# Teste de Software Orientado a Objetos: Técnicas, Estratégias e Ferramentas

**Juliana Silva Herbert***
**Ana Maria de Alencar Price***

## Resumo

A determinação da ausência de erros em um sistema de software é impossível de ser garantida, portanto, o teste de software deve ser realizado para aumentar a confiabilidade do produto. Sistemas Orientados a Objetos (OO) são mais complexos para testar do que sistemas procedimentais, devido a características tais como herança, encapsulamento, polimorfismo e ligação dinâmica. Há várias técnicas de teste de software OO propostas na literatura, sendo as mais exploradas as relacionadas ao teste de estados dos objetos. O conhecimento e a adoção de técnicas de teste permitem que esta atividade seja feita de forma sistemática, automatizada e, portanto, menos suscetível à ocorrência de erros. A complexidade e o tamanho dos sistemas de software exigem que haja uma maior interação entre as pessoas envolvidas no processo de desenvolvimento (assim como no teste). Desta forma, é necessário que as tarefas que compõem a atividade de teste sejam distribuídas aos componentes de uma equipe, devendo haver procedimentos específicos para tal. O principal objetivo deste tutorial é apresentar técnicas, estratégias e ferramentas que podem ser utilizadas no teste de software OO.

## Abstract

The absence of errors in software systems is impossible to be assured, therefore software testing must be done to increase product's reliability. Object-Oriented (OO) systems are more complex to test than procedural ones, due to features such as inheritance, encapsulation, polymorphism and dynamic binding. There are many OO software testing techniques proposed on the literature, and most of them are related to object state based testing. Knowing and adopting techniques leads testing to be done in a systematic and automated way, and therefore, less susceptible to failures. Systems of high complexity and size require a bigger interaction among that software development (and testing) involved people. Therefore, testing activities must be distributed to team's members, according some established rules. The main purpose of this tutorial is to present techniques, approaches and tools which can be used to test object-oriented software.

Palavras-chave: Software Orientado a Objetos; Teste de Software; Trabalho Cooperativo; Ferramentas CASE (*Computer Aided Software Engineering*).

*{juliana | anaprice}@inf.ufrgs.br - Instituto de Informática – UFRGS - Porto Alegre, RS

# 1 Introdução

Um dos critérios indispensáveis na medição da qualidade de software é a verificação de confiabilidade e funcionalidade de sistemas [HER95b]. Apesar do conceito de qualidade não ser definido apenas com base nestes dois fatores, dificilmente um sistema será considerado portador de boa qualidade enquanto apresentar falhas na realização de suas funções. Tal idéia é verificada tanto quando fala-se na definição intuitiva de qualidade de software, como na apresentação de modelos (modelo CMM - *Capability Maturity Model*) [HUM88] ou normas de qualidade (série ISO 9000, por exemplo). Portanto, a verificação de confiabilidade e de funcionalidade do programa como critério essencial de qualidade é tratada como um consenso comum e amplamente aceito.

Também amplamente aceita está a disseminação da utilização do paradigma Orientado a Objetos (OO) para o desenvolvimento de software. Novas linguagens de programação, sistemas gerenciadores de bancos de dados e sistemas operacionais, entre outros sistemas de apoio ao desenvolvimento de software, têm implementado, mesmo que parcialmente, o paradigma OO. Aliás, o paradigma OO está naturalmente associado à produção de software de alta qualidade, já que possibilita maior e melhor entendimento do código desenvolvido, melhor mapeamento entre objetos do "mundo real" e estruturas de dados e maior reutilização, desde a análise até a codificação.

Entretanto, como apresentado anteriormente, um sistema de software somente é considerado portador de boa qualidade, quando atinge níveis satisfatórios e adequados de confiabilidade na realização de sua funcionalidade. O teste sistemático, como parte do processo de desenvolvimento de software, é uma das formas que pode ser utilizada para o aumento da garantia da confiabilidade em sistemas desenvolvidos [MYE79] [PRE95].

Teste é a atividade do ciclo de desenvolvimento de software na qual pode ser observada maior distância entre a teoria (técnicas de teste propostas na literatura) e a prática (aplicação destas técnicas) [McG96]. As técnicas propostas não são aplicadas principalmente porque o teste é visto apenas como uma atividade auxiliar no desenvolvimento de software, e não como parte do processo. Com esta visão, os esforços gastos nesta atividade são considerados "extras". Ou seja, no início de desenvolvimento de um produto de software há uma determinada previsão de custos. O custo previsto acaba se excedendo, na maioria das vezes, devido às atividades de teste e de manutenção.

Testa-se pouco e raramente de forma sistemática. Em relação ao software OO, a situação é ainda mais grave. Além do "descaso" normalmente associado ao processo de teste, tem-se associados os problemas comuns de uma área recente de pesquisa: algumas técnicas foram propostas, entretanto poucas apresentam relacionamentos entre si, integração com o processo de desenvolvimento de software, formalização e aplicação em projetos de significativa importância e tamanho. Além disso, as técnicas de teste aplicadas ao teste de software procedimental, que podem ser aplicadas no paradigma OO, não foram ainda mapeadas completamente para o novo paradigma.

Há várias técnicas de teste de software OO propostas na literatura, sendo as mais exploradas as relacionadas ao teste baseado nos estados dos objetos, tais como as propostas por Binder [BIN95] e Turner [TUR93], entre outros. Pode-se encontrar também, na literatura, critérios de seleção de casos de teste para máquinas de autômatos de estados finitos, tais como: DS [GOE70], UIO [SAB88], W [CHO78] e Wp [FUJ91], apresentados em [NAK95], que podem ser adaptados para o teste de software OO.

Entretanto, a maioria destes trabalhos é recente, não tendo sido ainda tratada com maior formalismo, servido de base para outros trabalhos correlatos, ou até mesmo implementada, com a exceção dos critérios de seleção de casos de teste para máquinas de autômatos de estados finitos. Entretanto, o mapeamento do comportamento de programas OO para autômatos de estados finitos ainda não foi completamente definido, sendo, inclusive, utilizado de forma errônea por autores como Binder [BIN95] e Turner [TUR93], quando referenciam trabalhos conhecidos na área de autômatos finitos como o de Chow [CHO78].

As técnicas de teste e depuração aplicadas ao software procedimental devem ser adaptadas ao software OO, motivo que não impede sua aplicação, mas que tem como conseqüência a exigência de maior atenção às características mais enfatizadas pelo paradigma. Características como encapsulamento, herança, polimorfismo e ligação dinâmica, do paradigma OO, já existiam de forma "simulada" no paradigma procedimental, entretanto, as técnicas de teste procedimental não as consideravam, em geral, já que eram pouco utilizadas e poderiam ser evitadas. No paradigma OO, tais características são intrínsecas ao modelo de desenvolvimento, devendo, portanto, ser consideradas para a real efetividade do teste.

Todas as questões anteriormente levantadas foram utilizadas como motivação para a composição deste tutorial. O texto está organizado da seguinte forma: na presente seção, foram apresentados os vários aspectos relacionados ao teste de software OO, justificando a proposição e estudo de técnicas, estratégias e ferramentas de teste; na Seção 2 são apresentadas as principais técnicas de teste de software OO, sendo algumas delas oriundas do paradigma procedimental, enquanto que outras específicas do paradigma OO; na Seção 3 são apresentadas algumas estratégias de teste de software, que têm o principal objetivo de organizar a aplicação das técnicas; já na Seção 4 são apresentadas algumas ferramentas de teste que implementam tanto as técnicas quanto estratégias de teste aqui apresentadas; a Seção 5 apresenta questões relativas ao gerenciamento da atividade de teste e, finalmente, na Seção 6, são apresentadas conclusões relacionadas a este trabalho.

## 2 Técnicas de Teste de Software OO

Tanto o teste de software quanto a depuração são atividades relacionadas diretamente à validação de software. A "validação" é freqüentemente confundida com a "verificação de software". A verificação garante que o software implementa corretamente uma função específica, enquanto que a validação garante que o software que foi construído é adequado aos requisitos do cliente. Boehm [BOE78] diferencia as duas atividades através de duas perguntas:

VERIFICAÇÃO: "*Estamos construindo certo o produto?*"

VALIDAÇÃO: "*Estamos construindo o produto certo?*"

Um dos livros mais clássicos sobre teste de software, escrito em 1979 por Myers [MYE79], apresenta o objetivo principal desta atividade como sendo o de encontrar erros: "*A atividade de teste não pode mostrar a ausência de 'bugs'; ela só pode mostrar se defeitos de software estão presentes*" [MYE79] [PRE95].

As seções a seguir apresentam as principais categorias de técnicas de teste consideradas tanto em pesquisas quanto na implementação de ferramentas comerciais. A apresentação destas categorias utiliza a linguagem de programação Java para a codificação de exemplos, quando necessário [HER99a]. A Seção 2.1 apresenta critérios utilizados no teste estrutural de um programa, enfatizando a análise de cobertura do mesmo. Na Seção 2.2, são apresentados conceitos básicos relacionados ao teste funcional, onde o programa é visto como uma caixa-preta. Finalmente, a Seção 2.3 apresenta uma técnica de teste OO, baseada em estados, e uma abordagem para a integração de métodos de uma mesma classe.

## 2.1 Teste Estrutural

O teste estrutural também é conhecido como teste caixa-branca (*white-box*) ou teste caixa aberta. No teste estrutural, os casos de teste são derivados a partir da análise da estrutura interna do programa. O objetivo dos casos de teste é causar a execução de caminhos identificados no programa por critérios baseados no fluxo de controle e/ou no fluxo de dados. A ferramenta *JavaScope*, da *SunTest Suite*, que será apresentada na Seção 4 deste tutorial, implementa critérios de teste estrutural.

Os principais problemas desta abordagem são:

- programas com laços (repetições) possuem um número infinito de caminhos, já que a análise da estrutura interna do programa é feita sobre o grafo de fluxo de controle do programa estaticamente. Desta forma, seria necessário aplicar o teste exaustivo (teste com a submissão de todas as entradas possíveis e suas combinações), o que é considerado impraticável [MYE79];
- existência de caminhos não executáveis (*infeasible paths*) no programa, os quais ocasionam o desperdício de tempo e recursos financeiros na tentativa de gerar casos de teste que possam executar estes caminhos [VER94] [VER97];
- a execução bem sucedida de um caminho do programa selecionado não garante que este esteja correto, já que com outro caso de teste um erro pode ocorrer.

A fim de minimizar os problemas acima apresentados, são utilizados critérios de seleção de caminhos, que podem ser divididos em dois grupos, de acordo com as características nas quais mais se baseiam: fluxo de controle e fluxo de dados. Estes critérios, conhecidos como "critérios de cobertura", ou "critérios de seleção", são condições que devem ser preenchidas pelo teste, as quais selecionam determinados caminhos que visam cobrir o código ou a representação gráfica deste. A seguir são

apresentadas as definições dos principais critérios de cada categoria. Utilizando como base uma pequena aplicação Java [CHA99], cujo código é apresentado na Figura 1, são apresentados os caminhos necessários para satisfazer os critérios baseados no fluxo de controle. A Figura 2 apresenta o grafo de fluxo de controle (GFC) do método *bsearch*, apresentado na Figura 1.

```
/*******************************************************
 *  PESQUISA BINARIA - Exemplo de teste de metodo      *
 *                                                     *
 *  FONTE: "Java - 1001 Dicas de Programacao"          *
 *         Mark Chan et al - Makron Books - 1999       *
 *******************************************************/
 public class arrayBinary
 {
 public static int bsearch(int array[], int value) {
   boolean found = false;
   int high      = array.length - 1;
   int low       = 0;
   int cnt       = 0;
   int mid       = (high+low)/2;

   System.out.println("Looking for " + value);
   while (!found && (high >= low)) {
     System.out.println("Low " + low + "    Mid " + mid);
     if (value == array[mid]) {
       found = true;
     }
     else if (value < array[mid]) high = mid - 1;
          else low = mid + 1;
     mid = (high+low)/2;
     cnt++;
   }
   System.out.println("Steps " + cnt);
   return((found) ? mid: -1);
}

public static void main(String[] args) {
  int array[] = new int[100];
  for (int i=0; i < array.length; i++) {
    array[i] = i;
  }
  System.out.println("Results " + bsearch(array,67));
  System.out.println("Results " + bsearch(array,33));
  System.out.println("Results " + bsearch(array,1));
  System.out.println("Results " + bsearch(array,1001));
}
}
```

Figura 1. Exemplo de aplicação Java para a exemplificação de critérios.

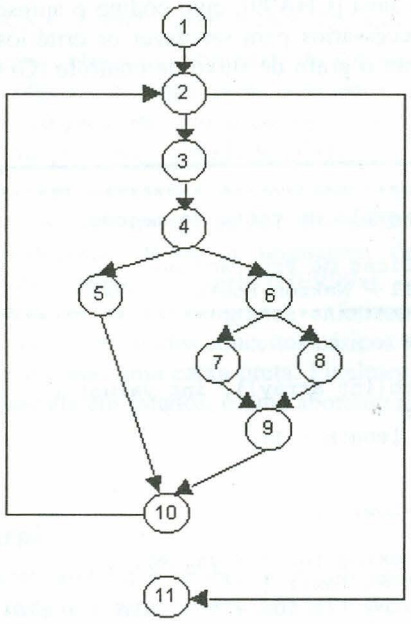


Figura 2. Grafo de fluxo de controle do método "*bsearch*" da Figura 1.

Os critérios de cobertura de código de unidade (uma rotina, no paradigma procedimental, e um método, no paradigma OO), dividem-se nas duas categorias citadas anteriormente: baseados em fluxo de controle e baseados em fluxo de dados. Estas duas categorias são a seguir apresentadas.

### ⇨ Critérios Baseados no Fluxo de Controle

Os critérios de cobertura baseados no fluxo de controle fundamentam-se na seleção de um conjunto $C$ de caminhos no Grafo de Fluxo de Controle (GFC) do programa em teste. Um caminho em um GFC é uma seqüência de nodos $(n_1, n_2,...,n_k)$, $k \geq 2$, tal que exista um arco de $n_i$ para $n_{i+1}$ para $i = 1,2,...,k-1$. Exemplos de critérios baseados no fluxo de controle da unidade são listados a seguir, sendo identificado quando estes são satisfeitos [MYE79] [RAP85]:

- COBERTURA POR NODOS (OU COMANDOS): todos os nodos/comandos devem ser executados pelo menos uma vez.
- COBERTURA POR ARCOS (OU DECISÕES): todos os arcos/decisões devem ser executados pelo menos uma vez.

- COBERTURA DE TODOS OS CAMINHOS: todos os caminhos possíveis devem ser executados pelo menos uma vez. Em programas com laços (repetições), o número de caminhos necessário para satisfazer este critério seria infinito, já que o GFC não fornece informações sobre o critério de término da iteração. Para evitar tal problema, são selecionados apenas dois caminhos relacionados a cada laço do programa: um que não executa o corpo do laço, e outro que o executa apenas uma vez. Tal restrição torna a aplicação do critério possível, e pode ser visualizada no exemplo a seguir.

Com base na Figura 2, os seguintes caminhos satisfariam os critérios anteriormente definidos:

- TODOS-NODOS:

✓ (1, 2, 3, 4, 5, 10, 2, 11)
✓ (1, 2, 3, 4, 6, 7, 9, 10, 2, 11)
✓ (1, 2, 3, 4, 6, 8, 9, 10, 2, 11)

- TODOS-CAMINHOS – COM RESTRIÇÃO DE SELEÇÃO EM RELAÇÃO AO LAÇO:

✓ (1, 2, 3, 4, 5, 10, 2, 11)
✓ (1, 2, 3, 4, 6, 7, 9, 10, 2, 11)
✓ (1, 2, 3, 4, 6, 8, 9, 10, 2, 11)
✓ (1, 2, 11)

- TODOS-ARCOS:

✓ (1, 2, 3, 4, 5, 10, 2, 11)
✓ (1, 2, 3, 4, 6, 7, 9, 10, 2, 11)
✓ (1, 2, 3, 4, 6, 8, 9, 10, 2, 11)

### ⇨ Critérios Baseados no Fluxo de Dados

Os critérios de fluxo de dados baseiam-se na análise de fluxo de dados do programa. A análise de fluxo de dados é utilizada tradicionalmente na implementação de compiladores, para a otimização de código. Este tipo de análise considera as relações entre definições e usos de variáveis, preenchendo algumas lacunas deixadas pelos critérios baseados no fluxo de controle [RAP85]. Para o teste de unidade, o GFC é estendido, acrescentando-se aos nodos e arcos as relações de definições e usos de variáveis, gerando um *grafo def/uso* (definições/usos). Um caminho $(i,n_1,...,n_m,j)$, $m \geq 0$, do GFC, que não contenha uma definição de uma variável nos nodos $n_1,...,n_m$ é chamado de *caminho livre de definição* com respeito a (c.r.a) $x$ do nodo $i$ ou nodo $j$ e do nodo $i$ ao arco $(n_m,j)$.

Um nodo $i$ possui uma definição global de uma variável $x$ caso ocorra uma definição de $x$ no nodo $i$ e exista um caminho livre de definição de $i$ para algum nodo ou para algum arco que contenha um c-uso ou um p-uso, respectivamente, da variável $x$. Um c-uso da variável $x$ em um nodo $j$ é um c-uso global se não existir uma definição de $x$ no nodo $j$ precedendo este c-uso. Caso contrário, este é um c-uso local. O conjunto de definições globais associadas a um nodo $i$ é denominado *def(i)*. O de c-usos globais, *c-*

*usos(i)*, e o conjunto de p-usos associado a um arco *(i,j)* é denominado *p-usos(i,j)*. Seja *x* uma variável, tal que $x \in def(i)$. *dcu(x,i)* é o conjunto de todos os nodos *j* tais que $x \in$ *c-usos(j)* e há um caminho livre de definições c.r.a *x* de *i* para *j*. *dpu(x,i)* é o conjunto de todos os arcos *(j,k)* tais que $x \in$ *p-uso(j,k)* e há um caminho livre de definições c.r.a *x* de *i* para *j*. Vários critérios foram propostos por Rapps e Weyuker [RAP85]. Três destes critérios são apresentados a seguir, como exemplos, onde *G* representa um grafo definição/uso e *P* um conjunto de caminhos completos de *G*:

- TODAS-DEFINIÇÕES: *P* satisfaz este critério se para todo nodo *i* de *G* e para todo $x \in$ *def(i)*, *P* inclui um caminho livre de definições c.r.a *x* de *i* para algum elemento de *dcu(x,i)* e *dpu(x,i)*.
- TODOS-P-USOS: *P* satisfaz este critério se para todo nodo *i* de *G* e para todo $x \in$ *def(i)*, *P* inclui um caminho livre de definições c.r.a *x* de *i* para todos os elementos de *dpu(x,i)*.
- TODOS-USOS: *P* satisfaz este critério se para todo nodo *i* de *G* e para todo $x \in$ *def(i)*, *P* inclui um caminho livre de definições c.r.a *x* de *i* para todos os elementos de *dcu(x,i)* e para todos os elementos de *dpu(x,i)*.

Em [MAL91], Maldonado apresenta o conceito de *potencial-uso*. A relação definição-uso necessária nos outros critérios baseados no fluxo de dados é caracterizada sem a necessidade de ocorrência de um uso. Com a utilização do conceito de *potencial-uso*, Maldonado propõe dois critérios: todos-potenciais-usos e todos-potenciais-du-caminhos. Estes critérios não serão apresentados com maiores detalhes neste tutorial, por não estarem implementados nas ferramentas de teste apresentadas na Seção 4.

## 2.2 Teste Funcional

As técnicas de teste funcional derivam os casos de teste a partir da análise da funcionalidade (dados de entrada/saída e especificação) do programa, sem levar em consideração a estrutura interna do mesmo [MYE79] [PRE95]. O teste funcional também é conhecido como teste de caixa preta (*black box*), e é implementado na ferramenta *JavaSpec*, da *SunTest Suite*, apresentada na Seção 4 deste tutorial.

A abordagem funcional tem o objetivo de complementar o efeito das técnicas do teste estrutural. As categorias de erros mais evidenciadas pelo teste funcional são: erros de interface, funções incorretas ou ausentes, erros nas estruturas de dados ou no acesso a bancos de dados externos, erros de desempenho e erros de inicialização e término [PRE95]. O teste funcional é geralmente aplicado quando todo ou quase todo o sistema já foi desenvolvido. Entre as técnicas mais clássicas de teste funcional podem ser citadas as seguintes [MYE79]:

- o grafo causa-efeito, onde são representadas as principais entradas do programa, suas combinações e saídas correspondentes geradas;
- o particionamento de equivalência, onde o domínio dos valores de entrada é dividido em classes de equivalência, reduzindo, assim, os dados de entrada a serem submetidos

ao programa, já que todos os dados de uma mesma classe supostamente teriam um comportamento equivalente;

- a análise do valor-limite, onde além de dividir o domínio dos valores de entrada em classes de equivalência, procura-se escolher dados localizados nos limites destas classes, aumentando assim a probabilidade de identificação de erros.

## 2.3 Teste de Software Orientado a Objetos

O teste de software OO apresenta algumas facilidades em relação ao teste de software procedimental [McG96]:

- as interfaces de classes e métodos estão bem definidas e explícitas;
- o número reduzido de parâmetros implica um número menor de casos de teste para sua cobertura;
- a herança sugere uma maneira natural de reutilizar casos de teste.

Entretanto, desvantagens existem e devem ser consideradas, na prática [McG96]:

- o encapsulamento de informações dificulta a avaliação da correteza da classe;
- os múltiplos pontos de entrada (métodos) de uma classe dificultam o gerenciamento do teste;
- o polimorfismo e a ligação dinâmica expandem as possíveis interações entre objetos.

No teste de software OO, devido à explícita separação entre especificação e implementação da classe, casos de teste funcional podem ser gerados separadamente de casos de teste que requeiram conhecimento do código da aplicação (teste estrutural). Pode-se testar todas as classes, mas não todos os objetos. O gerenciamento de mensagens entre objetos é similar ao de chamadas de rotinas, entretanto, a troca de mensagens ocorre mais freqüentemente. Algumas definições e declarações são reutilizadas em vários níveis da árvore de herança, interagindo com novas definições e declarações.

De acordo com McGregor [McG96], a especificação de sistemas de software OO deve conter os seguintes itens, a fim de verificar sua completeza:

- a especificação de todos os objetos: que enviam e/ou recebem mensagens, que são utilizados como parâmetros de entrada e/ou saída e os objetos de exceção;
- a especificação das pré-condições, pós-condições e invariantes de um método;
- a especificação de todos os métodos de uma classe e seu modelo de estados. Este modelo fornece informações de como instâncias da classe reagem a estímulos específicos;

A partir das informações da especificação, deve-se escrever um plano de teste. O plano deve especificar a extensão do teste, ferramentas a serem utilizadas, critérios de teste, estimativa do tempo necessário para o teste, descrição dos casos de teste e dados de teste associados. Para a obtenção dos casos de teste, a seguinte seqüência de seleção é

proposta por McGregor [McG96]:

1. desenvolvimento de um conjunto de teste funcional que cubra a especificação completa da classe;
2. desenvolvimento de casos de teste baseados em estados até que todas as transições no modelo dinâmico sejam cobertas;
3. utilização de uma ferramenta de cobertura de teste;
4. desenvolvimento de casos de teste estruturais adicionais para a cobertura de cada linha de código;
5. desenvolvimento de casos de teste para testar as interações entre métodos de uma mesma classe;
6. desenvolvimento de casos de teste para a cobertura das interações entre objetos da classe sendo testada e de outras classes.

As subseções 2.3.1 e 2.3.2 apresentam duas abordagens diferentes de realização do teste de software OO. A primeira refere-se ao teste de estados, na qual são avaliadas as várias mudanças de estados pelas quais passam os objetos de determinada classe. O teste baseia-se no modelo dinâmico da classe (diagrama/máquina de estados), sendo formado por estados, transições (execução de métodos), pré e pós-condições associadas. Já a Subseção 2.3.2 apresenta uma abordagem de teste de integração entre métodos de uma mesma classe, na qual é utilizada a hierarquia de herança de classes para a reutilização e/ou criação de casos de teste funcionais e estruturais para o teste de cada método.

### 2.3.1 Teste de Estados

De acordo com Binder [BIN95], o comportamento de uma classe é definido por determinado conjunto de valores encapsulados, que a classe possui em determinado momento. O comportamento da classe é controlado por valores encapsulados, seqüências de mensagens, ou ambos. O objetivo do teste de estados é testar o sistema OO sem tentar todas as combinações possíveis. E este teste deve ser suficientemente confiável a fim de selecionar combinações que testem comportamentos representativos do conjunto original.

A abordagem apresentada por Binder, em [BIN95] é a base da estratégia de teste FREE (*Flattened Regular Expression*). Esta estratégia baseia-se no modelo dinâmico (máquina de estados finitos) apresentado por várias metodologias de análise OO.

Um estado é definido como sendo um subconjunto do conjunto de todas as combinações possíveis dos valores de atributos da classe. Para as atividades de projeto e de teste, pode-se definir um estado como um subconjunto de valores que podem ser combinados, e que compartilham alguma propriedade de interesse.

Por exemplo, o saldo de uma conta-corrente pode ser considerado como um atributo base para a realização do teste de estados. Caso qualquer variação no saldo seja considerada uma mudança de estado, a máquina de estados terá um tamanho tal que será

praticamente impossível monitorar e analisar qualquer caminho executado. Para reduzir tal complexidade, e tornar a representação de máquina de estados mais adequada ao teste, neste exemplo, poderiam ser selecionados apenas os estados "conta negativa" (caso o saldo seja menor do que zero), "conta zerada" (caso o saldo seja igual a zero) e "conta positiva" (caso o saldo seja maior do que zero). Ou seja, a semântica da aplicação deve ser considerada, o que consiste em uma das maiores dificuldades do teste baseado em estados.

Os estados podem ser definidos por expressões booleanas. Por exemplo, o estado "conta positiva" pode ser definido como: *saldo* > *0*. As expressões de pré-condições e de pós-condições definem o contrato da classe. As classes servidor (as que recebem as mensagens) devem definir pré-condições para que uma mensagem seja aceita, enquanto que as classes cliente (as que enviam as mensagens) devem definir pós-condições para definir os resultados. Uma pós-condição também define o estado obtido pela ativação do método. Há, no mínimo, um estado para cada pós-condição de uma classe. Se houver *n* operadores lógicos *ou* na expressão de pós-condição, então o método pode computar *n+1* estados.

A seguir, é apresentado um exemplo transcrito de [BIN95], que descreve operações que podem ser realizadas sobre uma conta bancária. O formato (*template*) da classe *Conta* é apresentado na Figura 3.

A seguinte especificação controla as operações feitas sobre *Conta*:

- uma conta aberta aceita todas as transações, com exceção de zerar;
- quando o saldo é negativo, a conta está negativa;
- uma conta negativa pode aceitar uma consulta de saldo, créditos e débitos que não sejam feitos pelo cliente;
- quando uma conta está bloqueada, ela pode ser apenas liberada ou ter seu saldo consultado. Nenhum outro tipo de transação pode ser realizada;
- se sobre uma conta não for realizada nenhuma transação durante cinco anos, ela se torna inativa. Somente uma conta inativa pode aceitar a transação zerar. Esta transação tem como conseqüência direta o fechamento da conta;
- uma conta deve estar zerada para ser fechada. Após seu fechamento, nenhuma transação pode ser efetuada.

A máquina de estados correspondente à classe *conta* é apresentada na Figura 4. Cada transição é legendada com o nome do método correspondente. Expressões entre colchetes indicam condições para que a transição ocorra, podendo representar uma pré-condição ou uma pós-condição. A partir do diagrama de estados, é gerada uma árvore de transição. Para tal, o estado inicial da máquina é transformado na raiz da árvore. Para cada transição, um arco é desenhado do nodo origem da transição para o nodo que representa o estado resultante. Este procedimento é repetido para cada nodo de estado resultante, até que o estado resultante já apareça na árvore como um nodo anterior ou o nodo resultante é um estado final. O atributo saldo é representado no exemplo por "*b*".

| | | |
|---|---|---|
| NOME: | Conta | |
| SUPERCLASSE: | Objeto | |
| OPERAÇÕES: | **Saldo** | - fornece o saldo da situação atual da conta. |
| | **Crédito** | - adiciona o valor de crédito à conta. |
| | **Débito** | - subtrai o valor de débito da conta. |
| | **Abre** | - cria uma conta. |
| | **Bloqueia** | - suspende transações sobre a conta. |
| | **Libera** | - suspende bloqueio. |
| | **Zera** | - zera o extrato da conta. |
| | **Fecha** | - finaliza todas as atividades da conta. |

Figura 3. *Template* da classe *Conta*.

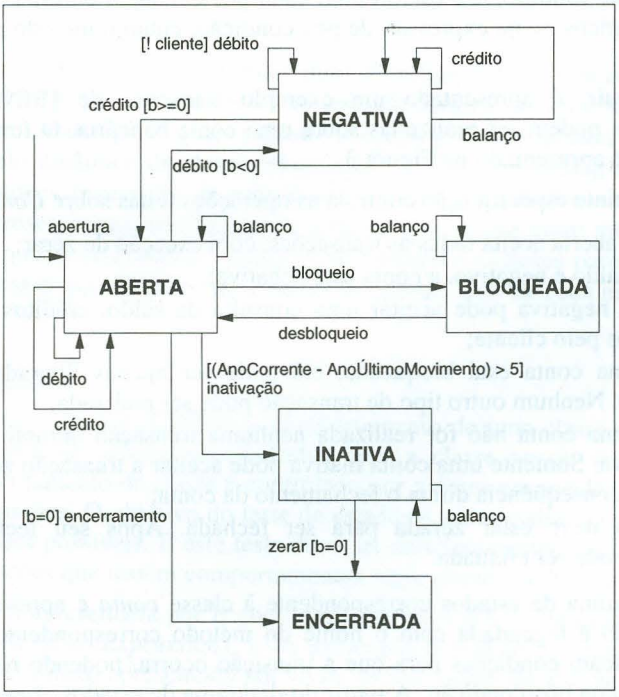


Figura 4. Máquina de estados da classe *conta*.

A próxima etapa consiste em transcrever seqüências de teste de transições a partir da árvore. Cada arco ou cada seqüência de arcos orientada em uma mesma direção consiste em um caso de teste. O plano de teste é completado identificando-se valores de parâmetros de métodos, estados esperados e exceções. O teste é executado inicializando-se o objeto com o estado inicial, aplicando a seqüência e comparando o estado resultante

com o estado esperado. Erros encontrados: transições perdidas, transições incorretas, ações de saída incorretas e estados incorretos. Este método de teste pode ser parcialmente utilizado com a ferramenta *AssertMate*, apresentada na Seção 4. Esta ferramenta permite a utilização de assertivas no meio do código, que podem representar pré e pós-condições, verificadas durante a execução do código instrumentado.

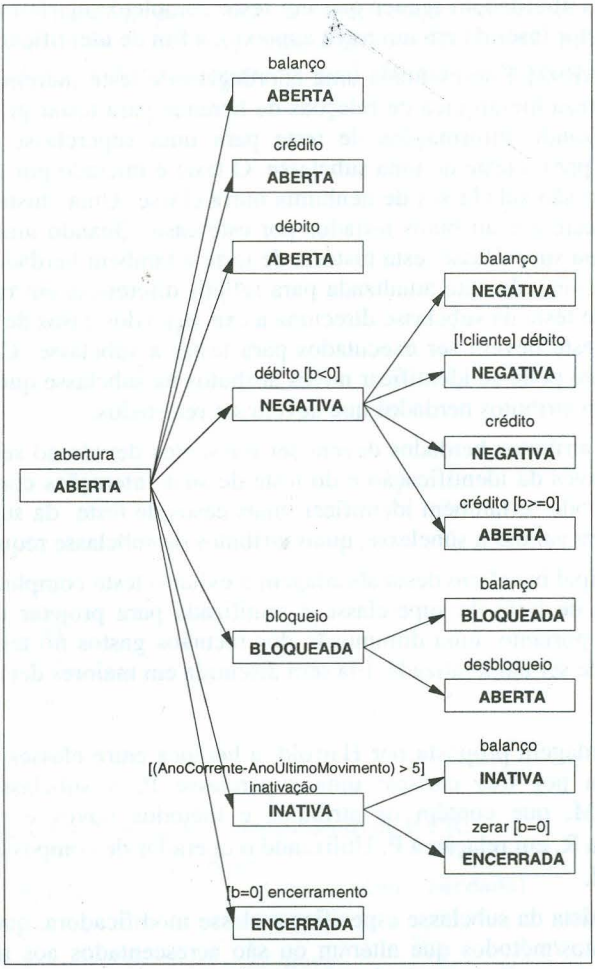


Figura 5. Árvore de transição da classe *conta*.

### 2.3.2 Teste Incremental - Harrold

A abordagem de teste incremental de classes, aqui apresentada, foi proposta por Harrold et al., em [HAR92]. O objetivo da abordagem é apresentar uma forma de testar classes isoladamente, de forma incremental, reutilizando casos de teste, na hierarquia de

herança, sempre que possível. É utilizada uma abordagem *top-down*, na qual as superclasses são testadas inicialmente.

O maior objetivo do paradigma OO é a criação de classes e/ou frameworks altamente reutilizáveis e confiáveis. Para tal, é necessário que estas classes sejam testadas. Uma das abordagens utilizadas é validar cada classe em uma biblioteca, individualmente. Entretanto, esta abordagem requer que um teste completo seja feito novamente, sempre que esta classe for inserida em um novo contexto, a fim de identificar erros de integração.

Em [HAR92] é apresentada uma abordagem de teste incremental de classes, que explora a natureza hierárquica de relações de herança para testar grupos relacionados de classes, reutilizando informações de teste para uma superclasse e incrementalmente atualizando-as para o teste de uma subclasse. O teste é iniciado por classes-base, que são classes que não são subclasses de nenhuma outra classe. Uma "história de teste" associa cada caso de teste aos atributos testados por este caso. Quando uma subclasse herda os atributos de uma superclasse, esta história de teste é também herdada. A história de teste herdada é incrementalmente atualizada para refletir diferenças em relação à superclasse. Uma história de teste da subclasse direciona a execução dos casos de teste, pois ela indica que casos de teste devem ser executados para testar a subclasse. Com esta abordagem, automaticamente pode-se identificar novos atributos na subclasse que devem ser testados, juntamente com atributos herdados que devem ser retestados.

Alguns atributos herdados devem ser retestados devido ao seu novo contexto (da subclasse), através da identificação e do teste de suas interações com os atributos novos da subclasse. Pode-se também identificar quais casos de teste da superclasse podem ser reutilizados para validar a subclasse, quais atributos da subclasse requerem novos casos.

O principal benefício desta abordagem é evitar o teste completo de cada subclasse, pois a história de teste da superclasse é reutilizada para projetar os casos de teste da subclasse. Há, portanto, uma diminuição dos recursos gastos no teste. Além disso, esta abordagem pode ser automatizada. Ela será discutida em maiores detalhes a seguir.

### ⇨ Herança

Na abordagem proposta por Harrold, a herança entre classes é considerada como sendo formada por três classes: uma superclasse P, a subclasse R e uma classe modificadora M, que contém os atributos e métodos novos e redefinidos a serem acrescentados a R, em relação a P. Utilizando o operador de composição $\oplus$, pode-se dizer que $R = P \oplus M$.

O projetista da subclasse especifica a classe modificadora, que deve conter vários tipos de atributos/métodos que alteram ou são acrescentados aos atributos/métodos da superclasse. Estes atributos/métodos (representados a seguir pela letra A) podem ser classificados como:

- NOVO: A foi definido em M, mas não em P; ou A está em M e P, mas os argumentos de A diferem entre M e P. Neste caso, A refere-se ao atributo ou método definido na classe resultante R, mas não em P.
- RECURSIVO (herdado): A é definido em P, mas não em M. Neste caso, A refere-se ao

atributo ou método localmente definido em P e que fica disponível em R.

- REDEFINIDO: A é definido em P e M, e a lista de argumentos de A é a mesma em P e M. Neste caso, A refere-se à definição do atributo ou método em M.

A relação de herança também determina a visibilidade, a disponibilidade e o formato dos atributos e métodos de P em R. Uma linguagem pode suportar mais de um tipo de mapeamento de herança, permitindo a especificação da visibilidade, como por exemplo em Java, com os tipos: *public, private* e *protected*. Um exemplo dos tipos de atributos/métodos herdados é dado a seguir, na qual são apresentadas as definições das classes P e R:

```
class P
{
  private int i;
  private int j;
  public  P()
     {}
  public void A(int a,int b)
    {i=a;
     j=a+2*b;}
  public int B()
    {return i;}
  public int C()
    {return j;}
};

class R extends P
{
  private float i;
  public R()
    {}
  public void A(int a)
    {super.A(a,0);}
  public int B()
    {return 3*super.B();}
  public int C()
    {return 2*super.C();}
};
```

Atributos e métodos de R, após o mapeamento de P e R:

```
private float i;                  // novo
public void A(int a,int b)              // recursivo (herdado)
    {i=a;
     j=a+2*b;}
void A(int a)                           // novo
    {super.A(a,0);}
void int B()                            // redefinido
    {return 3*super.B();}
int C()                                 // redefinido
    {return 2*super.C();}
```

### ⇨ Teste Incremental Hierárquico de Classes

Cada classe é testada isoladamente, através do teste das interações entre os métodos. As informações relativas ao projeto e à execução dos casos de teste são armazenadas na "história de teste". Quando uma subclasse é definida, a história de teste de sua superclasse, a definição do modificador e a relação de herança são utilizadas para determinar que métodos devem ser testados e que casos de teste da superclasse podem ser reutilizados. A abordagem é hierárquica porque é direcionada à ordenação parcial da relação de herança; e é incremental, porque utiliza resultados do teste de um nível da hierarquia para reduzir os esforços necessários pelos níveis subseqüentes.

### ⇨ Teste das Superclasses

Inicialmente, as superclasses são testadas utilizando-se técnicas de teste de unidade tradicionais, para testar métodos isoladamente na classe. Cada método é testado utilizando um conjunto de casos de teste funcional e estrutural. Estes testes requerem o uso de *drivers* e *stubs*, os quais simulam o programa que chama o método sendo testado e os métodos chamados [HER95a]. Para o teste funcional, pode ser utilizado o critério de partição do domínio de entrada [MYE79], por exemplo, enquanto que para o teste estrutural, podem ser utilizados os critérios de seleção de caminhos orientados ao fluxo de dados [RAP85].

A história de teste da superclasse contém associações entre cada método da classe e os casos de teste, no formato de triplas $\{m_i, (TS_i,test?), (TP_i,test?)\}$, onde $m_i$ é o método, $TS_i$ é o caso de teste funcional, $TP_i$ é o caso de teste estrutural e "test?" indica se o caso de teste deve ser resubmetido ou não.

As interações entre classes são testadas através de um "grafo de classes", no qual cada nodo representa um método na classe ou um atributo, e cada arco representa uma mensagem. Para o teste de integração das classes, os atributos e métodos indicados pelo grafo são combinados e casos de teste são desenvolvidos para testar as interfaces. Recomenda-se que o grafo seja dividido em subgrafos, para que a complexidade diminua. Assim sendo, a segunda parte da história de teste também consiste de triplas $\{m_i, (TIS_i,test?), (TIP_i,test?)\}$, onde $m_i$ é o método, $TIS_i$ é o caso de teste estrutural, $TIP_i$ é o caso de teste funcional e "test?" indica se o caso de teste deve ser resubmetido ou não. "test?" pode assumir três valores: Y, se deve ser totalmente resubmetido, P, se deve ser parcialmente e N se não necessita ser submetido.

O teste entre classes é oriundo das interações que ocorrem quando métodos de uma classe interagem com métodos de outra classe. Um exemplo é apresentado em [HAR92], onde a classe *Shape* é uma classe abstrata definida para a criação de classes de várias formas gráficas. Cada forma tem um ponto de referência que é utilizado para localizar a posição onde a forma é desenhada no sistema de coordenadas do programa.

A classe *Shape* define vários métodos que descrevem o comportamento de uma forma e inclui um método *draw()*, com implementação vazia associada ({ }) que estabelece uma interface comum para todas as classes na estrutura de herança. O método *move_to()* é definido com base nos métodos da classe *Shape*, assim como o método

*erase()*. A Tabela 1 apresenta a história de teste para a classe *Shape*.

Tabela 1. História de teste para a classe *Shape*.

| MÉTODO | CASO DE TESTE FUNCIONAL | CASO DE TESTE ESTRUTURAL |
|---|---|---|
| ***teste de unidade*** | | |
| *put_reference_point* | $(TP_1, Y)$ | $(TS_1, Y)$ |
| get_reference_point | $(TP_2, Y)$ | $(TS_2, Y)$ |
| move_to | $(TP_3, Y)$ | $(TS_3, Y)$ |
| erase | $(TP_4, Y)$ | $(TS_4, Y)$ |
| draw | $(TP_5, Y)$ | ---- |
| area | $(TP_6, Y)$ | $(TS_6, Y)$ |
| shape | $(TP_7, Y)$ | $(TS_7, Y)$ |
| shape | $(TP_8, Y)$ | $(TS_8, Y)$ |
| ***teste de interação*** | | |
| *move_to* | $(TIP_9, Y)$ | $(TIS_9, Y)$ |
| *erase* | $(TIP_{10}, Y)$ | $(TIS_{10}, Y)$ |

A classe *Shape* é a superclasse, e como tal, deve-se testar cada um de seus métodos com definições disponíveis. Um caso de teste funcional para o método *draw()* pode ser gerado, mas não um caso estrutural, já que a implementação não está disponível.

### ⇨ Teste das Subclasses

O algoritmo para o teste das subclasses é apresentado a seguir, de forma textual. O algoritmo usa uma abordagem incremental que transforma a história de teste da superclasse *P* na história de teste da subclasse *R*.

Cada método *A novo* deve ser completamente testado, já que não foi definido em *P*. *A* deve ser testado no contexto da subclasse *R*, interagindo com os outros métodos da classe. Quando *A* for herdado por outra classe, apenas o teste de integração será necessário. Para sinalizar que *A* deverá ser testado, deve-se indicar este fato na história de teste correspondente, atribuindo ao campo "test?" o valor "Y" (sim, deve-se retestá-lo).

Um método *recursivo* ou *herdado A* requer o reteste parcial, já que ele já foi individualmente testado em *P*. Se *A* interage apenas com outros métodos recursivos, o teste de interação também não precisa ser refeito. Entretanto, se *A* interage com métodos novos, ou acessa as mesmas instâncias na representação da classe como outros métodos, estas interações devem ser retestadas. Como as interações de *A* são testadas quando os novos métodos são acrescentados à classe, não há necessidade de fazer testes explícitos.

A herança mapeia *P* para *R*, podendo modificar a visibilidade de um atributo ou método. Por exemplo, se um atributo foi movido de um nível visível a um nível escondido, então ele não pode interagir com um atributo novo ou redefinido. Se um atributo é visível por qualquer método definidos em *R*, então as interfaces entre os novos

métodos e os métodos existentes que acessam os atributos devem ser testadas. Um método *A* redefinido em *M* requer reteste extensivo, entretanto os casos de teste funcionais podem ser reutilizados, pois somente a implementação foi modificada.

## 3 Estratégias de Teste de Software

As técnicas de teste apresentadas na Seção 2 devem ser utilizadas em conjunto organizadas em estratégias de teste, através das quais pode-se estabelecer como, em que ordem e quem realizará cada tarefa. De acordo com Pressman [MYE79], "uma estratégia de teste de software integra técnicas de projeto de casos de teste numa série bem-definida de passos que resultam na construção bem-sucedida de software". Algumas considerações gerais sobre estratégias de teste são apresentadas a seguir:

- o teste deve iniciar no nível de módulos e prosseguir na direção da integração de todo o sistema;
- diferentes técnicas de teste podem ser utilizadas em diferentes momentos;
- a atividade de teste é realizada pela equipe de desenvolvimento ou por um grupo independente;
- as atividades de teste e depuração são atividades diferentes, entretanto, a depuração deve ser inserida em qualquer estratégia de teste.

Em uma situação ideal, o sistema de software deve ser testado por pessoas diferentes daquelas que o programaram. Entretanto, considerando a divisão das tarefas de teste em quatro níveis relacionados ao escopo do software sendo testado, tem-se a seguinte distribuição de responsabilidades:

- TESTE DE UNIDADE: geralmente feito pelo próprio programador, pois exige um ambiente de produção adequado e especializado para acompanhar os testes realizados;
- TESTE DE INTEGRAÇÃO: pode ser feito por vários programadores, de forma que os integrantes do grupo, que não participaram do desenvolvimento dos módulos, participem da atividade de teste como testadores ativos, e os programadores dos módulos como testadores de suporte, que auxiliarão os testadores ativos na prestação de informações que eventualmente sejam necessárias;
- TESTE DE SISTEMA: o teste de sistema deve ser feito, preferencialmente, por pessoas que não estiveram ligadas diretamente ao desenvolvimento do sistema. Pode-se utilizar, neste momento, usuários selecionados, assim como analistas de suporte de outros projetos, atendentes de telefone que registram pedidos de alterações do sistema, entre outros;
- TESTE DE ACEITAÇÃO: tipo de teste que deve ser feito, obrigatoriamente, por usuários. Exemplos de teste desta categoria são: teste *alfa* e teste *beta*.

As subseções que seguem apresentam estratégias de teste utilizadas nas três categorias de teste apresentadas previamente (unidade, integração e sistema).

## 3.1 Teste de Integração

O teste de integração dos módulos individualmente testados no teste de unidade seria o primeiro momento no qual pode existir cooperação na atividade de teste. É interessante haver, neste momento, a presença de "testadores ativos" e "testadores de suporte". O programador da classe assume o papel de testador de suporte, trabalhando como um profissional de suporte ao testador ativo.

Os testadores de suporte devem ajudar na determinação da ordem de integração dos módulos. A integração dos módulos deve ser realizada de forma incremental. A abordagem de integração não-incremental não é recomendada, já que o escopo de código a ser considerado na ocorrência de um erro é bastante grande, tornando-se difícil localizá-lo. Por isso, neste tutorial, somente a abordagem incremental será abordada.

Na abordagem incremental, devem ser construídos módulos de apoio, chamados *drivers* e *stubs*. Um *driver* é um módulo que chama o(s) módulo(s) sendo testado(s), tendo em seu corpo apenas inicializações de variáveis globais, chamadas de rotinas e inicializações dos parâmetros necessários. Um *stub* é um módulo que é chamado pelo(s) módulo(s) sendo testado(s), contendo em seu corpo apenas a atribuição de valores que serão retornados, quando for necessário. No teste incremental, a necessidade de utilização de *drivers* e/ou *stubs* é determinada através da estratégia utilizada. As duas estratégias mais utilizadas são a *top-down* e a *bottom-up*.

Na estratégia *top-down*, a integração inicia com o módulo mais do topo (o módulo inicial) do sistema. Depois disso, devem ser selecionados módulos considerando-se que seu módulo superior (o módulo chamador) tenha sido bem integrado. Para o teste de um módulo superior, é necessária a utilização de *stubs*. Os módulos *stubs* não devem conter apenas mensagens indicando que o fluxo de controle passou por eles, mas sim retornos de valores ou a realização de funções específicas, sempre que necessário. Caso houver um erro na construção dos *stubs*, poderá haver uma falha no sistema, e a detecção da mesma implicará tempo e custos perdidos [PRE95].

Um dos problemas com a estratégia *top-down*, é que o fornecimento de valores geralmente não é feito de forma direta no momento inicial, já que módulos que contêm funções de entrada e/ou saída costumeiramente estão localizados na base do diagrama de chamadas. Desta forma, os valores de entrada necessários aos módulos sendo testados devem ser fornecidos pelos *stubs*. Para um conjunto variado de valores de entrada, deve-se editar um *stub* atribuindo novos valores em suas inicializações, ou então construí-lo de tal forma que sua estrutura permita diretamente a seleção de valores para cada execução. Assim que um conjunto de módulos, com seus *stubs* é testado, cada um dos *stubs* é substituído pelo módulo real correspondente, de forma que o teste de integração termine quando o sistema todo estiver integrado e tiver as interfaces entre seus módulos bem testada.

A seqüência através da qual são integrados os módulos pode variar, dependendo do critério adotado para tal. Podem ser escolhidos primeiro os módulos mais críticos, ou os que possuem funções de entrada e/ou saída ou ainda aleatoriamente. Esta decisão vai influenciar no esforço necessário para a codificação dos *stubs*. Um problema que deve ser

evitado é a passagem para o teste de um outro módulo sem completar a integração do anterior. Algumas outras considerações sobre a escolha da seqüência são apresentadas em [HET87], [MOS93] e [MYE79].

A estratégia *bottom-up*, por sua vez, inicia selecionando-se os módulos da base do diagrama de chamadas. Para que o teste seja realizado, não é necessária a construção de módulos *stubs*, mas sim de *drivers*, para que os módulos reais do sistema sejam chamados. Os *drivers* devem conter inicializações de variáveis globais e de variáveis passadas como parâmetros nas chamadas aos módulos. Desta forma, podem ser construídas várias versões de um módulo *driver*, para permitir a submissão de vários conjuntos de valores, bem como construí-lo de maneira que esta variação seja feita de forma automática (como citado anteriormente com os módulos *stubs*).

Um dos grandes problemas da estratégia *bottom-up*, é que não é construído, inicialmente, um esqueleto do programa. O programa funcional passa a existir somente quando o último módulo é integrado. A vantagem sobre a estratégia *top-down* é que as funções de entrada e/ou saída são geralmente integradas no início do teste, fazendo com que não sejam necessárias tantas versões de *drivers* quanto de *stubs* na estratégia *top-down*.

## 3.2 Teste de Aceitação

O teste de aceituação corresponde ao teste de sistema, que deve ser realizado após o teste de unidade e o teste de integração. De acordo com Pressman [PRE95], a "validação (fase de aceitação) é bem-sucedida quando o software funciona de uma maneira razoavelmente esperada pelo cliente". As expectativas do cliente devem estar documentadas em um documento de especificação, o qual deve ter sido escrito no início do desenvolvimento do sistema.

Nesta etapa, é fundamental a especificação de um plano de teste. Este plano de teste pode ter o formato simples de um *checklist*, ou então ser formado por uma lista de procedimentos a serem realizados. Em ambos os casos, o plano deve abranger todos os requisitos funcionais e de desempenho do sistema.

Além disso, a documentação do usuário também deve ser utilizada neste momento. Os manuais e sistemas de ajuda (*on-line* ou impressos) devem estar de acordo com a funcionalidade do software. Também a inclusão de tutoriais e exemplos de complexidade média de uso do sistema são bastante úteis para que o usuário entenda como o sistema funciona. Para haver consistência entre os documentos e o próprio sistema de software, é interessante que seja realizada a revisão da configuração do software, em paralelo com o teste de aceitação.

Pressman [PRE95] afirma que, após cada caso de teste de validação ter sido realizado, existirá uma das duas condições a seguir: (1) os requisitos funcionais e de desempenho conformam-se à especificação, ou (2) um desvio das especificações é descoberto e uma lista de deficiências é criada. A seguir são apresentadas algumas estratégias para o teste de validação de um sistema.

### 3.2.1 Teste Alfa

No teste alfa, o cliente utiliza o sistema de software nas instalações do desenvolvedor. Os desenvolvedores acompanham o trabalho do cliente, anotando problemas de uso, erros e queixas em relação à percepção do cliente sobre o sistema. Ou seja, o teste alfa é conduzido em um ambiente controlado.

Um dos problemas do teste alfa é que raramente o cliente consegue comportar-se tão "naturalmente" como no seu ambiente real de trabalho, devido à presença, muitas vezes incômoda, do desenvolvedor. Uma alternativa para dispensar esta presença seria a utilização de sistemas que registram todas as interações do usuário como por exemplo o utilitário "ScreenCam", da Lotus.

### 3.2.2 Teste Beta

Já no teste beta (mais conhecido e utilizado do que o teste alfa), o cliente utiliza o sistema em suas próprias instalações. O desenvolvedor não está presente, o que acaba caracterizando uma aplicação real do cliente, já que o ambiente não pode ser controlado pelo desenvolvedor. O cliente registra todos os problemas encontrados e repassa-os para o desenvolvedor, que realiza as modificações necessárias.

Os clientes que devem ser escolhidos para o teste beta devem possuir algumas das características listadas a seguir:

- capacidade crítica no uso do software, tendo a tendência natural de querer testar situações pouco comuns ou de exceção;
- bom entrosamento com a empresa desenvolvedora, já que o cliente deve entender que a versão que ele está utilizando está em teste beta, sendo ainda passível de erros e melhoramentos. Por isso, nenhuma operação que coloque em risco procedimentos ou dados da empresa deve ser realizada com aquela versão;
- preferencialmente, deve ser leigo em informática, não fazendo assim considerações que podem levar a interpretações errôneas do erro observado;
- ser organizado e dispor de tempo para o teste, a fim de que todas as observações sejam registradas, contribuindo para o aperfeiçoamento do produto final.

## 3.3 Teste de Sistema

O teste de sistema tem o objetivo principal de pôr à prova o sistema completo de software [PRE95]. Ou seja, após realizados o teste de unidade, de integração e de validação, o sistema é testado em conjunto com outros sistemas de software e elementos de hardware.

Desta forma, os seguintes tipos de teste de sistema são discutidos nas próximas subseções: teste de segurança, teste de estresse e teste de desempenho.

### 3.3.1 Teste de Segurança

O teste de segurança tem o objetivo de garantir que o sistema se comporte adequadamente mediante tentativas ilegais de acesso, tais como as que são comumente feitas por *hackers*. Os mecanismos de segurança implementados no sistema devem protegê-lo realmente destes acessos indevidos.

Esta estratégia de teste deve ser aplicada por programadores que não desenvolveram o software. Para isto, os testadores devem tentar penetrar no sistema de várias formas, tais como através da obtenção ilegal de senhas, desarme do sistema, tentativas de acesso durante a recuperação do sistema, após inserção de falha intencional, entre outros. É importante que o teste não seja feito pelos próprios desenvolvedores, a fim de evitar que estes testem apenas os mecanismos de segurança implementados, os quais são de seu próprio conhecimento.

### 3.3.2 Teste de Estresse

O teste de estresse deve confrontar o sistema com situações anormais de uso. A pergunta que deve ser feita pelo testador é: "até que ponto podemos aumentar tal recurso antes que falhe?" [PRE95].

Para responder esta pergunta, o testador deve utilizar o sistema com recursos em quantidade, freqüência e volume anormais, tais como procura excessiva de dados em disco, aumento excessivo de índices de dados, abertura de inúmeras janelas, até que haja um problema de falta de memória, utilização de arquivos com formato não compatível aos esperados pelo programa, entre outros.

O teste de estresse também deve ser feito por programadores que não desenvolveram o sistema. Pode ser utilizado um *checklist* com situações padrão de provocação de "estresse no sistema".

### 3.3.3 Teste de Desempenho

O teste de desempenho é fundamental para sistemas de tempo real. Nesta etapa, deve-se verificar se o desempenho do software está de acordo com seus requisitos especificados, no qual o desempenho de execução do software é testado, dentro do contexto de um sistema integrado.

Pressman [PRE95] sugere que o teste de desempenho seja feito combinado com o teste de estresse. Nestas situações, software e hardware são controlados, a fim de observar seu comportamento perante às situações nas quais foram colocados.

O teste de desempenho deve ser feito por outros programadores que não sejam os que desenvolveram o sistema. Entretanto, é interessante que o próprio programador atue como um "testador de suporte", a fim de fornecer informações que possam ser necessárias.

## 3.4 Teste de Caso de Uso

Os casos de uso foram definidos por Ivar Jacobson [JAC92], tendo sido adotados pela linguagem UML (*Unified Modeling Language*), da *Rational Software Corporation* [RAT97a] [RAT97b], a qual tem sido considerada uma linguagem de modelagem padrão para a especificação de sistemas de software OO. Em UML, os casos de uso são utilizados para a representação da funcionalidade do sistema sendo modelado, em um alto nível de abstração. No teste de sistemas de software OO, os casos de uso podem ser utilizados como um *checklist* de situações a serem testadas.

De acordo com Jacobson, um caso de uso é uma maneira específica de utilização do sistema, executando parte de sua funcionalidade. Cada caso de uso é constituído por uma seqüência de eventos iniciadas por um ator (alguém ou algo que interage com o sistema). O conjunto de casos de uso especifica todas as maneiras diferentes de utilização do sistema. A utilização de casos de uso é simples e prática para a modelagem de alto nível de abstração de sistemas OO. Entretanto a sua proposta original carece de descrições objetivas de utilização dos mesmos.

Mattingly [MAT98] apresenta algumas considerações práticas para a especificação e o refinamento dos casos de uso, que, apesar de terem sido propostos originalmente para as fases de análise e projeto, podem ser utilizadas diretamente no procedimento de teste. Os seguintes objetivos para a utilização de casos de uso foram apresentados por Mattingly e podem ser observados no processo de teste em questão:

- definir, utilizando a linguagem do usuário, o sistema a ser construído com base em três visões: do usuário final, do testador e do desenvolvedor;
- casos de uso podem ser refinados em casos de teste de aceitação;
- casos de uso podem fornecer a base para a construção da documentação para o usuário final.

A fim de atingir estes três objetivos, um *template* de especificação de caso de uso foi proposto em [MAT98], sendo apresentado na Figura 6.

Os fluxos de exceção representam a seqüência de etapas que o ator segue quando uma tarefa é interrompida ou uma pós-condição que valida um caminho de exceção é satisfeita. O fluxo de execução indica a causa da interrupção, indicando também como o ator o recupera. Caminhos de exceção devem fornecer pelo menos uma pós-condição para a etapa que valida o caminho de exceção correspondente. A pós-condição final, que valida o caso de uso não será necessariamente verdadeira caso algum caminho de exceção seja executado.

Exemplos de casos de uso, para um sistema de controle de contas bancárias, seriam os seguintes:

- cliente realiza débito em conta-corrente;
- cliente realiza crédito em conta-corrente.

Os casos de uso podem ser melhor especificados através de suas variações, que

devem ser consideradas no procedimento de teste em sua totalidade. Algumas variações para estes casos de uso são apresentadas a seguir.

A partir do caso de uso "Débito", os seguintes cenários podem ser identificados:

- Cliente retira valor menor do que o disponível em conta.
- Cliente retira valor maior do que o disponível em conta, tornando a conta negativa.
- Cliente retira valor igual ao disponível em conta, tornando a conta zerada.
- Cliente tenta realizar débito em conta negativa e tem a operação negada.
- Cliente tenta realizar débito em conta inexistente e tem a operação negada.

Já a partir do caso de uso "Crédito", os seguintes cenários podem ser definidos:

- Cliente realiza depósito em conta positiva.
- Cliente realiza depósito em conta negativa, com valor igual ao módulo do montante negativo, tornando a conta zerada.
- Cliente realiza depósito em conta negativa, com valor maior do que o módulo do montante negativo, tornando a conta positiva.
- Cliente realiza depósito em conta negativa, com valor menor do que o módulo do montante negativo.
- Cliente tenta realizar depósito em conta inexistente e tem a operação negada.

*Caso de uso*: nome do caso de uso.
*Descrição*: descrição breve do objetivo do caso de uso.
*Tipo do caso de uso*: concreto ou abstrato.
*Papéis/Atores*: lista separada por vírgulas de atores ou papéis relacionados ao caso de uso.
*Pré-condições*: pré-condições que devem ser satisfeitas antes da execução do caso de uso.

*Execução básica*:
1. Realização da primeira etapa.
- *Caminho alternativo*: caminho alternativo opcional para a primeira etapa.
- *Caminho de exceção*: caminho de exceção opcional para a primeira etapa.
- *Pós-condição*: descrição opcional da pós-condição da etapa.

2. Realização da segunda etapa:
- *Caminho alternativo.*
- *Caminho de exceção.*
- *Pós-condição.*

*Execução alternativa*:
1. Realização da primeira etapa.
- *Caminho de exceção*: caminho de exceção opcional para a primeira etapa.
- *Pós-condição*: descrição opcional da pós-condição da etapa.

2. Realização da segunda etapa:
- *Caminho de exceção.*
- *Pós-condição.*

*Execução de exceção*:
1. Realização da primeira etapa.
- *Pós-condição*: descrição opcional da pós-condição da etapa.

2. Realização da segunda etapa:
- *Pós-condição.*

*Pós-condições*: pós-condições que devem ser válidas e coincidir com o objetivo do caso de uso.

*Questões a serem determinadas ou resolvidas*: tabela ou lista de questões.

*Notas*: texto em formato livre.

Figura 6. *Template* de especificação de casos de uso.

As seguintes diretivas devem ser seguidas para a definição dos casos de uso:

- nunca devem ser escritos por usuários, já que estes tendem a elaborar casos de uso complexos, descrevendo telas do sistema, dificultando sua manutenção e testabilidade;
- basear a construção dos casos em entrevistas com usuários, abordando as seguintes questões, em relação ao sistema de software:
  - problemas que devem ser resolvidos pelo mesmo;
  - seus objetivos;
  - seus usuários diretos;
  - intenções destes usuários;
  - seus usuários indiretos;
  - intenções destes usuários;
  - outros sistemas que interagem com o sistema;
  - necessidade de notificação aos usuários quando alguns eventos ocorrem;
  - tarefas feitas pelos usuários de forma freqüente;
  - relacionamento entre o trabalho que um usuário precisa fazer e o sistema.

## 4 Ferramentas de Teste de Programas OO

Para a aplicação das técnicas e estratégias de teste apresentadas nas Seções 2 e 3, assim como para organização tanto de casos de teste quanto resultados, é necessária a utilização de ferramentas de apoio. Existem várias ferramentas de teste disponíveis no

mercado. Um ótimo exemplo de "lista de ferramentas" de teste comerciais pode ser encontrado no *site* de Brian Marick, denominado "Marick's Corner", cujo endereço é *http://www.rstcorp.com/marick/faqs/tools.htm*. Neste endereço, pode-se encontrar ferramentas de teste correspondentes às categorias apresentadas a seguir, de acordo com a(s) tarefa(s) de teste por elas apoiadas:

- elaboração de projeto de teste;
- seleção de casos de teste (ferramentas que ajudam na decisão de que testes devem ser executados);
- teste de aplicações com interface gráfica (GUI – *Graphical User Interface*);
- teste de regressão, através de funções de *capture/replay/verify* (ferramentas deste tipo capturam uma execução do sistema como referência, executam novamente o sistema, após modificações – *replay* - e comparam as duas execuções, detectando diferenças – *verify*);
- teste de sistemas cliente/servidor, incluindo testadores de carga;
- teste de desempenho;
- teste de carga máxima para sistemas (teste de estresse);
- gerenciadores de teste (atividades e dados);
- teste de aplicações não gráficas;
- automação de execução de conjunto volumoso de casos de testes;
- miscelânea de ferramentas, que ajudam a implementar testes. Por exemplo, ferramentas que geram automaticamente módulos *drivers* e *stubs* (teste de integração), geradores de assertivas, etc.;
- avaliação de testes;
- avaliação da qualidade de testes, tais como ferramentas de cobertura de teste;
- análise estática de código, tais como ferramentas com métricas de complexidade.

Através desta lista (e de outras fontes, tais como revistas especializadas, congressos, etc.), é possível escolher uma ferramenta de teste que seja ideal ao tipo de sistema sendo testado, bem como ao tipo de teste a ser realizado na instituição. A partir das fontes citadas, foram pesquisadas aproximadamente 40 ferramentas de teste. Deste estudo, foram selecionadas cinco ferramentas para a apresentação em detalhe. As cinco ferramentas são apresentadas nas Seções 4.1, 4.2 e 4.3, tendo sido escolhidas por estarem relacionadas aos principais tipos de teste que podem ser feitos com programas OO. Podem ser utilizadas tanto no teste estrutural como no funcional, considerando os escopos de teste de métodos, unidade, integração e de sistema.

As ferramentas são apresentadas de forma breve, a seguir:

- *AssertMate*: realiza o teste baseado em assertivas, inseridas no meio do código. Com as assertivas, é possível realizar tanto o teste funcional, estrutural, quanto o teste baseado em estados, como será apresentado a seguir. A ênfase da ferramenta é no teste de métodos e de classes.

- ⇨ *JavaStar*: realiza o teste de regressão, baseado na captura de eventos de interface, reexecução dos eventos, e comparação com execuções anteriores. A ferramenta enfatiza o teste de integração e de sistema.
- ⇨ *JavaSpec*: realiza o teste funcional de qualquer segmento de código submetido, seja ele um método, uma classe, ou um conjunto de classes. O teste é realizado utilizando-se combinações dos possíveis dados de entrada, e os resultados são comparados com a especificação fornecida à ferramenta.
- ⇨ *JavaScope*: realiza o teste de cobertura de código, enfatizando o teste de métodos.
- ⇨ *CodeWizard for Java*: realiza verificações no código que fornecem informações interessantes aos testadores, fazendo com que estes possam enfatizar a sua atenção em relação a algumas características específicas, dependendo do tipo de sistema em construção. As informações fornecidas pela ferramentas são indicativos de possíveis erros no código.

Além destas cinco ferramentas, serão apresentadas, na Seção 3.4, outras catorze ferramentas de teste para software OO. Apesar destas ferramentas serem apresentadas brevemente, são indicadas referências para os *sites* das mesmas, onde podem ser encontradas informações detalhadas sobre cada uma.

## 4.1 Ferramenta AssertMate

A ferramenta *AssertMate* foi desenvolvida pela *Reliable Software Technologies*, encontrando-se atualmente na versão 1.1. As informações sobre *AssertMate* aqui aqui apresentadas foram obtidas a partir do *site* da empresa, cujo endereço é *http://www.rstcorp.com*.

O objetivo da ferramenta é analisar assertivas especificadas por expressões em programas codificados na linguagem Java. As assertivas devem ser definidas na ferramenta como expressões booleanas que especificam o estado correto do programa. Em um programa correto, as assertivas devem ter sempre valor verdadeiro. Caso o valor resultante de sua avaliação seja falso, o programa apresenta um erro.

A ferramenta oferece suporte para quatro tipos de assertivas:

- ⇨ PRÉ-CONDIÇÕES: utilizadas para especificar restrições de dados que um método espera quando é chamado;
- ⇨ PÓS-CONDIÇÕES: utilizadas para especificar os dados de retorno de um método e das variáveis de classes, que devem estar em um estado correto assim que o método termina sua execução;
- ⇨ ASSERTIVAS DE DADOS: são utilizadas para especificar estados de dados em um ponto arbitrário do método. Elas verificam tanto variáveis que são locais aos métodos quanto atributos da classe.
- ⇨ INVARIANTES DE CLASSE: são utilizadas para especificar um estado consistente de dados para qualquer instância da classe, verificando se este estado permanece consistente durante toda a execução do programa.

Assertivas são colocadas no meio do código Java utilizando uma sintaxe similar à utilizada na inserção de comentários para o utilitário *JavaDoc*. O formato de uma assertiva é o seguinte:

@<assertion type> <boolean expression>[, <message>][;]

onde "<assertion_type>" é um dos quatro tipos de assertivas: "@assert" é uma assertiva de dados, "@pre" é uma pré-condição, "@post" é uma pós-condição e "@invariant" é uma invariante de classe. "<boolean expression>" é a assertiva que deve ser verificada. A mensagem é um campo opcional que pode ser utilizado para fornecimento de informações sobre o significado da assertiva. O símbolo ";" pode ser usado para separar assertivas de outros comentários colocados no mesmo bloco. A mensagem é apresentada ao usuário quando a assertiva retorna um valor falso.

Para compilar um programa com assertivas, deve-se preceder a linha normal de comando com a palavra "assert". O programa com assertivas é executado normalmente. Invariantes podem ser utilizadas em classes internas às outras (*inner classes*). Neste caso, como as classes internas podem acessar também variáveis privadas da classe externa, cuidados no processamento da classe interna devem ser considerados para que o estado da classe externa não seja corrompido. Para executar o programa sem a verificação de assertivas, basta recompilá-lo. Como estas estão especificadas dentro de blocos de comentários para o aplicativo "JavaDoc", a execução ocorre normalmente quando o programa não é compilado com a utilização da palavra "assert" antes da linha de comando de compilação.

As invariantes de classe verificam o estado da classe na entrada e na saída de todos os métodos nela definidos. Se uma invariante é violada na entrada do método, o estado da classe foi corrompido em um segmento de código externo à classe. Caso a violação ocorra na saída do método, isto significa que há um erro no método. A invariante pode ser inserida em qualquer lugar do código correspondente à definição da classe. Normalmente, uma invariante é inserida próximo à declaração dos atributos aos quais ela se relaciona. Um exemplo é apresentado a seguir:

```
class MyStack
{
  ...
  public static final int MAX_SIZE = 100;
  private Object[] m_stack;
  private int m_top;
  /** @invariant ((0 <= m_top) && (m_top < MAX_SIZE)),
    * "m_top nao eh maior ou igual a 0 e menor que MAX_SIZE"; */
  ...
}
```

As assertivas de dados podem ser inseridas em qualquer posição do código, podendo referenciar qualquer variável cujo escopo pertence ao segmento de código no qual foi inserida a assertiva. A seguir é apresentado um exemplo de sua aplicação:

```
public void foo()
{
  ...
  /** @assert (i>=0), "i nao pode ser negativo"; */
```

```
  ...
}
```

Uma pré-condição deve ser inserida antes do início do corpo do método, assim como a pós-condição. A pós-condição utiliza geralmente a meta-variável *$return* para referenciar o valor de retorno do método. O exemplo a seguir apresenta uma pré-condição que verifica se o parâmetro de entrada não é um número negativo, e a pós-condição verifica se a multiplicação do valor de retorno por ele mesmo resulta no valor de entrada (ou seja, sqrt(x*x) = x). O exemplo é apresentado a seguir:

```
/**
 * Este metodo calcula a raiz quadrada de um numero
 * @pre (x>=0),
 * "nao eh possivel calcular a raiz quadrada de um numero negativo";
 * @post ($return * $return == x), "raiz quadrada errada"; */

public double sqrt(double x)
{
  ...
}
```

*AssertMate* provê uma linguagem de assertivas, que estende Java. Esta linguagem possui comandos que especificam operadores lógicos, de conjunto, de manipulação de *arrays* e de especificação de limites de coleções, entre outros. A ferramenta permite que ações específicas sejam definidas quando uma assertiva é avaliada com o valor falso, através de métodos da classe *AssetReporter*. A seguir é apresentada uma lista destes comandos:

- *$and*: mesmo significado de "&&".
- *$exists*: operador matemático "existe", sendo verdadeiro se uma determinada expressão for verdadeira para pelo menos um elemento no conjunto.
- *$forall*: operador matemático "para todos", sendo verdadeiro se uma determinada expressão for verdadeira para cada elemento de um conjunto.
- *$iff*: significa "se e somente se", tendo a mesma semântica de "= =".
- *$implies*: operador lógico "implica". Retorna o valor verdadeiro se o lado esquerdo da expressão for falso, ou o lado direito for verdadeiro. A precedência de *$implies* é menor do que ==, mas maior do que *&&*.
- *$not*: mesmo significado de "!".
- *$or*: possui o mesmo significado de "||".
- *$return*: meta-variável que pode ser utilizada apenas em pós-condições. Refere-se à expressão que é utilizada no comando de retorno de um método.
- *$xor*: possui a mesma semântica de "^".

Sem dúvidas, a utilização de assertivas inseridas no código do programa e sua avaliação, a fim de verificar pontos onde os estados dos objetos podem estar inconsistentes é extremamente importante na realização do teste de software OO. Um

exemplo direto de sua utilização é a especificação de pré e pós-condições, no teste de estados, conforme apresentado na Seção 2 deste tutorial.

## 4.2 Ferramentas da SuntTest (SunTest Suite):

A *Sun Microsystems* possui uma divisão exclusivamente dedicada ao teste de programas Java: a *SunTest* [SUN98]. Esta divisão implementou e comercializa três ferramentas de teste, cada uma enfatizando características distintas e complementares de programas Java:

- JavaStar: teste de aplicações baseadas em GUI (*Graphical User Interface*);
- JavaSpec: teste baseado em especificações;
- JavaScope: teste baseado em critérios de cobertura.

Tais ferramentas formam a *SunTest Suite* (*http://www.sun.com/suntest*), que possui as seguintes características gerais:

- as ferramentas podem ser executadas em qualquer plataforma: "*write tests once, run tests anywhere*";
- baseadas em scripts em Java;
- scripts podem ser executados *stand-alone* ou integrados em ambientes de programação.

As três ferramentas que compõem o *SunTest Suite* são descritas a seguir.

### 4.2.1 JavaStar

A ferramenta *JavaStar* realiza o teste de aplicações baseadas em GUI (*Graphical User Interface*), permitindo gravar e automaticamente reexecutar as ações feitas pelo usuário. Em qualquer momento de reexecução, o estado dos objetos pode ser comparado ao estado esperado. Também é possível comparar estados da execução do momento com outras execuções, em outras plataformas. A ferramenta oferece maior poder de teste, maior flexibilidade, facilidade de uso, robusteza e maior longevidade para os scripts, já que estes são escritos em Java, não tendo uma linguagem específica para tal. Eventos são armazenados em um arquivo .JAVA, que JavaStar compila em um arquivo .CLASS, o qual pode ser executado posteriormente. As seguintes funções são utilizadas para comparação entre execuções:

- VERIFY: compara conteúdo, atributos e estados habilitados de cada componente GUI. Se a verificação falhar, informações sobre a mesma são armazenadas em um arquivo *log*, havendo a continuidade de execução do script;
- SYNCHRONIZE: a comparação ocorre, mas se houver alguma falha, o script termina, gerando uma exceção. Pode-se então detectar modificações de estado inapropriadas que ameacem a integridade do sistema.

Os processos de JavaStar executam no mesmo espaço de memória que os processos da aplicação sendo testada. Por isso JavaStar pode ter completo acesso aos atributos dos componentes da aplicação.

É possível modularizar os scripts, facilitando seu reuso, pois tamanho e complexidade podem ser reduzidos. Um exemplo de modularização seria um script desenvolvido exclusivamente para testar determinada janela (espécie de "coesão funcional" nos scripts). Através do comando COMPOSE TEST, scripts podem ser combinados. No modo COMPOSE VIEW, cada script é um nodo em uma árvore de teste que o usuário projeta. Tal árvore é apresentada graficamente. Relações de dependências entre nodos podem ser estabelecidas, baseadas em condições normais ou exceções, a fim de estabelecer o fluxo de teste baseado em resultados.

### 4.2.2 JavaSpec

A ferramenta *JavaSpec* realiza o teste funcional de unidade. *JavaSpec* testa a API de uma classe Java, sendo adequada para o desenvolvimento de testes que exercitem a funcionalidade do produto, e não da interface. Como esta é uma ferramenta de teste funcional, o desenvolvimento de casos de teste é feito a partir da especificação de projeto da classe.

Para a criação da especificação de teste para cada classe, devem ser especificados parâmetros de entrada, objetos, e assertivas sobre o estado destes objetos, retornando valores e condições de exceção. Pode-se também especificar código para ser executado antes e depois da chamada do método, sendo que este mesmo código inserido pode ser utilizado para a geração de casos de teste. *JavaSpec* interpreta a especificação completa para produzir e executar os testes, fornecer relatórios de resultados de testes e documentar a atividade a partir de relatórios relacionados diretamente à especificação.

*JavaSpec* define três entidades que compõem a estrutura de teste: caso de teste, teste e conjunto de teste. Cada conjunto de teste contém um ou mais testes, enquanto que cada teste contém um ou mais casos de teste. Um mesmo método, por exemplo, pode ser testado através de um conjunto de teste com dois testes: um com casos que representem condições normais, e outro que represente condições inválidas.

Os testes especificados da classe são armazenados em um arquivo denominado "Nome da classe".spi. Para executá-lo, o arquivo deve ser compilado em conjuntos de teste, através do *Test Suite Manager* da ferramenta. Os dados de teste são definidos através da criação de *data providers*. *Data provider* é um segmento de código que retorna dados de teste, que podem ser constantes, objetos ou dados de um banco de dados de teste, por exemplo. Para cada parâmetro de entrada de um método sob teste, pode-se criar vários *data providers* que retornam valores para teste. Após esta especificação, *JavaSpec* cria casos de teste através da combinação de todos os dados de teste. Além disso, vários conjuntos de assertivas podem ser especificados para cada caso de teste, os quais podem ser utilizados para a formação de combinações de teste também.

O processo de criação de especificação para o teste, utilizando *JavaSpec*, é composto pelas seguintes etapas:

1. Seleção da classe sob teste.
2. Seleção de métodos sob teste (MST).
   a. Definição dos dados de teste, incluindo qualquer objeto sob teste (OST) necessário.

b. Escrita do código que *JavaSpec* deve executar antes da execução do MST.

c. Escrita do código que *JavaSpec* deve executar após a execução do MST.

d. Declaração de assertivas para condições de exceção.

e. Declaração de assertivas para condições normais.

f. Definição de arquivos a serem importados.

3. Repetição do passo 2 para cada um dos outros MST's.

### 4.2.3 JavaScope

*JavaScope* é uma ferramenta composta por vários programas, que permite realizar a análise de cobertura do código, determinando a abrangência dos testes realizados. Como as outras ferramentas do *SunTest Suite*, *JavaScope* foi codificada em Java, o que permite que a ferramenta possa ser utilizada em qualquer plataforma.

Inicialmente, o código do sistema em teste deve ser instrumentado, ou seja, pontos de prova devem ser inseridos no código, a fim de que este gere um arquivo com informações sobre as execuções. O programa *jsintr* instrumenta o programa em teste, armazenando o código original no diretório *jsoriginal*. A fim de recuperar o código original, pode-se utilizar o programa *jsrestore*.

*JavaScope* verifica a cobertura de código, a qual pode ser avaliada em todo o sistema, e não apenas em unidades de compilação. Entretanto, os critérios de cobertura utilizados avaliam características de métodos, isoladamente, de forma similar ao teste estrutural procedimental. As informações de cobertura podem ser associadas a um *project*.

Como recurso de visualização *JavaScope* utiliza a coloração de código, indicando partes cobertas. *JavaScope* fornece uma API que possibilita a construção de browsers e geradores de relatórios personalizados.

Para o teste do código instrumentado, os seguintes critérios de cobertura são implementados pela ferramenta e podem ser selecionados pelo usuário:

- **Cobertura de métodos**: a ferramenta indica a porcentagem de métodos chamados pelo menos uma vez, o número de métodos cobertos e o número total de métodos do sistema em teste;
- **Cobertura de métodos construtores**;
- **Cobertura de blocos básicos**: similar ao critério *todos-nodos*, entretanto, por causa das exceções, um bloco básico nem sempre é executado integralmente;
- **Cobertura de arcos**;
- **Cobertura de comandos *switch***: todas as alternativas do comando devem ser executadas pelo menos uma vez;
- **Cobertura de *catch***: todos os blocos *catch* devem ser executados pelo menos uma vez;
- **Cobertura de condições**;
- **Cobertura relacional**: testa a comparação com todas as combinações de dados que

possam acusar o uso incorreto de um operador relacional.

- **Critérios de cobertura de gramática:**

  Cobertura de produção: requer que todas as produções da gramática devem ser cobertas pelo menos uma vez. Similar à cobertura de métodos.

  Cobertura de expansão: requer que todas as possíveis expansões da gramática sejam cobertas pelo menos uma vez.

## 4.3 CodeWizard for Java - Parasoft

A ferramenta *CodeWizard for Java*, desenvolvida pela empresa *Parasoft* (*http://www.parasoft.com*), analisa código em Java, com o objetivo de encontrar violações que contrariam princípios de boa programação Java, tais como: falta de inicialização de variáveis, colocação de rótulos "*case*" dentro de laços e encapsulamento de variáveis de instância herdadas. *CodeWizard for Java* envia suas mensagens à janela em que foi chamada, podendo também utilizar uma interface gráfica. A Tabela 2 apresenta uma lista de violações detectadas a partir da utilização da ferramenta *CodeWizard for Java*. São apresentados também os níveis de gravidade associados a cada tipo de violação, já que algumas representam falhas graves, enquanto que outras apenas fornecem informações sobre o sistema em teste.

| VIOLAÇÃO | GRAVIDADE |
|---|---|
| Variável não utilizada. | Possibilidade de violação. |
| Falta de inicialização de uma variável. | Possibilidade de violação. |
| Comando "case" mal formado em comando "switch". | Possibilidade de violação severa. |
| Encapsulamento de atributos herdados. | Possibilidade de violação. |
| Encapsulamento de métodos do tipo *static* herdados. | Possibilidade de violação severa. |
| Classes instanciadas não declaradas como sendo do tipo "final". | Informacional. |
| Atribuições dentro de uma condição do comando "if". | Violação. |
| Existência de atributos de pacotes e públicos. | Informacional. |
| Métodos públicos e de pacote não estão listados em conjunto, no início da declaração da classe. | Informacional. |
| Encapsulamento de atributos em métodos. | Possibilidade de violação. |
| Implementação de métodos declarados em interfaces. | Violação. |
| Utilização de "equals" no lugar de "==". | Possibilidade de violação. |
| Utilização de "Stringbuffer", no lugar de "String". | Informacional. |
| Utilização do rótulo "default:" para cada alternativa do comando "switch". | Possibilidade de violação. |

Tabela 2. Violações detectadas pela ferramenta *CodeWizard for Java* e gravidades correspondentes.

# 5 Organização da Atividade de Teste

A complexidade e o tamanho dos sistemas de software estão crescendo a cada dia. Estes fatores exigem que haja uma maior interação entre as pessoas envolvidas no

processo de desenvolvimento (e, conseqüentemente, no teste). Com este cenário, é necessário que as tarefas que compõem a atividade de teste sejam distribuídas aos componentes de equipe, devendo haver procedimentos específicos para a organização e distribuição destas tarefas.

Algumas das características desejadas em um testador são: conhecimento do domínio da aplicação, do ambiente/contexto aonde o sistema será utilizado e das pessoas que utilizarão o sistema; criatividade e organização na geração, submissão e avaliação de casos de teste; organização no seguimento de procedimentos no teste, de forma criteriosa, levando a uma padronização e sistematização do processo de teste.

Havendo a viabilidade de realização do teste em equipe, é bem mais provável que grande parte destas características sejam encontradas em várias pessoas que cooperem para alcançar um mesmo objetivo (encontrar erros em um programa), do que em uma única pessoa.

A Seção 5.1 apresenta sugestões de "papéis" e responsabilidades associadas, que podem ser adotados em uma empresa desenvolvedora de software. Informações sobre modelos de empresas podem ser encontrados em [HER99b]. Na Seção 5.2, é apresentado um modelo de teste de software OO. Já a Seção 5.3 apresenta as estratégias mais comuns de teste de software OO, encontradas na literatura.

## 5.1 Definição de "Papéis" e Responsabilidades Associadas

Para a organização das atividades do teste em equipe, sugere-se que os "papéis" apresentados na Tabela 3, acompanhados por responsabilidades correspondentes, sejam implantados na empresa.

| PAPEL | RESPONSABILIDADES |
|---|---|
| Analista de Sistemas | Elaborar a especificação do sistema a ser desenvolvido.<br>Distribuir tarefas de desenvolvimento do sistema.<br>Elaborar casos de uso para teste.<br>Indicar modificações do sistema, posteriores à liberação. |
| Programador | Implementar o sistema, de acordo com especificações do analista.<br>Implementar modificações do sistema, posteriores à sua liberação.<br>Corrigir erros detectados pelos testadores.<br>Realizar o teste de unidade do código por ele desenvolvido.<br>Realizar o teste de integração do código desenvolvido por outros colegas programadores. |

| | |
|---|---|
| Gerente de Teste | Elaborar o plano de teste e distribuir tarefas aos testadores.<br>Organizar documentação proveniente do teste.<br>Incentivar a "alimentação" e a utilização do repositório de erros.<br>Controlar o registro e as atualizações do repositório de erros.<br>Controlar a realização dos testes, verificando se existem testadores sobrecarregados, ou com atividades pendentes.<br>Homologar o término dos testes, baseado em informações fornecidas pelos testadores sobre a realização do teste. |
| Testador | Testar o programa, realizando especificamente a tarefa a ele designada pelo gerente de teste.<br>Utilizar os casos de uso especificados pelo analista no teste.<br>Registrar erros e comportamentos anômalos do programa no repositório de erros. |
| Usuário/ Cliente | Especificar a funcionalidade desejada ao analista na especificação.<br>Colaborar, no teste *alfa* e *beta*, para a completa validação do sistema de software desenvolvido. |

Tabela 3. Papéis e responsabilidades no teste cooperativo.

## 5.2 Modelo de Equipe de Teste

Inicialmente, os programadores realizam o teste de unidade de seus próprios códigos. Devem ser selecionados tanto casos de teste funcionais quanto estruturais (Capítulo 2), e o teste deve ser realizado em um ambiente controlado, onde o módulo em teste esteja isolado dos demais que integram o ambiente. No paradigma OO, o programador deve realizar tanto o teste dos métodos quanto da classe por ele desenvolvidos.

Posteriormente, um programador designado realiza o teste de integração de cada módulo com os outros módulos do sistema. O teste de integração deve ser feito por um programador, porque é necessário conhecimento do código e da metodologia de desenvolvimento do sistema. Deve ser feito por um programador que não tenha participado da implementação do sistema até então, para evitar que vícios de raciocínio encubram erros no programa. Para a integração, o programador deve utilizar uma estratégia de integração.

Após realizados os testes de unidade e de integração, deve ser realizado o teste de sistema. Este teste de sistema pode ser dividido em duas etapas. Na primeira, deve-se envolver o gerente de teste e/ou o analista de sistemas, a fim de que estes façam a validação inicial do sistema. Na segunda etapa, tem-se o teste feito por uma equipe de homologação. Este tipo de teste deve ser feito, preferencialmente, por alguém que não se envolveu no processo de desenvolvimento do sistema diretamente, e que conheça o domínio do negócio a que se refere o produto. Para tanto, são sugeridas as seguintes categorias de profissionais para a composição da "equipe de homologação":

- analistas de negócios, que conheçam profundamente o negócio, e portanto têm

condições de verificar se o sistema está conforme com as especificações do produto;

- no caso de desenvolvimento de software para uso interno, profissionais "antigos" na empresa, que conheçam também bastante bem o negócio, devido à convivência com a empresa;
- profissionais que atuam no serviço de *help-desk* da empresa. Estas pessoas recebem vários telefonemas por dia de usuários que encontraram algum problema no uso do produto de software. Portanto, são pessoas que realmente conhecem o tipo de comportamento dos usuários, já que têm contato com eles periodicamente;
- profissionais que instalam sistemas de software nas empresas clientes, que, como os que atuam no serviço de *help-desk*, convivem com usuários e conhecem, portanto, o comportamento dos mesmos e sua percepção em relação ao sistema.

O teste de sistema é tipicamente funcional, e deve ser realizado utilizando-se como guia os casos de uso especificados. Neste momento do teste devem também ser registrados os erros em um repositório. Erros encontrados tanto nas fases de teste de unidade, de integração quanto de sistema podem vir a fazer parte do repositório de erros. Quando algum erro é julgado "interessante" sob o ponto de vista de complexidade e possibilidade de generalização, deve ser comunicado ao gerente de teste, que autoriza ou não a inclusão do mesmo no repositório. Este repositório não pode ser muito grande, a ponto de dificultar o acesso e pesquisa aos erros mais "interessantes", como não pode ser muito pequeno, incluindo apenas uma pequena quantidade de erros, sob pena de desperdício de situações que poderiam servir como experiências valiosas. Um exemplo de erro "interessante" é uma situação na qual há problemas de índices de um banco de dados, cuja solução é simples (reindexar o banco), entretanto os seus "sintomas" podem ser variados e muitas vezes confusos.

A Figura 7 apresenta uma visão esquemática do processo de teste no modelo de empresa apresentado nesta seção.

Após ser realizado o teste de sistema, usuários típicos devem ser convidados para a realização dos testes *alfa* e *beta*. Devem ser selecionados usuários e/ou empresas que apresentem boa capacidade crítica, certa cumplicidade com a empresa desenvolvedora e que, de preferência, já sejam usuários de outros sistemas desenvolvidos pela mesma empresa. Este seria um "favor" prestado pelo usuário, que teria como benefício indireto o aperfeiçoamento do seu sistema, tanto por correções de erros eventualmente existentes, quanto por aperfeiçoamentos de funcionalidades já implementadas.

Uma alternativa que possibilita um maior comprometimento dos usuários, permitindo com que a empresa possa cobrá-lo sobre os resultados dos testes, é a troca mútua de benefícios. Por exemplo, a empresa desenvolvedora fornece licenças de uso do produto de software gratuitamente ao usuário, e este, em contrapartida, fornece relatórios de uso e de erros à empresa desenvolvedora, em períodos pré-determinados em um contrato estabelecido entre ambas as partes.

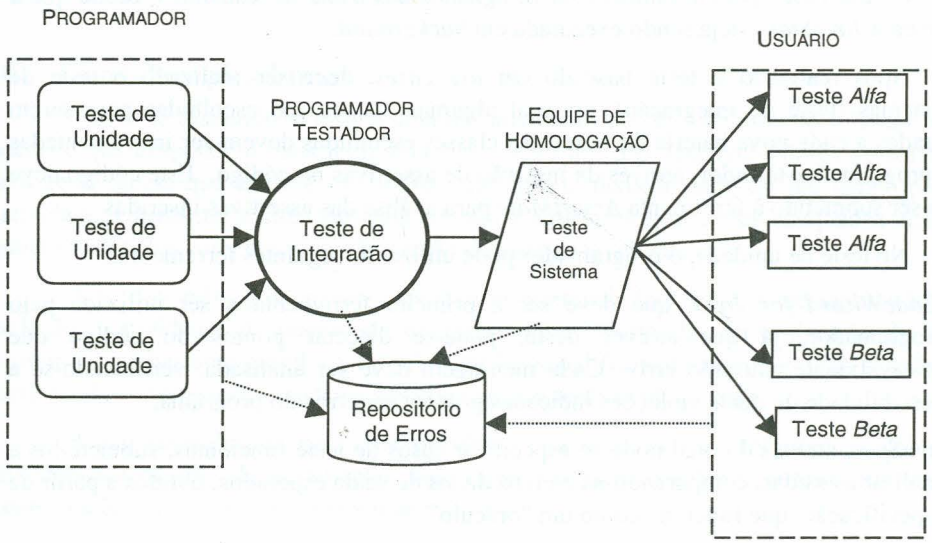


Figura 7. Representação esquemática do primeiro modelo de teste.

## 5.3 Estratégias de Teste de Software OO

O teste de sistemas de software OO pode ser realizado de duas maneiras: *top-down* (teste de sistema ao teste de unidade), ou *bottom-up* (teste de unidade ao teste de sistema).

A estratégia de teste *top-down* tem como principal benefício a reutilização de casos de teste a partir da execução dos *use cases*. Como a execução de alto nível é monitorada por ferramentas, tem-se registrados todos os caminhos executados por cada *use case*, considerando tanto métodos quanto classes. Caso ocorra um erro, este caminho pode ser revisado, tendo-se desta forma o escopo de código reduzido para a análise.

Utilizando-se a estratégia *bottom-up*, é possível fazer o teste durante o processo de desenvolvimento do sistema, não havendo a necessidade, como acontece na estratégia *top-down*, de ter-se o sistema completamente desenvolvido para testá-lo.

### 5.3.1 Estratégia de Teste Top-down

No caso do teste *top-down*, a abordagem de tese poderia ser direcionada pela especificação de *use cases*. Cada conjunto de *use cases* relacionados, representados pelo preenchimento de um *template*, pode ser designado a uma equipe de homologação, para o teste de sistema. Como o *use case* representa uma função de alto nível do sistema em teste, a sua execução pode ser gravada utilizando-se a ferramenta *JavaStar*, da *SunTest Suite*. Com a execução gravada como uma seqüência de eventos, é possível reexecutá-la quantas vezes for necessário, analisando-se diferentes aspectos em cada reexecução.

Os *use cases* podem também ser designados aos usuários testadores, desde que a ferramenta *JavaStar* esteja sendo executada em *background.*

Após realizado o teste baseado em *use cases*, deve ser realizado o teste de subsistemas (teste de integração), no qual algumas classes são escolhidas para serem integradas a cada nova bateria de testes. As classes escolhidas devem ser instrumentadas pelo programador-testador, através da inserção de assertivas no código. Este código deve então ser submetido à ferramenta *AssertMate* para análise das assertivas inseridas.

No teste de unidade, o programador pode utilizar as seguintes ferramentas:

- *CodeWizard for Java*, que deve ser a primeira ferramenta a ser utilizada pelo programador, já que através desta, pode-se detectar pontos do código que possivelmente causarão erros. Cada mensagem deve ser analisada, verificando-se a possibilidade de quais violações indicadas possam ser erros do programa.
- *JavaSpec*, através da qual pode-se especificar casos de teste funcionais, submetê-los e analisar as saídas, comparando-as com os dados de saída esperados, obtidos a partir da especificação, que funciona como um "oráculo".
- *JavaScope*, através da qual pode ser realizado o teste de cobertura (teste estrutural) dos métodos. O programador deve selecionar um critério de cobertura. Quanto maior for a necessidade de garantia de confiabilidade da classe, mais forte deve ser o critério de cobertura escolhido. Devem então ser estabelecidas porcentagens de cobertura para satisfação do critério, já que, devido à existência de caminhos não executáveis no programa, há a possibilidade de inviabilidade de satisfação total de um determinado critério.

### 5.3.2 Estratégia de Teste Bottom-up

Na estratégia de teste *bottom-up*, tem-se as mesmas etapas do teste *top-down*, com a diferença na ordem em que estas ocorrem. Por isso, em cada etapa, as mesmas ferramentas citadas na Seção 5.2.1 podem ser utilizadas.

Nesta estratégia, a atividade de teste inicia com o teste de unidade, feito pelo programador. O programador deve testar cada método, para então testar as interações entre os métodos de uma determinada classe. Este tipo de teste é feito utilizando-se, principalmente, técnicas de teste baseado em estados, auxiliadas pela ferramenta *AssertMate.*

Após o teste de interações de métodos em uma mesma classe, deve ser realizado o teste de integração de métodos de classes distintas, o que deve ser feito pelo programador testador, que assume o papel de testador ativo. O programador da classe assume o papel de testador passivo, trabalhando como um profissional de suporte ao testador ativo.

Feito o teste de integração, devem ser realizados os testes de sistema e de usuários,

conforme apresentado na Seção 5.2.2.

## 6 Considerações Finais

Como já discutido, o teste é uma das atividades do ciclo de vida do software mais caras e que despendem mais tempo na sua realização. Os desenvolvedores de software em geral sabem que testar é importante, que devem dedicar parte do seu tempo para a atividade, entretanto, a situação mais comum é aquela na qual testes adequados acabam sendo realizados somente após a liberação do sistema ao cliente.

Um dos principais motivos para este eventual descaso é a desmotivação em relação à atividade. A quantidade de informações envolvidas na realização do teste sistemático é bastante grande, estas informações são difíceis de serem obtidas, e acaba-se arriscando e entregando o sistema aos clientes sem testá-lo idealmente.

Uma das soluções para este problema é a utilização de ferramentas de teste. Existem vários tipos de ferramentas, para diferentes escopos e tipos de teste. Entretanto, suas funções principais são as seguintes:

- gerar casos de teste, a partir de uma especificação, por exemplo;
- organizar casos de teste;
- gravar e reexecutar casos de teste;
- realizar a análise dinâmica do código.

Ou seja, as ferramentas em geral enfatizam aspectos relacionados às técnicas de teste de software. Todas estas técnicas de teste devem ser consideradas de forma conjunta e complementar. Somente desta forma tem-se condições de realizar uma atividade de teste eficaz, completa e produtiva.

A ampla difusão do paradigma OO para a programação de sistemas deveria vir acompanhada da aplicação de técnicas de programação e teste, a fim de não diminuir ou quase anular todas as vantagens deste paradigma. Com a organização do processo de desenvolvimento de sistemas, tem-se, mais provavelmente, a obtenção de produtos mais confiáveis, manuteníveis e testáveis, com sua funcionalidade garantida de forma segura.

## Referências


[BIN95] BINDER, R. V. State-Based Testing. Object, vol. 5(6). July-August 1995.

[BOE78] BOEHM, B. et al. Characteristics of Software Quality. New York:Elsevier North Holland, 1978.

[CHA99] CHAN, M. C. et al. Java: 1001 Dicas de Programação. Editora Makron Books. 1999. 714 p.

[CHO78] CHOW, T. S. Testing Software Design Modeled by Finite-State Machines. IEEE Transactions on Software Engineering, vol. 4(3). 1978. Pp. 178-187.

[FUJ91] FUJIWARA, S. et al. Test Selection Based on Finite State Models. IEEE Transactions on Software Engineering, vol. 17(6). June 1991. Pp. 591-603.

[GOE70] GÖENÇ, G. A Method for Design of Fault-Detection Experiments. IEEE Transactions on Computers, vol. C-19(6). Jun. 1970. Pp. 551-558.

[HAR92] HARROLD, M. J. et al. Incremental Testing of Object-Oriented Class Structure. In: 14th International Conference on Software Engineering, ACM. 1992.

[HER95a] HERBERT, J. S.; PRICE, A. M. de A. Geração Automática de Cenários de Teste a partir de Gramática Livre do Contexto. Seminário Integrado de Software e Hardware. XV Congresso da SBC. Pp. 929-940. Canela, RS. Ago. 1995.

[HER95b] HERBERT, J. S.; PRICE, A. M. de A. Métodos para a Avaliação de Qualidade de Software. XIV Jornadas de Atualização em Informática. XV Congresso da SBC. Canela, RS. Ago. 1995.

[HER99a] HERBERT, J. S.; PRICE, A. M. de A. Técnicas e Ferramentas de Teste para a Linguagem Java. Tutorial. IV Simpósio Brasileiro de Linguagens de Programação. Porto Alegre, RS. Maio, 1999.

[HER99b] HERBERT, J. S.; PRICE, A. M. de A. Aprimorando a Qualidade de Software através do Teste Cooperativo. Mini-curso. X Conferência Internacional de Tecnologia de Software. Curitiba, PR. Maio, 1999.

[HET87] HETZEL, W. Guia Completo ao Teste de Software. Editora Campus. 1987. 206 p.

[HUM88] HUMPHREY, W. S. Characterizing the Software Process: A Maturity Framework. IEEE Software. Mar. 1988.

[JAC92] JACOBSON, I. et al. Object-Oriented Software Engineering – A Use Case Driven Approach. Addison-Wesley. 1992. 528 p.

[MAL91] MALDONADO, J. C. Critérios Potenciais Usos: Uma Contribuição ao Teste Estrutural de Software. Tese de Doutorado. DCA/FEE/UNICAMP. Campinas, SP. Julho, 1991.

[MAT98] MATTINGLY, L.; RAO, H. Writing Effective Use Cases and Introducing Collaboration Cases. Journal of Object-Oriented Programming. October, 1998. Pp. 77-84.

[McC99] McCABE & ASSOCIATES. McCabe Visual Testing ToolSet. Available at *http://www.mccabe.com.*

[McG96] McGREGOR, J. Testing Object-Oriented Components. In: 10th European Conference on Object-Oriented Programming. Tutorial Notes. July 1996.

[MOS93] MOSLEY, D. The Handbook of MIS Application Software Testing. Yourdon Press - Prentice-Hall. 1993.

[MYE79] MYERS, G. The Art of Software Testing. . New York: John Willey & Sons, 1979. 170 p.

[NAK95] NAKAZATO, K. K. Módulo de Geração de Seqüências de Teste baseada em Máquinas de Estado Finito. São Carlos, ICMSC/USP, 1995. Dissertação de Mestrado. 126 p.

[PRE95] PRESSMAN, R. S. Engenharia de Software. São Paulo: Makron Books. 1995. 1056 p.

[RAP85] RAPPS, S.; WEYUKER, E. J. Selecting Software Test Data Using Data Flow Information. IEEE Transactions on Software Engineering, vol. 11(4). Abril, 1985.

[RAT97a] RATIONAL Software Corporation. Object Constraint Language Specification – version 1.1. 1997. Available at http://www.rational.com/uml.

[RAT97b] RATIONAL Software Corporation. UML Notation Guide – version 1.1. 1997. Available at http://www.rational.com/uml.

[SAB88] SABNANI, K. K.; DAHBURA, A. A Protocol Test Generation Procedure. Computer Networks and ISDN Systems, vol. 15(4). April 1988. Pp. 285-297.

[SUN98] SUN MICROSYSTEMS. SunTest Suite Java Testing Tools. Available at: http://www.sun.com/suntest/JavaCC/javaccdownload.html. 1998.

[TUR93] TURNER, C. D.; ROBSON, D. J. The Testing of Object-Oriented Programs. Technical Report:TR-13/92. Feb. 1993.

[VER94] VERGÍLIO, S. R. et al. Caminhos Não-executáveis no Teste de Integração: Caracterização, Previsão e Determinação. In: SBES, 8., 1994. Curitiba. Anais... Curitiba: [s.n.], 1994.

[VER97] VERGÍLIO, S.R et al. Aumentando a Eficácia dos Critérios Estruturais Através da Utilização de Critérios Restritos. In: WORKSHOP DO PROJETO VALIDAÇÃO E TESTE DE SISTEMAS DE OPERAÇÃO. Águas de Lindóia, SP. Anais... Jan. 1997.