capítulo é apresentada a ferramenta de submissão e avaliação de trabalhos práticos desenvolvida durante este trabalho — PROGTEST — que fornece suporte ao processo de ensino integrado de programação OO e teste de software.

ProgTest: Ambiente de Submissão e Avaliação de Trabalhos Práticos

4.1 Considerações Iniciais

Para fornecer suporte ao ensino integrado de fundamentos de programação e teste de software é necessário a disponibilização de um ambiente que possa promover uma avaliação o mais automática possível dos trabalhos dos alunos. É interessante que no curso introdutório onde os conceitos de fundamentos de programação são integrados com os conceitos de teste de software seja utilizado um ambiente de submissão e avaliação de trabalhos práticos. Nesse sentido, um dos objetivos deste trabalho é a construção de um ambiente baseado na *Web*, denominado PROGTEST, para submissão e avaliação automática de trabalhos práticos baseados em atividades de teste de software para auxiliar os processos de ensino e aprendizagem de programação e teste de software.

Neste capítulo são discutidos os aspectos de desenvolvimento e operacionais do ambiente PROGTEST desenvolvido. Na Seção 4.2 é apresentada a motivação para a construção desse ambiente. Na Seção 4.3 é discutida a estratégia adotada para a avaliação automática dos programas baseadas em atividades de teste. Na Seção 4.4 é detalhada a arquitetura do ambiente, baseada no MVC. A modelagem e os aspectos de implementação são discutidos nas Seções 4.5 e 4.6, respectivamente. Na Seção 4.7 são mostrados os aspectos operacionais do ambiente.

4.2 Motivação para o Ambiente PROGTEST

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 o desempenho 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 (Edwards, 2004). 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, o aluno pode saber como está o seu desempenho (Isong, 2001).

Como mostrado no Capítulo 2, alguns ambientes para submissão e avaliação automática de trabalhos práticos foram propostos no sentido de facilitar o ensino integrado de teste e de fundamentos de programação, como (Goldwasser, 2002; Isong, 2001; Jackson e Usher, 1997; Jones, 2001b; Korhonen et al., 2002; Saikkonen et al., 2001). 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. A ênfase desses ambientes está na saída dos programas produzidos pelos alunos, não sendo utilizadas ferramentas de teste de análise de cobertura para avaliar os trabalhos entregues.

A fim de apoiar o ensino de fundamentos de programação e de teste de software, desenvolveu-se um ambiente de submissão e avaliação automática de trabalhos práticos, denominado PROGTEST. Este ambiente é baseado na *Web* e avalia tanto os programas quanto os casos de teste fornecidos pelos alunos. Assim, são analisadas tanto a qualidade do programa quanto a atividade de teste realizada pelos estudantes. Para isso, ferramentas de teste foram integradas ao ambiente para permitir que seja realizada uma análise de cobertura baseada em critérios de teste. No caso da PROGTEST, a ferramenta de teste integrada foi a JaBUTi (Vincenzi, 2004). Outras ferramentas podem ser agregadas posteriormente.

4.3 Abordagem para Avaliação de Programas

O ambiente PROGTEST é um ambiente baseado na *Web*, na concepção de software livre, para submissão e avaliação automática de trabalhos práticos. Conforme discutido nos capítulos anteriores, o paradigma OO e a linguagem Java foram escolhidos para a elaboração do módulo

educacional de programação e teste de software. Dessa forma, para fornecer suporte ao ensino integrado desses domínios, o ambiente PROGTEST foi desenvolvido para aceitar programas escritos em Java. Os casos de teste que são submetidos pelos alunos devem estar no formato do JUnit(Massol e Husted, 2005) que é padrão dos casos de teste aceitos pela ferramenta de teste JaBUTi (Vincenzi, 2004), utilizada para realizar a análise de cobertura dos programas.

Na Figura 4.1 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 4.1: 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. A ferramenta de teste JaBUTi (Vincenzi, 2004) juntamente com o JUnit (Massol e Husted, 2005) servem 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.

O ambiente avalia automaticamente os programas dos alunos da seguinte forma. Para cada trabalho existe uma implementação de referência, o programa “oráculo”, e um conjunto de casos de teste que são utilizados para testar essa implementação de referência. O aluno submete ao sistema o código fonte do seu programa escrito em Java e os casos de teste escritos no JUnit. Após a submissão, o sistema compila o programa e os casos de teste do aluno e depois realiza as execuções trocadas utilizando a ferramenta de teste JaBUTi, realizando o teste estrutural, e também o framework JUnit para realização do teste funcional. Como mencionado anteriormente, as execuções ocorrem da seguinte maneira:

- Para cada trabalho:
 1. Executa-se o programa do professor (P_{Inst}) com os casos de teste do professor (T_{Inst}) — $P_{Inst} - T_{Inst}$;
- Para cada aluno:
 1. Executa-se o programa do aluno (P_{Al_i}) com os seus respectivos casos de teste (T_{Al_i}) — $P_{Al_i} - T_{Al_i}$;
 2. Executa-se o programa do professor (P_{Inst}) com os casos de teste do aluno (T_{Al_i}) — $P_{Inst} - T_{Al_i}$;
 3. Executa-se o programa do aluno (P_{Al_i}) com os casos de teste do professor (P_{Inst}) — $P_{Al_i} - T_{Inst}$.

Para cada uma dessas execuções a ferramenta JaBUTi realiza a análise de cobertura do código com os casos de teste utilizados e fornece os índices de cobertura nos critérios de teste por ela implementados, obtendo dessa forma os resultados do teste estrutural dos programas dos alunos. Durante essas execuções, os casos de teste também são executados no JUnit. Dessa forma, têm-se também os resultados do teste funcional dos programas dos alunos, uma vez que executado um caso de teste, a saída obtida é comparada com a saída esperada.

É importante destacar que o aluno durante a curso de programação OO e teste de software pode ter livre acesso à ferramenta de teste JaBUTi com todas as suas funcionalidades, sendo possível realizar o teste estrutural de seus programas. O aluno ainda pode realizar o teste estrutural por meio do ambiente PROGTEST, uma vez que o sistema gera relatórios com as

coberturas obtidas tanto com a realização do teste funcional no JUnit como do teste estrutural na ferramenta JaBUTi. Com isso, eles podem melhorar sua nota geral, visto que é possível melhorar o programa e o conjunto de casos de testes e submeter novamente ao sistema até a data limite de aceitação do trabalho, objetivando um conjunto de casos de teste adequado aos critérios estruturais.

Como exemplo, suponha que um trabalho foi submetido e que suas execuções foram realizadas no JUnit e na JaBUTi. Na Tabela 4.1 é ilustrado um relatório do JUnit onde são mostrados quais dados são armazenados pelo sistema quando o programa do aluno é executado com seus respectivos casos de teste no JUnit. De acordo com o relatório, é possível saber quantos casos de teste o aluno desenvolveu, quantos erros e quantas falhas os casos de teste detectaram e como consequência, qual é a taxa de sucesso do teste funcional realizado pelo aluno.

Tabela 4.1: Relatório do JUnit (Teste Funcional)

JUNIT	$P_{Al_i} - T_{Al_i}$
Nº Casos de Teste	20
Erros	0
Falhas	1
Taxa de Sucesso	95%

Considere agora a Tabela 4.2, a qual mostra um relatório de cobertura do teste estrutural contendo uma síntese dos valores de índices de cobertura que a JaBUTi fornece ao sistema. Nesse caso, têm-se valores de cobertura referentes a três critérios de teste: Todos-Nós, Todos-Arcos e Todos-DefUso. Para cada execução realizada na JaBUTi são encontrados valores para cada um desses critérios, como é mostrado na Tabela 4.2. Por exemplo, quando executou-se o programa do aluno (P_{Al_i}) com os casos de teste do professor (T_{Inst}), obteve-se a cobertura de 95 % no critério Todos-Nós.

Tabela 4.2: Relatório de Cobertura obtido com a execução na JaBUTi (Teste Estrutural)

	TODOS-NÓS	TODOS-ARCOS	TODOS-DEFUSOS
$P_{Al_i} - T_{Al_i}$	98%	90%	85%
$P_{Inst} - T_{Inst}$	92%	89%	85%
$P_{Al_i} - T_{Inst}$	95%	92%	90%
$P_{Inst} - T_{Al_i}$	90%	88%	82%

Com os dados obtidos com a execução das ferramentas de teste parte-se para a fase de análise desses dados e a geração de sugestão de nota ao trabalho do aluno.

Um dos principais problemas com relação à técnica de teste estrutural refere-se à impossibilidade, em geral, de se determinar automaticamente se um caminho é executável ou não

(Vergílio et al., 1993). Ou seja, não existe um algoritmo que dado um caminho completo qualquer, decida se o caminho é executável e forneça o conjunto de valores que causam a execução desse caminho. Assim, é preciso a intervenção do testador para determinar quais são os caminhos não executáveis para o programa que está sendo testado (Barbosa et al., 2000).

A existência de caminhos não executáveis durante a avaliação dos trabalhos dos alunos não é tratada pelo ambiente PROGTEST. O fato de não atingir 100% de cobertura em um determinado critério de teste tanto nos programas dos alunos quanto na implementação de referência do professor pode indicar: (1) o conjunto de casos de teste não é de boa qualidade, não conseguindo cobrir todo o código; ou (2) a implementação desenvolvida contém caminhos não executáveis.

A PROGTEST, na análise dos dados para a geração da nota dos trabalhos dos alunos, considera o seguinte:

A cobertura de referência esperada da execução da implementação do professor com seus casos de teste ($P_{Inst} - T_{Inst}$) é de 100%. No entanto, se a implementação de referência do professor não atingir 100% considera-se que a parte não coberta, ou seja, os caminhos não executados é constituída de caminhos não executáveis. Assim, quando executam-se os casos de teste desenvolvidos pelos alunos na implementação do professor ($P_{Inst} - T_{Al_i}$) é necessário fazer uma normalização no resultado da análise de cobertura dessa execução. A normalização dessas coberturas é feita por meio da seguinte fórmula:

- $CoberturaNormalizada = (CoberturaProfTrocada * 100\%) / CoberturaProfessor$, onde a $CoberturaProfessor$ corresponde a cobertura da execução de $P_{Inst} - T_{Inst}$ e $CoberturaProfTrocada$ corresponde a cobertura da execução de $P_{Inst} - T_{Al_i}$.
- Se $CoberturaNormalizada > 100\%$, então $CoberturaNormalizada = 100\%$. Observa-se que se algum caminho não executado pelos casos de teste do professor (T_{Inst}) for executado pelos casos de teste do aluno (T_{Al_i}), a cobertura normalizada será maior que 100%. Assim, considera-se que a cobertura normalizada é 100%.

Com relação à implementação do aluno, a cobertura de referência esperada da execução do programa do aluno com seus casos de teste ($P_{Al_i} - T_{Al_i}$) também é de 100%. Se a cobertura de $P_{Al_i} - T_{Al_i}$ não atingir 100% também considera-se que a parte não coberta, que contém os caminhos não executados, é composta de caminhos não executáveis. Nesse caso, ao se executar o programa do aluno com os casos de teste do professor ($P_{Al_i} - T_{Inst}$) deve-se fazer os seguintes ajustes nas coberturas:

- Se a cobertura de $P_{Al_i} - T_{Al_i}$ for maior ou igual a cobertura de $P_{Al_i} - T_{Inst}$ atribui-se 100% para a cobertura obtida na execução de $P_{Al_i} - T_{Inst}$. Neste caso, os casos de teste do aluno foram eficientes para testar seu próprio programa.
- Se a cobertura de $P_{Al_i} - T_{Al_i}$ for menor que a cobertura de $P_{Al_i} - T_{Inst}$, os casos de teste do aluno não foram tão eficientes e deve ser atribuída uma penalidade à cobertura

obtida. Assim, a cobertura obtida na execução de $P_{Al_i} - T_{Inst}$ deve ser ajustada seguindo a seguinte fórmula:

$CoberturaPenalizada = 100\% - (CoberturaAlunoTrocada - CoberturaAluno)$, onde a $CoberturaAlunoTrocada$ corresponde a cobertura da execução de $P_{Al_i} - T_{Inst}$ e a $CoberturaAluno$ corresponde a cobertura da execução de $P_{Al_i} - T_{Al_i}$.

Aplicando-se essas considerações nas execuções da Tabela 4.2, os valores das coberturas são calculados novamente (Tabela 4.3).

Tabela 4.3: Valores Ajustados da Análise de Cobertura

	TODOS-NÓS	TODOS-ARCOS	TODOS-DEFUSOS
$P_{Al_i} - T_{Al_i}$	98%	90%	85%
$P_{Inst} - T_{Inst}$	92%	89%	85%
$P_{Al_i} - T_{Inst}$	100%	99%	93%
$P_{Inst} - T_{Al_i}$	98%	99%	96%

Com base nesses valores da análise de cobertura ajustados, é realizada a avaliação. Para a geração das notas do trabalho é necessário o estabelecimento de alguns critérios de avaliação pelo professor. Na criação de um trabalho no sistema, o professor pode estabelecer pesos ao critérios de teste Todos-Nós, Todos-Arcos, Todos-DefUso de acordo com o andamento do curso, o perfil da turma, etc. Com os pesos definidos pode-se estabelecer uma cobertura geral para cada execução utilizando a seguinte fórmula:

$CoberturaGeral = (Peso1 * (TodosNos) + Peso2 * (TodosArcos) + Peso3 * (TodosDefUso)) / (Peso1 + Peso2 + Peso3)$.

Como exemplo, na Tabela 4.4 é mostrado um relatório onde se chegou à cobertura geral obtida com a realização do teste estrutural na ferramenta JaBUTi. Os valores que se encontram nessa tabela correspondem aos valores normalizados da Tabela 4.3, aplicados à fórmula criada para cobertura geral com os seguintes pesos: $CoberturaGeral = (5 * (TodosNos) + 3 * (TodosArcos) + 2 * (TodosDefUso)) / (10)$. Dessa forma, obteve-se os valores de cobertura do $P_{Al_i} - T_{Al_i}$, $P_{Al_i} - T_{Inst}$ que indica a qualidade do programa desenvolvido pelo aluno, e do $P_{Inst} - T_{Al_i}$ que indica a qualidade do conjunto de casos de teste desenvolvido pelo aluno.

Tabela 4.4: Relatório de Cobertura Geral (Execuções Trocadas)

	Cobertura Geral
$P_{Al_i} - T_{Al_i}$	93%
$P_{Al_i} - T_{Inst}$ (Qualidade do Programa)	98,3%
$P_{Inst} - T_{Al_i}$ (Qualidade Conjunto de Casos de Teste)	97,9%

Um outro critério de avaliação que é definido pelo professor no momento da criação de um determinado trabalho é a determinação dos pesos de cada execução trocada ($P_{Al_i} - T_{Al_i}$, $P_{Al_i} - T_{Inst}$, $P_{Inst} - T_{Al_i}$) e do teste funcional realizado pelo aluno. Com esses pesos determinados é sugerida uma fórmula onde uma nota geral pode ser fornecida ao trabalho. A fórmula para a geração da nota geral do trabalho é a seguinte:

$$NotaGeralSugerida = (Peso1 * (TesteFuncional) + Peso2 * (P_{Al_i} - T_{Al_i}) + Peso3 * (P_{Al_i} - T_{Inst}) + Peso4 * (P_{Inst} - T_{Al_i})) / (Peso1 + Peso2 + Peso3 + Peso4)$$

Aplicando a fórmula da nota geral nos valores de cobertura encontrados para as execuções trocadas da Tabela 4.4 chegou-se a uma nota de 96,8. Esse nota foi obtida aplicando-se os seguintes pesos aos valores da Tabela 4.4: $NotaGeralSugerida = (1 * (TesteFuncional) + 3 * (P_{Al_i} - T_{Al_i}) + 3 * (P_{Al_i} - T_{Inst}) + 3 * (P_{Inst} - T_{Al_i})) / (10)$.

Destaca-se também que a nota que o sistema fornece é somente uma sugestão, cabendo ao professor decidir se utiliza a mesma como nota real do trabalho. O professor pode alterar essa nota sugerida pelo sistema assim que desejar.

Nesta seção foi detalhado em alto nível a forma como o ambiente PROGTEST realiza a avaliação de programas. Na seção seguinte é discutida a arquitetura do ambiente, mostrando em detalhes os módulos que o compõem.

4.4 Arquitetura do Ambiente PROGTEST

Como o ambiente PROGTEST é um sistema para *Web*, sua arquitetura foi elaborada seguindo o padrão de projeto de arquitetura *Model-View-Controller* (MVC) (Sun Microsystems, 2006b). Na Figura 4.2 está representada a arquitetura MVC para o ambiente PROGTEST. Segundo esse padrão, a arquitetura do sistema é dividida em três camadas: visão, controle e modelo. A camada de visão é a camada de apresentação da aplicação. A camada de controle é a camada intermediária entre a visão e o modelo da aplicação e age de forma controladora, recebendo as requisições da camada de visão e passando para a camada de modelo, onde é efetivamente realizado o processamento e o resultado é retornado para a camada de visão. Portanto, na camada de modelo ou lógica do negócio é onde se encontram as funcionalidades da aplicação. Essa camada é responsável pela comunicação com o banco de dados e integração com outros sistemas (Sun Microsystems, 2006b). No caso da PROGTEST a camada de modelo comunica-se com ferramentas de teste externas à aplicação.

A camada de modelo do ambiente PROGTEST é detalhada na Figura 4.3 onde são mostrados os módulos que compõem as funcionalidades do ambiente PROGTEST. O ambiente possui basicamente três módulos principais: gerenciamento de cursos, submissão e avaliação.

- **Módulo gerenciamento de cursos:**

O módulo de gerenciamento de cursos é o responsável pelo gerenciamento de usuários (alunos, professores), turmas e trabalhos envolvidos em um determinado curso. Com

Figura 4.2: Arquitetura *Model-View-Controller* MVC do ambiente PROGTEST

Figura 4.3: Camada de Modelo do ambiente PROGTEST

relação aos usuários, esse módulo realiza o cadastramento e a autenticação de usuários no sistema.

Neste módulo também é realizado o gerenciamento de turmas do sistema. O sistema permite a criação, edição, exclusão e visualização de turmas. Dentro de uma determinada turma o sistema permite a inclusão de alunos e a criação, edição, exclusão e visualização de trabalhos. O sistema ainda realiza o controle de notas de uma determinada turma.

É importante ressaltar que o módulo de gerenciamento comunica-se com o banco de dados no qual todas as informações referentes aos usuários, turmas e alunos estão armazenadas.

- **Módulo submissão:** O módulo de submissão cuida do controle da submissão dos trabalhos dos alunos e também dos oráculos pelos professores. Nesse módulo é criada a estrutura de diretórios onde serão armazenados os arquivos submetidos pelos alunos e oráculos submetidos pelos professores. É realizado também o controle de datas do sistema que permite o controle da data de início e da data limite de entrega de um trabalho. A submissão de um trabalho se inicia na data determinada pelo professor no momento de criação do trabalho. Cada trabalho também tem definido no momento de sua criação uma data limite para entrega do mesmo. No entanto, o sistema permite que os alunos continuem submetendo seus trabalhos mesmo já tendo passado a data limite de entrega.
- **Módulo avaliação:** Na Seção 4.3 foi discutido com detalhes como a avaliação de trabalhos é realizada. O módulo de avaliação é o responsável pelo controle das avaliações dos trabalhos, sendo composto de quatro outros módulos:
 - **Critérios de Avaliação:** O módulo de critérios de avaliação é o responsável pela determinação dos critérios de avaliação e os pesos que serão atribuídos aos critérios de teste existentes nas ferramentas de teste, entre outros aspectos. Os pesos são utilizados na formulação da “nota” que o sistema irá sugerir para o trabalho do aluno, como exemplificado anteriormente.
 - **Primitivas de Execução:** O módulo de primitivas de execução é responsável por compilar os programas e os casos de teste submetidos aos alunos e depois executá-los em uma ferramenta de teste para obter os resultados. É nesse módulo que a integração com a ferramenta de teste JaBUTi e com *framework* JUnit é realizada. A comunicação com a ferramenta de teste acontece via arquivo XML. Após a execução dos programas dos alunos na ferramenta de teste, os resultados obtidos são armazenados em outro arquivo XML que é utilizado no módulo de análise dos dados.
 - **Análise dos Dados:** Nesse módulo, é realizada uma análise dos resultados obtidos pelo módulo de primitivas de execução levando em consideração os critérios de avaliação definidos no módulo critérios de avaliação. É nesse módulo que é gerada a nota do trabalho sugerida ao aluno.
 - **Relatórios:** O módulo de relatórios é o responsável por gerar os relatórios de avaliação, tendo como base os resultados da análise dos dados.

A seguir são apresentados aspectos de modelagem e implementação referentes ao ambiente PROGEST.

4.5 Aspectos de Modelagem

A modelagem do sistema foi realizada utilizando-se parte da abordagem de Conallen (Conallen, 2003), chamada WAE (*Web Application Extension for UML*). Nessa abordagem, a notação UML é estendida com semânticas e restrições adicionais para permitir a modelagem de elementos da arquitetura específica da *Web* como parte do modelo do sistema (Conallen, 2003).

Primeiramente, foi realizada a especificação de requisitos, onde foram identificadas as características dos usuários e requisitos funcionais. Durante o processo de levantamento de requisitos, foram elaborados os diagramas de casos de uso e os diagramas de sequência. Estes diagramas e todos os outros mencionados aqui encontram-se num documento de trabalho que contém a descrição da modelagem e implementação do ambiente PROGTST (Corte et al., 2006b).

Nessa fase, foi elaborado um protótipo utilizado para refinar os requisitos. Esse protótipo foi utilizado inicialmente na análise dos requisitos e não foi descartado, sendo evoluído até se chegar ao produto final. Desta forma, pode-se dizer que o modelo de processo de software utilizado no desenvolvimento deste ambiente é a prototipação evolutiva (Pressman, 2005).

Um dos modelos que compõem a abordagem WAE é o mapa de navegação de nível superior para uma aplicação *Web*. No mapa de navegação é fornecida uma visão geral da estrutura de navegação da aplicação, não sendo mostrados no modelo detalhes das páginas. O mapa de navegação é um diagrama de classes adaptado. A abordagem WAE cria dois novos estereótipos: «**screen**» para representar as páginas e «**input form**» para representar os formulários. Os formulários podem ser modelados com um relacionamento entre uma página e outra ou como uma classe contida (Conallen, 2003).

O ambiente PROGTST possui uma visão referente ao professor e outra referente aos alunos. Nas Figuras 4.4 e 4.5 são mostrados respectivamente parte do mapa de navegação do ambiente PROGTST para as visões que o professor e o aluno, respectivamente, têm do sistema.

A Figura 4.4, contém parte do mapa de navegação da visão que o professor tem do sistema. Exemplificando, o professor, a partir da página inicial do ambiente, pode ir para a página de cadastro de usuários ou pode, caso já seja cadastrado, preencher o formulário e realizar a autenticação no sistema. Se a autenticação ocorrer com sucesso ele é direcionado para a página que contém a lista de turma a qual ele está vinculado. A partir dessa página, é possível navegar para diversas páginas como por exemplo, edição de uma turma, criação de uma nova turma, visualização dos dados de uma turma e assim por diante. Assim, a navegação da aplicação pode ser visualizada nesse modelo.

Na Figura 4.5 é mostrada parte do mapa de navegação na visão do aluno. Partindo da página inicial, o aluno efetua sua autenticação no sistema e visualiza a lista de turmas em que está matriculado. A partir da página contendo a lista de turmas, pode-se navegar e visualizar os dados de cada turma e ir para a página contendo a lista de trabalhos encerrados ou para a página contendo a lista de trabalhos abertos. Estando na página que contém a lista de trabalhos abertos,

Figura 4.4: Parte do Mapa de Navegação - Visão Professor

pode-se navegar até chegar a página onde ocorre a submissão dos trabalhos e submeter seu trabalho, indo posteriormente para a página que contém os resultados da avaliação do trabalho que acabou de ser submetido.

Com relação às funcionalidades do sistema, a parte principal do ambiente PROGTEST é a avaliação dos trabalhos submetidos pelo alunos. Na Figura 4.6 é ilustrado como essa avaliação é realizada pelo sistema por meio do diagrama de seqüência da parte relativa à avaliação de trabalhos. O aluno submete ao sistema seu programa e seu conjunto de casos de teste. O sistema realiza o *upload* dos arquivos dos alunos armazenando os mesmos em uma pasta especificamente criada pelo sistema para isso. Após ter armazenado os arquivos, um arquivo XML de configuração é gerado pelo sistema e é enviado à classe avaliador que realizará a avaliação dos trabalhos. Esse arquivo XML de configuração contém informações relativas à classe que será testada e aos casos de teste utilizados, ao caminho onde esses arquivos estão armazenados e, ainda, à qual ferramenta de teste que será utilizada para testar os programas. A classe avaliador

Figura 4.5: Parte do Mapa de Navegação - Visão Aluno

lê esse arquivo de configuração e solicita a compilação dos arquivos e depois executa-os em uma ferramenta de teste (neste caso, na ferramenta JaBUTi). Depois os dados obtidos pela ferramenta de teste são analisados de acordo com os critérios de avaliação previamente definidos e um relatório de avaliação é gerado e mostrado ao alunos.

Identificadas as funcionalidades do sistema foi realizado o projeto do banco de dados. O modelo relacional construído é mostrado na Figura 4.7.

Em seguida, foi desenvolvido o diagrama de classes do sistema, ilustrado na Figura 4.8. De acordo com o modelo, uma disciplina pode ter zero ou mais turmas. Cada turma deve ter um ou mais alunos matriculados e pode ter um ou mais professores lecionando na turma. Uma turma pode ter zero ou mais trabalhos vinculados a ela. Cada trabalho é controlado por um gerenciador de datas que indica quando o trabalho está aberto para submissão e quando as submissões estão encerradas. Para cada trabalho é definido um critério de avaliação. Cada trabalho pode ser submetido uma ou várias vezes pelo usuário, que pode ser tanto professor ou o aluno. Para cada trabalho é gerado um arquivo XML de configuração que é utilizado pela classe avaliador

Figura 4.6: Diagrama de Seqüência - Avaliação de Trabalhos

para realizar a avaliação do trabalho. A classe avaliador lê o arquivo XML de configuração, compila os arquivos, executa os mesmos em uma ferramenta de teste e depois gera os relatórios de avaliação baseados nos critérios de avaliação definidos. Para cada classe avaliador existe um compilador associado e pode existir uma ou mais ferramentas de teste associadas. Para cada classe avaliador pode existir um ou mais relatórios de avaliação.

A partir dessa modelagem o sistema foi implementado. A seguir são discutidos algumas questões relativas à implementação do ambiente.

Figura 4.7: Modelo Relacional

Figura 4.8: Diagrama de Classes

4.6 Aspectos de Implementação

Conforme mencionado na Seção 4.4, o ambiente foi implementado seguindo a arquitetura do MVC (*Model-View-Controller*). Para facilitar a implementação dessa arquitetura utilizou-se um *framework* desenvolvido pela Sun que une as camadas de apresentação e controle do MVC, chamado *Java Server Faces* (JSF) (Sun Microsystems, 2006a). A principal vantagem da utilização do JSF é a facilidade com que ele trata a separação entre a lógica e apresentação da aplicação. Desta forma, optou-se por utilizar o JSF para o desenvolvimento do ambiente PROGTEST. A implementação da camada de modelo foi realizada em Java e a apresentação e controle foram realizadas utilizando JSF.

O SGBD (Sistema Gerenciador de Banco de Dados) utilizado foi o MySQL (MySQL, 2006). O servidor de aplicações *Web* utilizado foi o Tomcat que é um container *Web* livre e de código aberto desenvolvido dentro do projeto Apache Jakarta e oficialmente endossado pela Sun como a Implementação de Referência (RI) para as tecnologias Java Servlet e JavaServer Pages (JSP) (The Apache Jakarta Project, 2006). É importante ressaltar que todas as linguagens e ferramentas que foram utilizadas na implementação deste ambiente estão dentro do contexto de software livre.

A camada de modelo do ambiente PROGTEST foi implementada completamente em Java como mencionado anteriormente. Cada módulo contém dezenas de classes sendo que o total de classes existentes nessa camada é de 72, como mostra a Tabela 4.5. Essa tabela mostra o esforço da implementação da camada de modelo do ambiente PROGTEST para cada módulo — gerenciamento de cursos, submissão e avaliação — em termos do número de classes e do número de linhas de código (LOC). Ao todo, o conjunto completo de classes que implementam essa camada correspondem a 7259 LOC, sendo que somente o módulo de avaliação possui 3211 LOC. É importante ressaltar que esses dados referem-se somente a implementação da lógica da aplicação. A camada de apresentação e controle implementadas em JSF não foram incluídos nesses dados.

Módulos	Nº Classes	Nº LOC
Gerenciamento de Cursos	32	2932
Submissão	12	1116
Avaliação	28	3211
Total	72	7259

Tabela 4.5: Esforço da implementação da camada de modelo do ambiente PROGTEST

Existem vários sistemas que tratam especificamente do gerenciamento de cursos online, como por exemplo, Panda (IME-USP, 2006), COL (LARC - USP, 2006). Dessa forma, o módulo de gerenciamento de cursos do ambiente PROGTEST é simples e foi criado para fornecer suporte para a submissão e avaliação automática de programas.

O módulo de submissão e avaliação do ambiente PROGTTEST foi implementado de forma a permitir sua integração com outros sistemas como por exemplo, Panda ou COL. O módulo avaliação desse ambiente está implementado em Java e se comunica com o restante do sistema através de arquivos de configuração que foram previamente definidos em XML. Dessa forma, o módulo de avaliação tem como entrada arquivos de configuração XML e como saída é gerado um relatório em XML com o resultado da avaliação, o qual é apresentado aos alunos. Com isso, esse módulo de avaliação torna-se independente, podendo ser integrado a outros sistemas que realizam gerenciamento de cursos online.

Um arquivo XML de configuração que serve de entrada para o módulo de avaliação foi definido. O DTD (*Document Type Definition*) desse arquivo XML é mostrado na Listagem 4.1, e nele estão definidas as seguintes informações:

Listagem 4.1: DTD de criação do arquivo XML de configuração

```
<!ELEMENT conf (source,base_class,test_case,tool+)>

<!ELEMENT source (path_src,path_build)>
<!ELEMENT path_src (#PCDATA)>
<!ELEMENT path_build (#PCDATA)>

<!ELEMENT base_class (name_class,path_class)>
<!ELEMENT name_class (#PCDATA)>
<!ELEMENT path_class (#PCDATA)>

<!ELEMENT test_case (name_test,path_test)>
<!ELEMENT name_test (#PCDATA)>
<!ELEMENT path_test (#PCDATA)>

<!ELEMENT tool (path_tool,name_project,test_report)>
<!ELEMENT path_tool (#PCDATA)>
<!ELEMENT name_project (#PCDATA)>
<!ELEMENT test_report (#PCDATA)>

<!ATTLIST tool name CDATA #REQUIRED>
<!ATTLIST test_report type CDATA #REQUIRED>
```

- O local onde estão armazenados os códigos fontes dos programas em Java para serem compilados e o local onde os arquivos compilados serão armazenados.
- O nome da classe base que será testada e onde o local onde ela está armazenada.
- O nome do arquivo com os casos de teste que serão utilizados para testar a classe base e o local onde este arquivo se encontra.
- O nome da ferramenta de teste que será utilizada juntamente com o nome do projeto de teste que será fornecido para ferramenta bem como o local onde o projeto de teste será armazenado. É também definido o tipo de relatório que será gerado bem como o nome do arquivo desse relatório.

Conforme mencionado anteriormente, têm-se as seguintes execuções trocadas: (1) o programa do aluno (P_{Al_i}) com os seus respectivos casos de teste (T_{Al_i}); (2) o programa do professor (P_{Inst}) com os casos de teste do professor (T_{Inst}); (3) o programa do professor (P_{Inst}) com os casos de teste do aluno (T_{Al_i}); (4) o se o programa do aluno (P_{Al_i}) com os casos de teste do professor (P_{Inst}). Para cada uma dessas execuções trocadas é gerado um arquivo XML de configuração e executado no módulo de avaliação.

A integração com ferramentas de teste, mas especificamente a JaBUTi, ocorre no módulo de avaliação. A versão atual da ferramenta de teste JaBUTi funciona por meio de sua interface gráfica. Para que o ambiente PROGTEST pudesse ser integrado com a ferramenta JaBUTi foi necessário a implementação de scripts para que a ferramenta executasse através de linhas de comando. Dessa forma, foram implementados os seguintes scripts: (1) criação do projeto de teste; (2) importação dos casos de teste no JUnit; e (3) geração dos relatórios de cobertura. Esses scripts encontram-se no Apêndice A. É importante ressaltar que uma versão *Web* da JaBUTi está sendo desenvolvida no contexto do trabalho de Ferrari (2005). Com sua disponibilização via *Web*, a idéia é que a mesma também possa ser utilizada como mecanismo de apoio ao aprendizado e revisão de conceitos de teste por alunos de outras instituições e por profissionais da indústria interessados na área.

Na próxima seção é brevemente ilustrado o funcionamento da PROGTEST.

4.7 Aspectos Operacionais

A PROGTEST é um ambiente de submissão e avaliação de trabalhos práticos. A seguir é mostrado quais são os aspectos operacionais da mesma. É importante destacar que um manual do usuário do ambiente encontra-se em elaboração (Corte et al., 2006a).

A PROGTEST possui dois tipos de usuários: professor e aluno. Cada um desses usuários tem uma visão diferente do sistema, mas com algumas funcionalidades em comum. As funcionalidades que fazem parte de ambas as visões são aquelas referentes ao cadastro de usuários e a autenticação no sistema.

A tela inicial do ambiente PROGTEST é mostrada na Figura 4.9. É nessa página que os usuários realizam a autenticação no sistema. O identificação dos usuários no sistema é feita usando o email cadastrado. Assim que o usuário realiza a autenticação, ele é direcionado para a página inicial referente à sua visão do sistema.

Na Figura 4.10 é mostrada a tela de cadastro de usuários. Nela os usuários entram com seus dados que são armazenados no banco de dados. Cabe aos professores e alunos o cadastro no sistema, sendo que para um aluno cadastrar-se no sistema é necessário que o professor já tenha criado uma turma para uma determinada disciplina.

Figura 4.9: PROGTEST - Tela Inicial **Figura 4.10:** PROGTEST - Cadastro de Usuários

Visão Professor

O professor pode realizar todo o gerenciamento das turmas e dos trabalhos vinculados à ela. Assim que o professor realiza a autenticação no sistema é direcionado para uma página que contém as listas de turmas às quais ele está vinculado (Figura 4.11). Nesse ponto do sistema, o professor pode editar, excluir ou visualizar os dados de uma determinada turma bem como

criar uma nova turma. Como mostra a Figura 4.11, existe uma tabela com as lista das turmas e navegando por cada linha da tabela o professor pode visualizar os dados da turma. Entre eles, pode-se visualizar os alunos matriculados, os trabalhos criados e o quadro de notas da turma.

Figura 4.11: PROGTEST - Lista de Turma Professor

Na Figura 4.12 é mostrada a página que contém os dados da turma. No lado esquerdo da página têm-se links para visualizar os alunos matriculados e o quadro geral de notas da turma. A parte principal da página contém a lista de trabalhos já criados para essa turma. Assim, pode-se editar, excluir, ver as notas da turma para um determinado trabalho já existente. A partir dessa página, pode-se ir para a página de cadastro de novos trabalhos.

A página de cadastro de novos trabalhos é mostrada, na Figura 4.13. Para o cadastro de novos trabalhos é necessário que o professor entre com o título do trabalho, uma breve descrição do que se trata o trabalho, a data inicial que o sistema irá abrir as submissões aos alunos e o data limite do término do trabalho. Cabe ainda ao professor, no momento da criação de um novo trabalho, estabelecer os critérios de avaliação dos trabalhos definindo os pesos que devem ser aplicados aos critérios de teste. Devem ainda ser definidos os pesos que vão gerar a nota sugerida pelo sistema, conforme explicado na Seção 4.3. Esses pesos correspondem ao teste funcional realizado pelo aluno e às execuções. O cadastro dessas informações é mostrado na página da Figura 4.13(a).

Cabe ainda ao professor, durante o processo de criação do trabalho, submeter ao sistema a implementação de referência do programa e dos casos de testes referentes ao trabalho. Isso é mostrado na Figura 4.13(b) onde tem-se a página que ilustra a submissão da implementação de referência do trabalho. No momento da criação do trabalho o sistema cria a estrutura de diretó-

Figura 4.12: PROGTEST - Lista de Trabalho Professor

rios onde serão armazenados os trabalhos submetidos pelos alunos e armazena as informações referentes ao trabalho no banco de dados.

Conforme discutido anteriormente, é possível acessar o quadro geral de notas da turma (Figura 4.14). Nesse quadro de notas é possível visualizar qual foi a nota de cada aluno em todos os trabalhos da turma bem com a média geral atingida por cada aluno. Em cada nota existe um link que mostra o histórico da entrega do trabalho e do desempenho de cada aluno em um determinado trabalho. Essa página com o histórico do trabalho do aluno é ilustrada na Figura 4.15.

Existem fatores externos ao sistema que podem influenciar a nota do trabalho de um aluno. Dessa forma, o professor pode alterar a nota do trabalho sugerida pelo sistema e ainda inserir um comentário a respeito do trabalho que será visualizado pelo aluno. Essa modificação das notas do trabalho de um aluno é realizada na página que mostra o histórico do aluno ilustrada na Figura 4.15. Ainda nessa página, o professor pode fazer o *download* dos arquivos que os alunos submeteram ao sistema caso seja necessário fazer uma avaliação manual. O professor também tem acesso ao relatório de avaliação do trabalho do aluno, explicado adiante.

Visão Aluno

Ao efetuar a autenticação no sistema, o aluno pode visualizar as turmas nas quais ele está matriculado, tendo acesso à lista de trabalhos criados para a turma. Na Figura 4.16 é ilustrada a lista de trabalhos de cada turma que os alunos têm acesso. Com relação aos trabalhos, existem

The screenshot shows a web browser window titled 'ProgTest - Microsoft Internet Explorer'. The address bar displays 'http://localhost:8080/progtestAtual/pages/principal/professor/template/trabTemplate.jsf'. The page features the 'PROGTEST' logo with a graduation cap icon. Below the logo is a welcome message 'Bem vindo!'. The main content area is titled 'ADICIONAR TRABALHO' and contains a form with the following fields: 'Titulo:' (text input), 'Descrição:' (text area), 'Data Inicial:' (date picker set to 22/02/2006), and 'Data Final:' (date picker set to 22/02/2006). Below these are two sections: 'DEFINIÇÃO DOS PESOS: CRITÉRIOS DE TESTE' with 'Todos-Nos', 'Todos-Arcos', and 'Todos-DefUse' inputs; and 'DEFINIÇÃO DOS PESOS: CRITÉRIOS DE AVALIAÇÃO' with 'Teste Funcional', 'PA - TINST', 'PA - TA', and 'PINST - TA' inputs. At the bottom of the form are 'Continuar', 'Limpar', and 'Cancelar' buttons.

(a)

The screenshot shows the same web browser window, but the page is titled 'ADICIONAR TRABALHO: SUBMISSÃO ORÁCULO'. The form contains the following fields: 'Classe Base' (text input), 'Programa' (text input with a 'Procurar...' button), 'Caso de Teste' (text input with a 'Procurar...' button), and 'Casos de Teste' (text input with a 'Procurar...' button). At the bottom of the form are 'Limpar' and 'Upload' buttons.

(b)

Figura 4.13: PROGTEST - Cadastro de Trabalhos

trabalhos fechados e trabalhos abertos para submissão. Os trabalhos fechados são aqueles cujo prazo de submissão já se encerrou e os abertos são que estão com submissões em andamento. No entanto, o sistema permite que mesmo um trabalho fechado, cuja data de entrega já tenha

PROGTEST

Bem vindo!

QUADRO GERAL DE NOTAS DA TURMA

Nome	Trabalho 2	Trabalho 3	Trabalho 4	Trabalho 5	Trabalho 6	Trabalho 7	Média
Pedro Jose	9.2	8.2	8.5	8.9	7.2	-	7.00
Ana Paula	7.0	8.0	7.9	7.0	9.0	-	6.48
Joao Paulo	8.7	7.7	6.7	8.7	7.7	-	6.58
Eduardo	7.5	5.5	8.5	6.5	7.5	-	5.92
Eduarda	8.5	8.5	4.5	9.5	6.5	-	6.25
Alessandro	9.5	7.5	9.5	8.5	8.5	-	7.25
Alexandre	7.7	8.7	7.5	7.7	7.7	-	6.55
Flavio	8.9	7.9	6.9	8.9	9.9	-	7.08
Luz Henrique	9.1	8.1	9.1	7.1	7.7	-	6.85
Renata	6.4	8.4	8.1	7.4	7.0	-	6.22
Marcio	7.8	6.8	7.8	7.8	8.8	-	6.50
Rodrigo	8.4	6.4	7.8	9.4	8.9	-	6.82

Figura 4.14: PROGTEST - Quadro Geral Notas da Turma

PROGTEST

Bem vindo!

HISTÓRICO AVALIAÇÃO TRABALHO

Aluno: Pedro José Data Submissão: 2006-02-09
 Trabalho: Trabalho 2 Hora Submissão: 11:30:05
 Descrição: Implementar árvore Trabalho Entregue no Prazo

Relatório Detalhado da Avaliação do Trabalho:

Arquivos Trabalho Submetidos:

Modificar Nota*:

Comentários:

* A nota sugerida pelo sistema é 89.9

Figura 4.15: PROGTEST - Histórico e Modificação de Notas do Aluno

passado do limite, que os alunos continuem submetendo seus trabalhos. Nesse caso, o aluno recebe uma mensagem dizendo que seu trabalho foi entregue fora do prazo e cabe ao professor julgar se modifica ou não a nota do aluno por esse motivo.

A lista de trabalhos mostrada aos alunos contém cores diferenciadas que indicam se o trabalho está ou não aberto para submissão e se ele já foi enviado pelo aluno. Existem cinco possibilidades representadas na página da Figura 4.16:

- Trabalho Aberto: Enviou trabalho
- Trabalho Aberto: Não enviou trabalho ainda
- Trabalho Encerrado: Enviou trabalho no prazo
- Trabalho Encerrado: Enviou trabalho com atraso
- Trabalho Encerrado: Não enviou trabalho

Título	Notas	Data Entrega	Histórico da Avaliação
Trabalho 2	7.0	2005-02-16	
Trabalho 3	8.0	2005-03-10	
Trabalho 4	7.9	2005-04-02	
Trabalho 5	7.0	2005-04-18	
Trabalho 6	9.0	2005-04-26	
Trabalho 7	---	---	

LEGENDA	
	Trabalho Aberto: já enviou trabalho
	Trabalho Aberto: não enviou trabalho ainda
	Trabalho Encerrado: enviou trabalho no prazo
	Trabalho Encerrado: enviou trabalho com atraso
	Trabalho Encerrado: não enviou trabalho

Figura 4.16: PROGTEST - Lista de Trabalhos Aluno

Partindo-se da página da Figura 4.16, se um trabalho já foi submetido, o aluno pode visualizar seu desempenho. Na Figura 4.17 é ilustrado o histórico da submissão do trabalho contendo informações relativas ao trabalho, à avaliação e os possíveis comentários do professor em relação ao seu trabalho.

Se um trabalho está aberto para submissões, o aluno pode submeter quantas vezes ele quiser sua solução até o data limite de entrega do trabalho. A página onde o aluno realiza a submissão do trabalho é mostrada na Figura 4.18. Primeiramente, o aluno tem acesso aos dados do trabalho como ilustra a Figura 4.18 (a) e posteriormente à página de submissões onde ele vai fazer o *upload* do seu programa e dos seus casos de teste (Figura 4.18(b)). É importante ressaltar que

Figura 4.17: PROGTEST - Histórico do Aluno

(a)

(b)

Figura 4.18: PROGTEST - Submissão de Trabalhos

tanto um trabalho estando aberto ou fechado o sistema somente armazena os dados da última submissão realizada pelo aluno.

Logo após a submissão, o aluno pode visualizar o relatório da avaliação do seu trabalho. A avaliação é realizada automaticamente, logo após a submissão. Na Figura 4.19 é ilustrado o relatório de avaliação gerado pelo sistema. Ele mostra o relatório do teste funcional utilizando o JUnit, o relatório contendo informações detalhadas a respeito do teste estrutural realizado pela JaBUTi. Ele ainda mostra o relatório com a cobertura geral obtida com as execuções trocadas cujo valor é calculado com base nos pesos que o professor definiu para os critérios de teste

estruturais. Por fim, é mostrada a nota sugerida pelo sistema, calculada com base nas cobertura geral obtida com as execuções trocadas e na realização do teste funcional.

Figura 4.19: PROGTEST - Relatório de Avaliação dos Trabalhos

O resultado da avaliação do trabalho gerado pela PROGTEST ilustrado na Figura 4.19 contém três relatórios: (1) relatório do teste funcional do aluno obtido com a execução no JUnit; (2) relatório de cobertura do teste estrutural obtido com a execução na JaBUTi; e (3) relatório de cobertura total do aluno obtido com as execuções trocadas. Por exemplo, pode-se observar neste relatório que o aluno obteve uma taxa de sucesso de 94,7% na realização do teste funcional; obteve uma cobertura de 87% no critério Todos-Nós, 77% no critério Todos-Arcos e 65% no critério Todos-DefUsos na execução do programa do aluno com os casos de teste do professor; obteve uma cobertura de 78,1% calculada com base nos pesos definidos pelo professor para os critérios de teste, na execução do programa do aluno com os casos de teste. O relatório mostra ainda, a nota sugerida pelo sistema que neste caso é 89.9.

É importante destacar que a validação do ambiente PROGTEST foi realizada de modo informal, com a inclusão de dados no sistema de forma *ad hoc*. A realização de um experimento controlado para aplicar e validar o ambiente PROGTEST em cenários reais de ensino e aprendizado é uma das atividades futuras a ser conduzida em decorrência deste trabalho.

4.8 Considerações Finais

Neste capítulo foi apresentado com detalhes como o ambiente PROGTEST foi desenvolvido. A principal funcionalidade desse ambiente é aceitar a submissão de trabalhos práticos (progra-

mas e casos de teste dos alunos escritos na linguagem Java) e avaliar automaticamente esses trabalhos utilizando, para isso, ferramentas de teste. Foi definida uma estratégia para realizar a avaliação dos trabalhos dos alunos baseada em critérios de teste. Para colocar em prática essa estratégia, o ambiente foi integrado com a ferramenta de teste JaBUTi. Aspectos arquiteturais, de modelagem, implementação e operacionais foram discutidos.

Conforme mencionado neste capítulo, ressalta-se que a forma como o módulo de avaliação foi implementado facilita sua integração com outros ambientes que realizam de forma eficaz o gerenciamento de cursos online como por exemplo, o Panda e o COL.

No próximo capítulo é apresentada uma visão geral da pesquisa realizada, sendo sintetizadas as principais contribuições deste trabalho e as linhas para pesquisa futura, decorrente de sua realização.

Conclusão

5.1 Considerações Gerais

Neste trabalho discutiu-se a relevância do ensino conjunto de teste de software e de fundamentos de programação. Tanto a programação como a atividade de teste de software não são disciplinas triviais de serem ensinadas. Existem vários problemas no ensino de programação e algumas iniciativas estão ocorrendo no sentido de tentar tornar a programação interessante e relevante para os alunos. Uma dessas iniciativas é a introdução de OO nessas disciplinas. Entretanto ainda assim, problemas relativos à atividade de programação continuam existindo, independentemente do paradigma utilizado.

Outras iniciativas sugerem a introdução de práticas de teste de software em conjunto com fundamentos de programação. Dentro desse contexto, a atividade de teste pode ser utilizada para ajudar os alunos a compreender melhor os conceitos abstratos de programação e desenvolver habilidades de compreensão e análise, uma vez que o teste requer que os alunos conheçam o comportamento de seus programas.

No entanto, a atividade de teste tem sido pouco explorada nos currículos dos cursos de graduação em Ciências da Computação. Experiências recentes têm sugerido que a atividade de teste poderia ser ensinada o mais cedo possível aos alunos, pois o teste força a aplicação prática da análise, projeto e implementação, que pode tornar os alunos melhores testadores e desenvolvedores. Em síntese, a introdução de conceitos e técnicas de teste o mais cedo possível pode: (1) melhorar a qualidade dos produtos desenvolvidos; e (2) levar a uma melhoria na qualidade do processo utilizado no desenvolvimento.

Dentro dessa perspectiva, esse trabalho apresentou alguns mecanismos para integrar o ensino de teste de software e de fundamentos de programação OO. Esses mecanismos consistiram no desenvolvimento de um módulo educacional integrado de fundamentos de programação OO e teste de software, e no desenvolvimento de um ambiente baseado na *Web*, denominado PROGTEST, para submissão e avaliação automática dos trabalhos práticos baseados em atividades de teste de software.

O módulo educacional integrado foi desenvolvido com a utilização do processo padrão para desenvolvimento de módulos educacionais estabelecido por Barbosa (Barbosa, 2004), com ênfase na aplicação da abordagem *AIM-CID*. Durante o processo de modelagem foi proposta uma evolução à abordagem *AIM-CID* a fim de adaptá-la as características desse módulo educacional. A abordagem *AIM-CID* trata somente da modelagem de um domínio do conhecimento. No entanto, durante o desenvolvimento do módulo integrado foram identificados dois domínios do conhecimento: programação OO e teste de software. Dessa forma, a abordagem *AIM-CID* foi evoluída a fim de incorporar na etapa da modelagem conceitual a integração de domínios de conhecimento distintos.

Inicialmente, foi realizada a modelagem conceitual seguindo a abordagem *AIM-CID*, para cada um dos domínios envolvidos: programação OO e teste de software. Em seguida, foram estabelecidos os pré-requisitos de cada um desses domínios e, com base nisso, foram definidos os pontos de conexão onde os conceitos de programação OO pudessem ser integrados com os conceitos de teste de software. A partir da definição desses pontos de integração, foi desenvolvido o modelo conceitual integrado de programação OO e teste de software. Tendo como base o modelo conceitual integrado, foram construídos os modelos instrucionais e didáticos de acordo com o que a abordagem *AIM-CID* propõe. Posteriormente, a partir dos modelos construídos, o módulo educacional integrado foi implementado gerando um dos resultados deste trabalho.

Outro resultado obtido com este trabalho é o ambiente PROGTEST para submissão e avaliação automática de trabalhos. O principal objetivo deste ambiente é promover um *feedback* adequado para avaliar a performance dos alunos com relação à programação e teste. Dessa forma, foi mostrada neste trabalho a abordagem estabelecida para avaliar automaticamente os programas baseadas em atividades de teste, a arquitetura, a modelagem e aspectos relativos à sua implementação e a integração com a ferramenta de teste JaBUTi (Vincenzi, 2004). Alguns aspectos operacionais também foram apresentados. Um ambiente desse tipo é importante pois avalia todo programa submetido no mesmo nível de eficiência e os resultados da avaliação são baseados nos mesmos padrões. Além disso, é possível reduzir a sobrecarga que existe sobre os professores, uma vez que o sistema avalia automaticamente tanto os casos de teste quanto o código do programa. São ainda fornecidos relatórios, após a avaliação, de cada trabalho dos alunos. Cabe ainda ressaltar que a validação da PROGTEST ocorreu de maneira informal com a inclusão de dados de forma *ad hoc*.

5.2 Contribuições

Como principais contribuições deste trabalho pode-se citar que, do ponto de vista da programação, a atividade de teste pode contribuir para a melhorar a capacidade de compreensão e análise dos alunos. Em contrapartida, do ponto de vista da atividade de teste, se a mesma for ensinada juntamente com fundamentos de programação, pode-se criar uma cultura de teste nos alunos, tornando a mesma uma prática comum entre os desenvolvedores, motivando-os a aplicar a atividade de teste desde o início do processo desenvolvimento de software. Dessa forma, pode-se melhorar tanto o produto que está sendo desenvolvido quanto o processo de desenvolvimento utilizado. A integração do ensino de teste de programação pode ainda levar a uma reformulação dos currículos dos cursos de Ciências da Computação com a introdução o mais cedo possível de aspectos de qualidade nos cursos introdutórios.

Dentro desse contexto, podem-se destacar como principais contribuições desse trabalho:

- Desenvolvimento de um módulo educacional integrado de programação OO e teste de software construído utilizando-se o processo padrão de Barbosa (Barbosa, 2004) com ênfase na aplicação da abordagem *AIM-CID*. Esse módulo é um dos mecanismos propostos nesse trabalho para dar subsídios para que a iniciativa de se integrar o ensino de programação OO e teste de software possa ser concretizada. O módulo educacional foi desenvolvido sob a perspectiva de “material didático livre” permitindo o reúso do mesmo.
- Evolução da abordagem *AIM-CID*. Essa abordagem trata a representação de um único domínio do conhecimento. A evolução da abordagem *AIM-CID* tem a finalidade de incorporar, na fase de modelagem conceitual, a integração de vários domínios distintos de conhecimento. Propõe-se que seja construído um modelo conceitual para cada domínio do conhecimento e sejam estabelecidos pontos de conexão entre esses conceitos. Depois, com base nos pontos de conexão definidos, constrói-se o modelo conceitual integrado. Os modelos intrucionais e didáticos são construídos com base no modelo conceitual integrado.
- Desenvolvimento do ambiente PROGTEST, baseado na *Web* e na concepção de software livre, para submissão e avaliação automática de trabalhos práticos. A avaliação dos trabalhos é realizada com base em atividades de teste. Para isso, a PROGTEST é integrada com ferramentas de teste, especificamente com a ferramenta JaBUTi. Esse ambiente é um dos mecanismos deste trabalho para fornecer suporte ao ensino integrado de programação OO e teste de software.

5.3 Trabalhos Futuros

Dando continuidade às atividades conduzidas durante o trabalho de mestrado, pode-se destacar os seguintes trabalhos futuros:

- Aplicar o módulo educacional integrado de programação OO e teste de software, em conjunto com o ambiente PROGTEST, em cursos introdutórios de Ciências da Computação oferecidos para estudantes de graduação no ICMC/USP. Para isso, um experimento sistemático e controlado vem sendo planejado. A objetivo é avaliar o uso prático dos mecanismos e idéias propostos em projetos e em cenários reais de aprendizado.
- Evoluir o módulo educacional de programação OO e teste de software, a fim de abordar os conceitos mais avançados de OO. Na versão atual do módulo integrado são tratados conceitos básicos e introdutórios de programação OO. Com a evolução desse módulo pode-se incluir conceitos mais avançados de OO como, por exemplo, polimorfismo, encapsulamento e herança.
- Integrar a PROGTEST com outras ferramentas e ambientes. O ambiente PROGTEST pode ser integrado a: (1) ferramentas que verificam se os programas dos alunos foram plagiados; (2) ambientes de programação pedagógicos, como DrJava (Allen et al., 2002) ou BlueJ (Patterson et al., 2003), criando assim um ambiente completo onde os alunos desenvolvam e avaliem seus trabalhos num mesmo ambiente; (3) ambientes de gerenciamento de cursos online, como Panda (IME-USP, 2006) e o COL (LARC - USP, 2006). Pode-se ainda evoluir o ambiente PROGTEST para permitir a integração com outras ferramentas de teste, possibilitando tanto a avaliação de programas em Java quanto em outras linguagens de programação.
- Definir e investigar mecanismos automatizados para a interpretação e a execução de modelos didáticos como suporte à geração automática de conteúdos educacionais, personalizados de acordo com interesses pedagógicos, perfil do usuário, características do módulo, entre outros aspectos.
- Investigar, explorar e estender os mecanismos para desenvolvimento de módulos educacionais propostos no trabalho de Barbosa (2004) (processo padrão para desenvolvimento de módulos educacionais e a abordagem *AIM-CID*) no contexto de ensino à distância. A idéia é identificar os elementos requeridos nesse novo contexto, evoluindo os mecanismos estabelecidos no trabalho de Barbosa (2004) para o desenvolvimento de material didático para cursos à distância.
- Investigar o uso dos modelos conceituais no desenvolvimento de ontologias de domínio e vice versa. Nesse sentido, os modelos conceituais para fundamentos de programação e

teste de software podem ser explorados no desenvolvimento de uma ontologia integrada para esses domínios.

- Aplicar os mecanismos investigados neste trabalho na construção de um módulo educacional de teste de software Orientado a Aspectos (OA) (Lemos et al., 2004). Esta é uma das linhas de pesquisa do grupo de Engenharia de Software do ICMC/USP. Dentro deste contexto, a ferramenta JaBUTi (Vincenzi, 2004) foi estendida para apoiar o teste estrutural de programas OA escritos em AspectJ, resultando na JaBUTi/AJ (Lemos, 2005). Desta forma, o domínio de conhecimento de teste de software OA deve ser modelado na construção do módulo e a ferramenta JaBUTi/AJ pode ser integrada ao mesmo, apoiando o ensino do teste de software OA.

Referências Bibliográficas

- ACREE, A. T.; BUDD, T. A.; DEMILLO, R. A.; LIPTON, R. J.; SAYWARD, G. F. *Mutation analysis*. Relatório Técnico GIT-ICS-79/08, Georgia Institute of Technology, Atlanta, GA, 1979.
- ADAMS, J.; FRENS, J. Object centered design for Java: Teaching OOD in CS1. In: *34th SIGCSE Technical Symposium on Computer Science Education (SIGCSE'03)*, Reno, Nevada, USA, 2003, p. 273–277.
- AGRAWAL, H.; DEMILLO, R. A.; R.HATHAWAY; HSU, W.; KRAUSER, E. W.; MARTIN, R. J.; MATHUR, A.; SPAFFORD, E. H. *Design of mutant operators for the c programming language*. Relatório Técnico SERC-TR-41-P, Software Engineering Research Center, Purdue University, West Lafayette, IN, 1989.
- ALLEN, E.; CARTWRIGHT, R.; STOLER, B. DrJava: A lightweight pedagogic environment for Java. In: *33rd ACM SIGCSE Technical Symposium on Computer Science Education (SIGCSE'02)*, Cincinnati, Kentucky, 2002, p. 137–141.
- ALPHONCE, C.; VENTURA, P. Object orientation in CS1-CS2 by design. In: *7th Annual Conference on Innovation and Technology in Computer Science Education (ITiCSE'02)*, Aarhus, Denmark, 2002, p. 70–74.
- BARBEY, S.; STROHMEIER, A. The problematics of testing object-oriented software. In: *Second Conference on Software Quality Management (SQM'94)*, Edinburgh, Scotland, UK, 1994, p. 411–426.
- BARBOSA, E. F. *Uma contribuição ao processo de desenvolvimento e modelagem de módulos educacionais*. Tese de Doutorado, ICMC/USP, São Carlos, SP, (in Portuguese), 2004.
- BARBOSA, E. F.; MALDONADO, J. C.; LEBLANC, R.; GUZDIAL, M. Introducing testing practices into objects and design course. In: *16th Conference on Software Engineering Education and Training (CSEE&T'03)*, Madrid, Spain, 2003a, p. 279–286.

- BARBOSA, E. F.; MALDONADO, J. C.; MAIDANTCHIK, C. L. L. Padronização de processos para o desenvolvimento de módulos educacionais. In: *XXIX Latin-American Conference on Informatics (CLEI'2003)*, La Paz, Bolívia, 2003b.
- BARBOSA, E. F.; MALDONADO, J. C.; RICARTE, I. L. M. Learning materials: Towards the establishment of guidelines for domain modeling. In: *IFIP WG 3.2 Working Conference on Informatics Curricula, Teaching Methods and Best Practice (ICTEM'2002)*, Florianópolis, SC, 2002, p. 192–206.
- BARBOSA, E. F.; MALDONADO, J. C.; VINCENZI, A. M. R.; DELAMARO, M. E.; SOUZA, S. R. S.; JINO, M. Introdução ao teste de software. In: *Minicurso apresentado no XIV Simpósio Brasileiro de Engenharia de Software (SBES'00)*, João Pessoa, PB, 2000, p. 330–378.
- 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: *ACM SIGCSE Bulletin*, 2002, p. 125–128.
- BECK, K. *Extreme programming explained: Embrace change*. Addison Wesley, 1999.
- BIEMAN, J. M.; GHOSH, S.; ALEXANDER, R. T. A technique for mutation of java objects. In: *16th IEEE International Conference on Automated Software Engineering (ASE'01)*, San Diego, Califórnia, 2001, p. 337–340.
- BINDER, R. V. *Testing object-oriented systems: Models, patterns, and tools*, v. 01. Addison-Wesley Longman, Inc., 1999.
- BUCK, D.; STUCKI, D. J. 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*, 2000, p. 16–20.
- BUDD, T. A. *Computer program testing*, cap. Mutation analysis: Ideas, example, problems and prospects North-Holand Publishing Company, p. 129–148, 1981.
- 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 Annual ACM Symposium on Principles of Programming Languages (POPL'80)*, Las Vegas, Nevada, 1980, p. 220–233.
- BUY, U.; ORSO, A.; ; PEZZÈ, M. Automated testing of classes. In: *International Symposium on Software Testing and Analysis (ISSTA'00)*, Portland, Oregon, 2000, p. 39–48.
- CAMPBELL, W.; BOLKER, E. Teaching programming by immersion, reading and writing. In: *32nd ASEE/IEEE Frontiers in Education Conference*, Boston, Massachusetts, 2002, p. 23–28.

- CHAIM, M. L. *Poke-tool - uma ferramenta para suporte ao teste estrutural de programas baseados em análise de fluxo de dados*. Dissertação de mestrado, DCA/FEEC/UNICAMP, Campinas, SP, 1991.
- CHEN, M.; KAO, H. M. Testing object-oriented programs - an integrated approach. In: *10th International Symposium on Software Reliability Engineering (ISSRE'99)*, Boca Raton, Florida, 1999, p. 73–83.
- CHEN, T. Y. Experience with teaching black-box testing in a computer science/software engineering curriculum. *IEEE Transactions on Education*, v. 47, n. 01, p. 42–50, 2004.
- CHEN, T. Y.; POON, P. L. Teaching black box testing. In: *IEEE International Conference Software Engineering: Education and Practice*, 1998, p. 324–329.
- CONALLEN, J. *Desenvolvimento de aplicações web com uml*. Editora Campus, 2003.
- COOPER, S.; DANN, W.; PAUSCH, R. Teaching objects-first in introductory computer science. In: *34th SIGCSE Technical Symposium on Computer Science Education (SIGCSE'03)*, Reno, Nevada, USA, 2003, p. 191–195.
- CORTE, C. K. D.; BARBOSA, E. F.; MALDONADO, J. C. *Manual do usuário do ambiente progtest*. Instituto de Ciências Matemáticas e de Computação, ICMC-USP, São Carlos, documento de Trabalho, em elaboração, 2006a.
- CORTE, C. K. D.; BARBOSA, E. F.; MALDONADO, J. C. *Modelagem do ambiente progtest*. Instituto de Ciências Matemáticas e de Computação, ICMC-USP, São Carlos, documento de Trabalho, em elaboração, 2006b.
- DALY, C.; HORGAN, J. M. An automated learning system for java programming. *IEEE Transactions on Education*, v. 47, n. 01, p. 10–17, 2004.
- DELAMARO, M. E.; MALDONADO, J. C. Proteum: A tool for the assessment of test adequacy for c programs. In: *Conference on Performability in Computing Systems (PCS'96)*, Brunswick, NJ, 1996, p. 79–95.
- DELAMARO, M. E.; MALDONADO, J. C.; MATHUR, A. P. Interface mutation: An approach for integration testing. *IEEE Transactions on Software Engineering (TSE)*, v. 27, n. 03, p. 228–247, 2001a.
- DELAMARO, M. E.; PEZZÈ, M.; VINCENZI, A. M. R.; MALDONADO, J. C. Mutant operators for testing concurrent java programs. In: *XV Simpósio Brasileiro de Engenharia de Software (SBES'01)*, Rio de Janeiro, RJ, 2001b, p. 272–285.

- DEMILLO, R. A. Mutation analysis as a tool for software quality assurance. In: *4th IEEE International Computer Software and Applications Conference (COMPSAC'80)*, Los Alamitos, CA., 1980, p. 390–393.
- DEMILLO, R. A.; LIPTON, R. J.; SAYWARD, F. G. Hints on test data selection: Help for the practicing programmer. *IEEE Computer*, v. 11, n. 04, p. 34–41, 1978.
- DOMINGUES, A. L. S. *Avaliação de critérios e ferramentas de teste para programas oo*. Dissertação de mestrado, ICMC/USP, São Carlos - SP, 2002.
- 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.
- FERRARI, F. C. Apoio ao teste estrutural e de mutação de software orientado a objetos e a aspectos. Exame Geral de Qualificação para Mestrado, ICMC/USP, São Carlos/SP - Brasil (Mestrado em andamento), 2005.
- GHOSH, S.; MATHUR, A. P. Interface mutation. *Software Testing, Verification and Reliability Journal (STVR)*, v. 11, n. 04, p. 227–247, 2001.
- GOLDBERG, M.; SALARI, S.; ; SWOBODA, P. World wide web - course tool: An environment for building www-based courses. In: *5th International World Wide Web Conference*, Paris, França, 1996, p. 1219–1231.
- GOLDWASSER, M. H. A gimmick to integrate software testing throughout the curriculum. In: *33rd SIGCSE Technical Symposium on Computer Science Education (SIGCSE'02)*, Cincinnati, Kentucky., 2002, p. 271–275.
- GUZDIAL, M.; RICK, J.; KEHOE, C. Beyond adoption to invention: Teacher-created collaborative activities in higher education. *Journal of the Learning Sciences*, 2002.
- HARROLD, M. J.; ; ROTHERMEL, G. Performing data-flow testing on classes. In: *2nd ACM SIGSOFT Symposium on Foundations of Software Engineering (SIGSOFT'94)*, New Orleans, LA, 1994, p. 154–163.
- HARROLD, M. J. Testing: A roadmap. In: *22nd International Conference on Software Engineering (ICSE'00), Future of Software Engineering Track*, Limerick, Irlanda, 2000, p. 61–72.
- HICKEY, T. J. Scheme-based web programming as a basis for a CS0 curriculum. In: *35th SIGCSE Technical Symposium on Computer Science Education*, Norfolk, Virginia, USA, 2004, p. 353–357.

- HORGAN, J. R.; MATHUR, A. P. Assessing testing tools in research and education. *IEEE Software*, v. 09, n. 03, p. 61–69, 1992.
- IME-USP Panda - Sistema On Line de Acompanhamento Didático. [On-line]. Disponível em <http://panda2.arca.ime.usp.br/panda/index.html> (Último acesso em 10/01/2006)
- IRIMIA, A. Enhancing the introductory computer science curriculum: C++ or java. *The Journal of Computing in Small Colleges (JCSC)*, v. 17, n. 02, p. 159–166, 2001.
- ISO *Information technology - software life cycle processes*. International Standard Organization, padrão ISO/IEC 12207, 1995.
- ISONG, J. Developing an automated program checker. *The Journal of Computing in Small Colleges (JCSC)*, v. 16, n. 03, p. 218–224, 2001.
- JACKSON, D.; USHER, M. Grading student programs using assyst. In: *Twenty-eighth SIGCSE technical symposium on Computer science education*, San Jose, California, United States, 1997, p. 335–339.
- JONES, E. L. An experimental approach to incorporating software testing into the computer science curriculum. In: *31st ASEE/IEEE Frontiers in Education Conference*, Reno, Nevada, 2001a, p. 7–11.
- JONES, E. L. Grading student programs - a software testing approach. In: *4th Annual Consortium on Small Colleges Southeastern Conference*, Salem, Virginia, 2001b, p. 185–192.
- KAMIN, S. N.; MICKUNAS, M. D.; REINGOLD, E. M. *An introduction to computer science using java*. Second edition ed. McGraw-Hill, 2002.
- KIM, S.; CLARK, J. A.; MCDERMID, J. A. Assessing test set adequacy for object-oriented programs using class mutation. In: *Symposium on Software Technology (SoST'99)*, 1999a, p. 72–83.
- KIM, S.; CLARK, J. A.; MCDERMID, J. A. The rigorous generation of java mutation operators using hazop. In: *12th International Conference on Software & Systems Engineering and their Applications (ICSSEA'99)*, 1999b.
- KNUDSEN, J. L.; MADSEN, O. L. Using object-orientation as a common basis for system development education. *ACM SIGPLAN Notices*, v. 31, n. 12, p. 52–62, 1996.
- KOLLING, M.; ROSENBERG, J. Guidelines for teaching object orientation with java. In: *6th Annual Conference on Innovation and Technology in Computer Science Education (ITiCSE'01)*, Canterbury, UK, 2001, p. 33–36.

- KORHONEN, A.; MALMI, L.; SAIKKONEN, R. Experiences in automatic assessment on mass courses and issues for designing virtual courses. In: *7th Annual Conference on Innovation and Technology in Computer Science Education (ITiCSE'02)*, Aarhus, Denmark, 2002, p. 55–59.
- 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.
- LARC - USP CoL (Cursos on Line). [On-line].
Disponível em <http://col.usp.br/portal/> (Último acesso em 10/01/2006)
- LEMONS, O. A. L. *Teste de programas orientados a aspectos: Uma abordagem estrutural para aspectj*. Dissertação de mestrado, ICMC/USP, São Carlos/SP, 2005.
- LEMONS, O. A. L.; VINCENZI, A. M. R.; MALDONADO, J. C.; MASIERO, P. C. Teste de unidade de programas orientados a aspectos. In: *Anais do 18th Simpósio Brasileiro de Engenharia de Software (SBES'2004)*, Brasília/DF - Brasil, 2004, p. 55–70.
- LUCENA, C. J. P.; FUKS, H.; MILIDIU, R.; MACEDO, L.; SANTOS, N.; LAUFER, C.; RIBEIRO, M. B.; FONTOURA, M. F.; NOYA, R. C.; CRESPO, S.; TORRES, V.; DAFLON, L.; LUKOWIECKI, L. Aulanet - an environment for the development and maintenance of courses on the web. In: *International Conference on Engineering Education (ICEE'98)*, Rio de Janeiro, RJ, 1998.
- MAIDANTCHIK, C. L. L. *Gerência de processos de software para equipes geograficamente dispersas*. Tese de doutoramento, COPPE/Universidade Federal do Rio de Janeiro, Rio de Janeiro, RJ, 1999.
- MALDONADO, J. C. *Crerios potenciais usos: Uma contribuio ao teste estrutural de software*. Tese de doutoramento, DCA/FEEC/UNICAMP, Campinas, SP, 1991.
- MALDONADO, J. C.; FABBRI, S. C. P. F. *Qualidade de software: Teoria e prtica*, cap. Teste de Software PrenticeHall, p. 73–84, 2001.
- MASSOL, V.; HUSTED, T. *Junit em ao*. 1^a edio ed. Cincia Moderna, 2005.
- MERRIL, M. D. Component display theory. In: *Instructional Design Theories and Models: An Overview of their Current States*, Hillsdale, NJ:Lawrence Erlbaum, 1983.
- MYERS, G. J.; SANDLER, C.; BADGETT, T.; THOMAS, T. M. *The art of software testing*. 2nd. ed. John Wiley & Sons, 2004.

MySQL MySQL. [On-line].

Disponível em <http://www.mysql.com/> (Último acesso em 10/01/2006)

NEUBAUER, B. J.; STRONG, D. D. The object-oriented paradigm: More natural or less familiar? *The Journal of Computing in Small Colleges (JCSC)*, v. 18, n. 01, p. 280–289, 2002.

NOVAK, J. D. Concept mapping: A useful tool for science education. *Journal of Research in Science Teaching*, v. 27, n. 10, p. 937–949, 1990.

OFFUTT, J.; IRVINE, A. Testing object-oriented software using the category-partition method. In: *Technology of Object-Oriented Languages and Systems (TOOLS'95)*, Santa Barbara, CA, 1995, p. 293–304.

OFFUTT, J.; MA, Y.; KWON, Y. Inter-class mutation operators for java. In: *13th International Symposium on Software Reliability Engineering (ISSRE'02)*, Annapolis, MD, 2002, p. 352–366.

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

PERRY, D. E.; KAISER, G. E. Adequate testing and object-oriented programming. *The Journal on Object Oriented Programming (JOOP)*, v. 02, n. 05, p. 13–19, 1990.

PRESSMAN, R. S. *Software engineering - a practitioner's approach*. 5th ed. McGraw-Hill, 2001.

PRESSMAN, R. S. *Software engineering - a practitioner's approach*. 6th ed. McGraw-Hill, 2005.

RAJARAVIVARMA, R.; PEVAC, I. When to introduce objects in teaching java. In: *IEEE 35th Southeastern Symposium on System Theory*, Morgantown, Virginia, 2003, p. 437–440.

RAPPS, S.; WEYUKER, E. J. Data flow analysis techniques for test data selection. In: *6th International Conference on Software Engineering (ICSE'82)*, Tóquio, Japão, 1982, p. 272–278.

RAPPS, S.; WEYUKER, E. J. Selecting software test data using data flow information. *IEEE Transactions on Software Engineering (TSE)*, v. 11, n. 04, p. 367–375, 1985.

REID, D. The reid list of the first course language for computer science majors. <http://www.csee.wvu.edu/vanscoy/reid.htm>, 2004.

- RICHARDS, B. Experiences incorporating java into the introductory sequence. *The Journal of Computing in Small Colleges (JCSC)*, v. 19, n. 02, p. 247–253, 2003.
- ROBERTS, E.; PICARD, A. Designing a java graphics library for cs1. In: *3rd Annual Conference on Innovation and Technology in Computer Science Education (ITiCSE'98)*, Dublin, Ireland, 1998, p. 213–218.
- SAIKKONEN, R.; MALMI, L.; KORHONEN, A. Fully automatic assessment of programming exercises. In: *6th Annual Conference on Innovation and Technology in Computer Science Education (ITiCSE'01)*, 133-136, 2001, p. Canterbury, United Kingdom.
- SATRATZEMI, M.; XINOGALOS, S.; DAGDILELIS, V. An environment for teaching object-oriented programming: Objectkarel. In: *The 3rd IEEE International Conference on Advanced Learning Technologies (ICALT'03)*, Athens, Greece, 2003, p. 342–343.
- SAVITCH, W. *Java - an introduction to computer science & programming*. Second edition ed. Prentice Hall, 2001.
- SEBESTA, R. W. *Conceitos de linguagens de programação*. 4ª edição ed. Bookman Companhia Editora Ltda, 2000.
- SHEPARD, T.; LAMB, M.; KELLY, D. More testing should be taught. *Communications of the ACM*, v. 44, n. 06, p. 103–108, 2001.
- SOUTER, A. L.; POLLOCK, L. L. The construction of contextual def-use associations for object-oriented systems. *IEEE Transactions on Software Engineering (TSE)*, v. 29, n. 11, p. 1005–1018, 2003.
- SOUTER, A. L.; POLLOCK, L. L.; HISLEY, D. Inter-class def-use analysis with partial class representations. In: *ACM Workshop Program Analysis for Software Tools and Engineering*, Toulouse, France, 1999, p. 47–56.
- SUN MICROSYSTEMS JavaServer Faces Technology. [On-line].
Disponível em <http://java.sun.com/j2ee/jvaserverfaces/> (Último acesso em 10/01/2006)
- SUN MICROSYSTEMS Model-View-Controller. [On-line].
Disponível em <http://java.sun.com/blueprints/patterns/MVC.html> (Último acesso em 10/01/2006)
- THE APACHE JAKARTA PROJECT The Apache Tomcat 5.5 Servlet/JSP Container. [On-line].
Disponível em <http://tomcat.apache.org/tomcat-5.5-doc/index.html> (Último acesso em 10/01/2006)

- TURINE, M. A. S. *Hmbs: Um modelo baseado em statecharts para a especificação formal de hiperdocumentos*. Tese de doutoramento, IFSC/USP, São Carlos, SP, 1998.
- TURINE, M. A. S.; OLIVEIRA, M. C. F.; MASIERO, P. C. Designing structured hypertext with hmbs. In: *Eighty ACM Conference on Hypertext (Hypertext 97)*, Southampton, UK, 1997, p. 241–256.
- VERGÍLIO, S. R.; MALDONADO, J. C.; JINO, M. Uma estratégia para a geração de dados de teste. In: *VII Simpósio Brasileiro de Engenharia de Software*, Rio de Janeiro - RJ, 1993, p. 307–319.
- VINCENZI, A. M. R. *Orientação a objetos: Definição e análise de recursos de teste e validação*. Tese de doutoramento, ICMC/USP, São Carlos, SP, 2004.
- 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'03), Sessão de Ferramentas*, Manaus, AM, 2003, p. 79–84.
- ZHU, H. A formal analysis of the subsume relation between software test adequacy criteria. *IEEE Transactions on Software Engineering (TSE)*, v. 22, n. 04, p. 248–255, 1996.

Scripts utilizados na integração da JaBUTi com o ambiente PROGTTEST

Conforme mencionado no Capítulo 4, a versão atual da ferramenta de teste JaBUTi funciona por meio de sua interface gráfica. Desta forma, para que o ambiente PROGTTEST pudesse ser integrado com a ferramenta JaBUTi foi necessária a implementação de *scripts* para que a ferramenta executasse através de linhas de comando. Assim, foram implementados os seguintes *scripts*¹: (1) criação do projeto de teste; (2) importação dos casos de teste no JUnit; e (3) geração dos relatórios de cobertura. Esses scripts são listados a seguir:

Listagem A.1: Criação do projeto de teste

```
package br.jabuti.cmdtool;

import org.apache.bcel.classfile.*;
import java.io.*;
import java.util.*;
import br.jabuti.graph.*;
import br.jabuti.project.*;
import br.jabuti.lookup.*;

public class CreateProject {
    public static void usage() {
        System.out.println("JaBUTi CreateProject");
        System.out.println("\nUSAGE:");
        System.out.println("java -cp \"classpath\" cmdtool.CreateProject -b <base_class_file> \" +
        "-p <project_file_name> [-mob] [-cn] [-i <file_name>] [-a <avoided_packages>]\n");
        System.out.println(" -b <base_class> The name of the base class file \" +
```

¹Esses scripts foram implementados com a colaboração do Prof. Dr. Auri M. R. Vincenzi (UFG)

```

        "(without the .class extension)");
System.out.println(" Observe that the classpath variable should be");
System.out.println(" set such that the base class can be found.");
System.out.println(" -p <project_file_name> The full name of the " +
        "project file to be created.");
System.out.println(" The default extension of the file is .jbt.");
System.out.println(" -mob To test mobile code. If not specified " +
        "it is considered non-mobile.");
System.out.println(" -cn Expand the call nodes during " +
        "the CFG construction. If not specified");
System.out.println(" the call nodes are hidden.");
System.out.println(" -i <file_name> This is the name of a text file where the");
System.out.println(" name of the classes to be instrumented can be found.");
System.out.println(" The names should be one per line. If ommited, all");
System.out.println(" user classes derived from the base class are selected");
System.out.println(" to be instrumented.");
System.out.println(" -a <avoided_packages> This is the name of a text file where the");
System.out.println(" packages to be avoided can be found.");
System.out.println(" The names of the packages should be one per line.");
System.out.println(" If ommited, no package is avoided.");
System.out.println("\nCopyright (c) 2005\n");
}

public static void main(String args[]) throws Throwable {
    boolean isMobility = false;

    JavaClass jc = null;

    HashSet toInstrument = null;
    HashSet toAvoid = null;

    String baseClass = null;
    String instFile = null;
    String avoidFile = null;
    String prjFile = null;
    int cfgOption = CFG.NO_CALL_NODE;

    if (args.length > 0) {
        int i = 0;
        if (args.length == 1) {
            usage();
            return;
        }

        while (i < args.length && args[i].startsWith("-")) {
            // -b: Base Class Name
            if (("b".equals(args[i])) && (i < args.length - 1)) {
                if (baseClass == null) {
                    i++;
                    baseClass = args[i];
                } else {
                    System.out.println("Only one -b option is allowed. ");
                    System.out.println("try java cmdtool.CreateProject -h for help.");
                    return;
                }
            }
            // Testing mobile code
            else if (("mob".equals(args[i]))) {
                isMobility = !isMobility;
            }
            // Show call nodes

```

```

else if (("-cn".equals(args[i]))) {
    cfgOption = CFG.NONE;
} // The project file name
else if (("-p".equals(args[i])) && (i < args.length - 1)) {
    if (prjFile == null) {
        i++;
        prjFile = args[i];
    } else {
        System.out.println("Only one -p option is allowed. ");
        System.out.println("try java cmdtool.CreateProject -h for help.");
        return;
    }
} // Classes to Instrument file...
else if (("-i".equals(args[i])) && (i < args.length - 1)) {
    if (instFile == null) {
        i++;
        instFile = args[i];
    } else {
        System.out.println("Only one -i option is allowed. ");
        System.out.println("try java cmdtool.CreateProject -h for help.");
        return;
    }
} // Packages to Avoid file...
else if (("-a".equals(args[i])) && (i < args.length - 1)) {
    if (avoidFile == null) {
        i++;
        avoidFile = args[i];
    } else {
        System.out.println("Only one -a option is allowed. ");
        System.out.println("try java cmdtool.CreateProject -h for help.");
        return;
    }
}

else {
    System.out.println("Unrecognized option: " + args[i]);
    System.out.println("try java cmdtool.CreateProject -h for help.");
    return;
}

i++;
}

// Reading the file containing the name of the classes to be
// instrumented
if ( instFile != null ) {
    BufferedReader br = null;
    toInstrument = new HashSet();

    try {
        br = new BufferedReader( new FileReader( instFile ) );
        String fname = br.readLine();
        while( fname != null ) {
            toInstrument.add( fname );
            fname = br.readLine();
        }
    } catch( FileNotFoundException fnfe ) {
        System.err.println( "File " + instFile + " cannot be found!" );
        fnfe.getMessage();
        fnfe.printStackTrace();
    }
}

```

```

    catch( IOException ioe ) {
        System.err.println( "Error reading " + instFile + " file!" );
        ioe.getMessage();
    }
    finally {
        if ( br != null )
            br.close();
    }
}

// Reading the file containing the name of the packages to be
// avoided
if ( avoidFile != null ) {
    BufferedReader br = null;
    toAvoid = new HashSet();

    try {
        br = new BufferedReader( new FileReader( avoidFile ) );
        String fname = br.readLine();
        while( fname != null ) {
            toAvoid.add( fname );
            fname = br.readLine();
        }
    } catch( FileNotFoundException fnfe ) {
        System.err.println( "File " + avoidFile + " cannot be found!" );
        fnfe.getMessage();
        fnfe.printStackTrace();
    }
    catch( IOException ioe ) {
        System.err.println( "Error reading " + avoidFile + " file!" );
        ioe.getMessage();
    }
    finally {
        if ( br != null )
            br.close();
    }
}

String classPath = System.getProperty("java.class.path");
JabutiProject jbtProject = null;

    // Try to create a project from the base class
    if ( baseClass != null && prjFile != null ) {
        try {
            jbtProject = new JabutiProject( baseClass, classPath );
        }
        catch (Exception e) {
            System.err.println( "Error creating the project " + prjFile );
            e.getMessage();
        }

        if ( jbtProject == null ) {
            System.err.println( "Error creating the project " + prjFile );
            return;
        }
        // Setting the project file name.
        jbtProject.setProjectFile( new File( prjFile ) );
        jbtProject.setMobility( isMobility );
        jbtProject.setCFGOption( cfgOption );
    }

```

```

        // Checking the set of packages to be avoided
        Program prg = jbtProject.getProgram();
        // If null no package is avoided
        if ( toAvoid != null ) {
            String[] packages = prg.getCodePackages();
            HashSet pkgSet = new HashSet();
            for ( int j = 0; j < packages.length; j++ ) {
                pkgSet.add( packages[j] );
            }
            String[] names = (String[]) toAvoid.toArray( new String[0] );
            for( int j = 0; j < names.length; j++ ) {
                if ( !pkgSet.contains( names[j] ) ) {
                    System.out.println( "NOT Avoided Package: " + names[j]);
                    toAvoid.remove( names[j] );
                } else {
                    System.out.println( "Avoided Package: " + names[j] );
                }
            }
            Iterator it = toAvoid.iterator();
            while( it.hasNext() ) {
                jbtProject.addAvoid( (String) it.next() );
            }
        }

        // Check the set of classes to be instrumented
        // If null, all the user classes are instrumented
        if ( toInstrument != null ) {
            String[] classes = prg.getCodeClasses();
            HashSet classSet = new HashSet();
            for ( int j = 0; j < classes.length; j++ ) {
                classSet.add( classes[j] );
            }
            String[] names = (String[]) toInstrument.toArray( new String[0] );
            for ( int j = 0; j < names.length; j++ ) {
                if ( !classSet.contains( names[j] ) ) {
                    System.out.println( "NOT Instrumented Class: " + names[j]);
                    toInstrument.remove( names[j] );
                } else {
                    System.out.println( "Instrumented Class: " + names[j]);
                }
            }
            Iterator it = toInstrument.iterator();
            while( it.hasNext() ) {
                jbtProject.addInstr( (String) it.next() );
            }
        } else { // All user classes selected to be instrumented
            String[] classes = prg.getCodeClasses();
            for ( int j = 0; j < classes.length; j++ ) {
                jbtProject.addInstr( classes[j] );
            }
        }
        // Rebuilding all the project info, including the classes
        // to be instrumented
        jbtProject.rebuild();

        TestSet.initialize( jbtProject, jbtProject.getTraceFileName() );

        try{

```

```

        jbtProject.saveProject();
    } catch (Exception e) {
        System.err.println( "Error saving project file: " +
                            jbtProject.getProjectFile() );
    }

    System.out.println( "Project Coverage" );
    System.out.println( jbtProject.coverage2TXT("") );

    } else {
        System.out.println("One base class and a project file name should be specified!");
        System.out.println("try java cmdtool.CreateProject -h for help.");
        return;
    }
} else {
    usage();
}
}
}

```

Listagem A.2: Importação de casos de teste JUnit

```

package br.jabuti.cmdtool;

import org.apache.bcel.util.ClassPath;
import org.apache.bcel.classfile.*;
import java.io.*;
import java.util.*;
import br.jabuti.project.*;
import br.jabuti.util.JUnitLoader;

public class JUnitTestCase {
    public static void usage() {
        System.out.println("JaBUTi JUnitTestCase");
        System.out.println("\nUSAGE:");
        System.out.println("java -cp \"classpath\" cmdtool.JUnitTestCase -list
-p <project_file_name> " +
            " -tc <tc_class_name> [-f n] [-t n] [-x \"list\"]\n");
        System.out.println(" -p <project_file_name> " +
            " The full name of the project file to be created.");
        System.out.println(" The default extension of the file is .jbt.");
        System.out.println(" -tc <tc_class_name> " +
            " The name of the test case class file (without the .class extension)");
        System.out.println(" Observe that the classpath variable should be");
        System.out.println("set such that the base class can be found.");

        System.out.println("java -cp \"classpath\" cmdtool.JUnitTestCase -import " +
            "-p <project_file_name> -tc <tc_class_name> [-f n] [-t n] [-x \"list\"]\n");
        System.out.println(" -p <project_file_name> " +
            " The full name of the project file to be created.");
        System.out.println(" The default extension of the file is .jbt.");
        System.out.println(" -tc <tc_class_name> " +
            " The name of the test case class file (without the .class extension)");
        System.out.println(" Observe that the classpath variable should be");
        System.out.println(" set such that the base class can be found.");
        System.out.println(" -f n " +
            " n determines the number of the first test case on which\n");
        System.out.println(" the selected operation is going to be applied. Test cases are\n");
    }
}

```

```

        System.out.println(" numbered from 0. ");
        System.out.println(" -t n" +
            " n determines the number of the last test case which the selected\n");
        System.out.println(" operation is going to be applied.The number of the " +
            "first test case to be imported.");
        System.out.println(" -x list" +
            " Determines a list of test cases which the operation is going to be\n");
        System.out.println(" applied. Since this list might have more than one number," +
            " it should be\n");
        System.out.println("enclosed in double quotes (\") in order to protect it from the shell.");
        System.out.println(" If -f or -x option is omitted, all test cases are imported.");
        System.out.println("\nCopyright (c) 2005\n");
    }

    public static void main(String args[]) throws Throwable {
        JavaClass jc = null;

        String projectName = null;
        String testCaseName = null;
        boolean fOption = false;
        int fValue = -1;
        boolean tOption = false;
        int tValue = -1;
        boolean xOption = false;
        String xValue = new String();

        boolean listOption = false;
        boolean importOption = false;

        if (args.length > 0) {
            int i = 0;

            if (args.length == 1) {
                usage();
                return;
            }

            if ("-list".equals(args[i])) {
                listOption = true;
            } else if ("-import".equals(args[i])) {
                importOption = true;
            } else {
                System.out.println("-list|-import option expected...");
                return;
            }
            i++;

            while (i < args.length && args[i].startsWith("-")) {
                // -tc: Test Case Class Name
                if ("-tc".equals(args[i])) && (i < args.length - 1)) {
                    if (testCaseName == null) {
                        i++;
                        testCaseName = args[i];
                    } else {
                        System.out.println("Only one -tc option is allowed. ");
                        System.out.println("try java cmdtool.ImportJUnitTestCase for help.");
                        return;
                    }
                }
                i++;
            }
            // The project file name

```

```

else if ("p".equals(args[i])) && (i < args.length - 1)) {
    if (projectName == null) {
        i++;
        projectName = args[i];
    } else {
        System.out.println("Only one -p option is allowed. ");
        System.out.println("try java cmdtool.ImportJUnitTestCase for help.");
        return;
    }
} // Initial test case number
else if ("f".equals(args[i])) && (i < args.length - 1)) {
    if (fOption == false) {
        i++;
        fValue = Integer.parseInt(args[i]);
        fOption = true;
    } else {
        System.out.println("Only one -f option is allowed. ");
        System.out.println("try java cmdtool.ImportJUnitTestCase for help.");
        return;
    }
} // Final test case number
else if ("t".equals(args[i])) && (i < args.length - 1)) {
    if (tOption == false) {
        i++;
        tValue = Integer.parseInt(args[i]);
        tOption = true;
    } else {
        System.out.println("Only one -t option is allowed. ");
        System.out.println("try java cmdtool.ImportJUnitTestCase for help.");
        return;
    }
} // List of test case numbers
else if ("x".equals(args[i])) && (i < args.length - 1)) {
    if (xOption == false) {
        i++;
        xValue = args[i];
        xOption = true;
    } else {
        System.out.println("Only one -x option is allowed. ");
        System.out.println("try java cmdtool.ImportJUnitTestCase for help.");
        return;
    }
} else {
    System.out.println("Unrecognized option: " + args[i]);
    System.out.println("try java cmdtool.CreateProject -h for help.");
    return;
}
i++;
}

if (projectName == null) {
    System.out.println("Missing project name...");
    return;
}

String classPath = System.getProperty("java.class.path");
File theFile = new File(projectName);

if (!theFile.isFile()) {

```



```

        System.out.println("File " + theFile.getName() + " not found...");
        throw new FileNotFoundException();
    }

    ClassPath cp = new ClassPath(classPath);
    File tcFile = new File(cp.getPath(testCaseName+".class"));

    if (!tcFile.exists()) {
        System.out.println("Test case file " + tcFile.getName() + " not found...");
        throw new FileNotFoundException();
    }

    if (importOption) {
        String list = new String();

        if (fOption && !tOption) {
            System.out.println("-f option without -t option");
            return;
        }

        if (!fOption && tOption) {
            System.out.println("-t option without -f option...");
            return;
        }

        if (fOption && tOption) {
            for (int j = fValue; j <= tValue; j++) {
                list += j + " ";
            }
        }

        if (xOption) {
            list += xValue;
        }

        Vector param = new Vector();
        param.add(testCaseName);

        Vector toImport = JUnitLoader.testCaseNumbers2Names(tcFile, classPath, list, false);
        param.addAll(toImport);

        System.out.print("\n>> Found " + (param.size()-1) + " test cases in '"
            + tcFile.getName() + "': ");
        for (int j = 1; j < param.size(); j++) {
            System.out.print(param.get(j) + " ");
        }
        System.out.println();

        JabutiProject jbtProject = JabutiProject.reloadProj(theFile.toString(), true);

        //Importing test cases
        int result = JUnitLoader.loadTestCase(jbtProject, tcFile, param);

        switch (result) {
            case JUnitLoader.NO_TEST_CASE:
                System.out.println("No test case selected do be imported.");
                break;

            case JUnitLoader.NO_PUBLIC_SET_UP:

```

```

        System.out.println("Please, provide public access to the setUp() "
            + "method such that it can be invoked by reflection.");
        break;

    case JUnitLoader.NO_PUBLIC_TEAR_DOWN:
        System.out.println("Please, provide public access to the tearDown() "
            + "method such that it can be invoked by reflection.");
        break;

    case JUnitLoader.INSTRUMENTER_ERROR:
        System.out.println("Error during instrumentation");
        break;

    case JUnitLoader.SUCCESS:
        System.out.println(param.size() - 1 + " new test cases imported successfully!");
        break;
    }

    } else if (listOption) {
        System.out.println(JUnitLoader.listTestCases(tcFile, classPath, true));
    }
} else {
    usage();
}
return;
}
}

```

Listagem A.3: Geração dos relatório de cobertura em XML

```

package br.jabuti.cmdtool;

import java.io.*;
import org.w3c.dom.Document;
import br.jabuti.project.*;
import br.jabuti.util.*;

public class JabutiReport {
    public static void usage() {
        System.out.println("java cmdtool.JabutiReport -p <project_file> -all " +
            "-pr -cl -me -ts -tc [-o <report_file_name>] \n");
        System.out.println(" -p <project_file> Specify the name of the Project to be used. The");
        System.out.println(" Project must be a valid Project file (.jba) generated by");
        System.out.println(" instrument the base class.");
        System.out.println(" -all Geraneta a complete report, including");
        System.out.println(" information from project to method level, and the");
        System.out.println("test set and test case coverages.");
        System.out.println(" -pr Coverage summary by project.");
        System.out.println(" -cl Coverage summary by class.");
        System.out.println(" -me Coverage summary by method.");
        System.out.println(" -ts Coverage summary by test set.");
        System.out.println(" -tc Coverage summary by test case.");
        System.out.println(" -o <report_file_name> " +
            " The name of the testing report to be generated. If ommited\n" +
            " it is generated a file named jabuti_report.xml.");
        System.out.println("\nCopyright (c) 2005\n");
    }
}

```

```

public static void main(String args[]) throws Throwable {
    String relatName = new String("jabuti_report.xml");
    String projectName = null;

    JabutiProject prj = null;

    boolean isPr=false;
    boolean isCl=false;
    boolean isMe=false;
    boolean isTs=false;
    boolean isTc=false;

    if (args.length > 0) {
        int i = 0;

        while ((i < args.length) && (args[i].startsWith("-"))) {
            // Recovering the project
            if ("p".equals(args[i])) && (i < args.length - 1) {
                i++;
                projectName = args[i];
            } // Report file name
            else if ("o".equals(args[i])) && (i < args.length - 1) {
                i++;
                relatName = args[i];
            } // Complete summary
            else if ("-all".equals(args[i])) {
                isPr=isCl=isMe=isTs=isTc=true;
            } // Project summary
            else if ("-pr".equals(args[i])) {
                isPr=true;
            } // Class summary
            else if ("-cl".equals(args[i])) {
                isPr=isCl=true;
            } // Method summary
            else if ("-me".equals(args[i])) {
                isPr=isCl=isMe=true;
            } // Test set summary
            else if ("-ts".equals(args[i])) {
                isTs=true;
            } // Test case summary
            else if ("-tc".equals(args[i])) {
                isTs=isTc=true;
            } else {
                System.out.println("Error: Unrecognized option: " + args[i]);
                System.out.println("try java cmdtool.CoverageSummary -h for help.");
                return;
            }
            i++;
        }

        // Checking if all essential parameters are not null
        if (projectName == null) {
            System.out.println("Error: Missing parameter!!!");
            usage();
            return;
        }

        try {
            File theFile = new File(projectName);

```

```
if (!theFile.isFile()){
    System.out.println("File " + theFile.getName() + " not found");
    return;
}

prj = JabutiProject.reloadProj( theFile.toString(), true );

if ( relatName.length() > 0 ) {
    if ( !relatName.endsWith( ".xml" ) ) {
        relatName = relatName + ".xml";
    }
}

Document htmlDoc = null;
try{
    htmlDoc = XMLGen.customReport(prj, isPr, isCl, isMe, isTs, isTc);
    XMLPrettyPrinter.writeDocument( htmlDoc, relatName );
    System.out.println("Report " + relatName + " successfully generated.");
} catch ( Exception pce ) {
    System.out.println("Error saving the file " + relatName + "!!!");
    return;
}
} catch (Exception e) {
    ToolConstants.reportException(e, ToolConstants.STDERR );
    return;
}
} else {
    usage();
}
return;
}
}
```