PUC

ISSN 0103-9741

Monografias em Ciência da Computação
n° preliminar xx/06

AutoTest - Arcabouço para a Automação dos Testes de Módulos Redigidos em C
Versão 2

Arndt von Staa

Departamento de Informática

PONTIFÍCIA UNIVERSIDADE CATÓLICA DO RIO DE JANEIRO
RUA MARQUÊS DE SÃO VICENTE, 225 - CEP 22453-900
RIO DE JANEIRO - BRASIL

PUC RIO - DEPARTAMENTO DE INFORMÁTICA ISSN 0103-9741

Monografias em Ciência da Computação, Nº preliminar xx/06
Editor: Carlos J. P. Lucena Março, 2006
revisão: fevereiro, 2006

AutoTest - Arcabouço para a Automação dos Testes de Módulos Redigidos em C Versão 2

Arndt von Staa

Responsável por publicações:

Rosane Teles Lins Castilho
Assessoria de Biblioteca, Documentação e Informação
PUC-Rio Departamento de Informática
Rua Marquês de São Vicente, 225 - Gávea
22453-900 Rio de Janeiro RJ Brasil
Tel. +55 21 3114-1516 Fax: +55 21 3114-1530
E-mail: bib-di@inf.puc-rio.br

Sumário

AutoTest - Arcabouço para a Automação dos Testes de Módulos Redigidos em C Versão 2

Arndt von Staa
arndt@inf.puc-rio.br

Abstract. A test automation framework is presented. This framework is specifically geared towards modules written in C. Initially an abridged description of the key testing concepts is presented. Afterwards two test automation approaches are described. The first one uses a specific test module containing functions that exercise the module being tested. In this approach, the test suite corresponds to the source code of the test control module. The second approach is based on a framework that must be instantiated to test a specific module. The instantiated test control module implements a test script interpreter. In this approach one or more tests scripts specifically geared to sufficiently test the given module defines the test suite. Afterwards techniques to control memory leakage control and test coverage are introduced. The article emphasizes incremental development of programs as well as of modules, combining this with a test driven development approach..

Keywords: C program testing framework; Incremental development; Memory leakage, Module testing; Software Engineering; Test automation; Test coverage; Test driven development.

Resumo. É apresentado um arcabouço (*framework*) para a automação de testes de módulos redigidos na linguagem C. O artigo apresenta, resumidamente, os conceitos básicos de teste de módulos necessários para a sua compreensão. O artigo descreve duas modalidades de automação de testes. A primeira é baseada em um módulo utilizando funções de teste especificamente implementadas para testar o módulo sob teste. Nesta abordagem a massa de teste é formada pelo código fonte do módulo de controle do teste. A segunda abordagem utiliza um interpretador de diretivas de teste focadas no módulo sob teste. Nesta segunda modalidade, a massa de teste é formada por um ou mais arquivos de diretivas contendo diversos casos de teste e a respectiva documentação. São apresentadas também técnicas avançadas para o controle de vazamento de memória e de cobertura dos testes. O artigo enfatiza o desenvolvimento incremental tanto de programas como de módulos, combinando isto com abordagens de desenvolvimento dirigido pelos testes.

Palavras-chave: Arcabouço de testes visando C; Automação do teste; Cobertura de testes; Desenvolvimento incremental; Desenvolvimento dirigido por testes; Engenharia de software; Teste de módulos; Vazamento de memória.

1 Introdução

Este documento tem por objetivo discutir e ilustrar técnicas de automatização de testes de módulos. Este documento complementa os capítulos de *Instrumentação, Teste de Módulos* e *Integração* contidos no livro *Programação Modular* [Staa 2000].

O presente documento está vinculado ao arcabouço `AutoTest` de apoio ao teste automatizado de programas redigidos em C contido em `ARCABOUCOTESTE-2-00.ZIP`. O presente texto apresenta os aspectos conceituais do teste automatizado, ilustrando-os com pequenos exemplos. Para não ser inutilmente extenso e repetitivo, os detalhes do uso do arcabouço encontram-se nos comentários dos módulos de definição e nos *scripts* de teste distribuídos. O uso dos instrumentos é ilustrado em um exemplo que também se encontra no arquivo de distribuição. Recomenda-se fortemente a leitura do documento `Arcabouco-Teste-2006-2-00-LeiaMe.pdf`.

Testar programas é uma das várias técnicas de *controle da qualidade de programas*. Ao testar um artefato[1] formado por código, este é submetido a várias massas de teste[2] compostos por vários casos de teste. Para cada um dos casos de teste executados, os resultados obtidos são comparados com os resultados esperados. Caso a comparação identifique uma discrepância, terá sido identificada uma falha[3]. Uma discrepância pode ser identificada de várias formas, entre elas:

- uma comparação resultando em diferenças entre o esperado e o obtido. Ocorre usualmente em processamento de inteiros ou de símbolos;
- uma comparação em que o valor obtido não está dentro dos limites de tolerância aceitáveis. Ocorre usualmente em processamento matemático envolvendo vírgula flutuante;
- uma estrutura de dados que não satisfaz as suas assertivas estruturais ou regras de negócio;
- uma interface com o usuário que confunde ou dificulta o trabalho deste;
- uma imagem com imperfeições de *renderização*;
- um arquivo, ou base de dados, cujo conteúdo e estrutura não corresponde ao esperado.

Testes podem ser realizados com diferentes objetivos em mente. Por exemplo, o ***teste de corretude***[4] procura encontrar diferenças entre o especificado e o implementado. O artefato

---

[1] ***Artefato*** é algum resultado tangível do desenvolvimento de um programa. São exemplos: documentos ou arquivos contendo especificações, projetos ou planos; arquivos contendo código fonte ou objeto; módulos; componentes; arquivos contendo dados para teste; *logs* que registram a execução dos testes; manuais (papel) para o usuário; arquivos contendo texto de auxílio (*help*); etc.

[2] ***Massa de teste*** é um conjunto de dados e comandos correspondentes a um ou mais ***casos de teste*** e que será utilizada em uma ***sessão de teste***. Caso o teste seja automatizado, a sessão de teste passa a ser uma ***execução de teste***.

[3] ***Falhas*** são comportamentos observados do programa e que são diferentes do esperado, comprometendo a confiabilidade do resultado. ***Defeitos*** são comportamentos do programa diferentes do esperado, mas que não comprometem a confiabilidade do resultado, no entanto afetam a percepção da sua qualidade.

[4] ***Corretude*** (Correto + -tude): propriedade de estar correto. Preferimos o neologismo *corretude* à palavra "correção" devido à ambigüidade inerente a esta última, já que pode significar i) o fato de estar correto, ii) o processo de tornar correto, ou iii) o processo de verificar se está correto.

estará correto caso esteja em conformidade exata[5] com sua especificação. Uma conseqüência desta definição é um programa poder ser dito "estar correto" mesmo que implemente uma especificação errada e, portanto, estar errado do ponto de vista do usuário. Já um programa que satisfaz o esperado pelo usuário, mas que não esteja em conformidade com a especificação será um programa incorreto.

No ***teste da interface*** a preocupação é identificar se a interface do componente[6] ou módulo[7] permite a construção de programas que utilizem estes artefatos, sem requerer quaisquer alterações, adaptações ou interfaces de conversão (*wrappers*). Já no ***teste de adequação*** procura-se determinar se o construto[8] resolve os problemas que o usuário[9] espera ver resolvidos. No ***teste de utilizabilidade*** procura-se verificar se o construto é fácil de usar (interface humana adequada às pessoas que irão utilizar o programa) e de aprender a utilizar. No ***teste de segurança*** procura-se encontrar brechas de segurança que permitam pessoas azaradas ou mal intencionadas a causar danos. No ***teste de implantação*** procura-se verificar se o construto pode ser colocado em correto funcionamento nas plataformas do usuário. É verificado se o instalador[10] correta e complemente instala o construto sem deturpar o funcionamento de outros programas já instalados na plataforma. É verificado, ainda, se as bases de dados e outros arquivos do usuário requeridos estão adequadamente povoados e em conformidade com as necessidades do novo construto. Finalmente, no ***teste de capacidade*** procura-se avaliar se o construto é capaz de atender à demanda esperada, bem como os limites de capacidade a partir dos quais o programa entra em colapso por excesso de demanda.

Conforme será visto mais adiante, uma forma de automatizar testes é criar um módulo, escrito na linguagem do módulo a testar [JUnit, CPPUnit]. Em outra forma a massa de teste é redigida em uma linguagem de diretivas (*script*) [FG 1999]. A massa de teste conterá agora diretivas para exercitar o módulo a testar e para comparar os resultados desses exercícios. A massa de teste pode conter também diretivas que monitorem o próprio processo de teste. Essas diretivas especiais são processadas pela instrumentação contida no arcabouço (*framework*) de teste.

De maneira geral programas, exceto os mais simples, são compostos por diversos módulos e/ou componentes. O desenvolvimento de um programa modular é usualmente realizado de forma incremental, evoluindo de construto a construto. Em cada incremento adiciona-se

---

5 Conformidade ***exata*** ocorre quando o módulo implementa tudo o que está especificado e nada além disso.

6 ***Componente*** é um conjunto de módulos implementando uma interface bem definida e incorporado ao programa sem requerer qualquer modificação ou ajuste de interface (*as is*, ou *verbatim*). Exemplos são arquivos `.jar`, `.dll`, `.so`, ou `.lib`.

7 ***Módulo*** é uma unidade de compilação. Exemplos: um arquivo de código fonte C, C++.

8 ***Construto*** é um programa operacional, possivelmente incompleto, que implementa uma parte da funcionalidade que o programa final almejado deverá vir a implementar. No ***desenvolvimento incremental***, cada construto incremental implementará mais funções do programa final do que o construto antecessor. A sucessão de incrementos culmina com o construto final correspondendo ao programa almejado plenamente desenvolvido.

9 ***Usuário*** de um módulo é tomado no sentido lato. São usuários: o ***módulo cliente*** que utiliza (acessa) um outro módulo (***módulo servidor***); o ***programador cliente*** que reutiliza o módulo em algum programa; o ***programador desenvolvedor*** que desenvolve ou mantém o módulo; a pessoa, ***usuário*** no *sentido estrito*, que utiliza o programa contendo o módulo.

10 ***Instalador*** é uma ferramenta que decodifica, expande e transcreve os arquivos fornecidos no pacote de distribuição, criando os programas executáveis e demais arquivos por eles requeridos, bem como ajusta os parâmetros do sistema operacional de modo que os programas possam ser adequadamente utilizados.

ao conjunto de elementos que forma o incremento antecessor poucos, idealmente um, módulos ou componentes devidamente aprovados. A cada incremento são testadas as novas propriedades adicionadas. Uma vez o módulo tendo sido aprovado, este é incorporado ao conjunto de módulos aceitos.

Neste artigo vamos nos restringir ao teste da corretude. Um dos problemas cruciais com corretude é a continua evolução das especificações. Especificações evoluem no tempo em virtude do aprendizado relativo ao problema a resolver ou à tecnologia empregada [TH 2004]. De uma certa forma é até lícito assumir que programas *convergem* para a forma correta e adequada ao invés de serem *construídos* para tal. O aprendizado ocorre durante o desenvolvimento, durante os testes e durante as avaliações realizadas após o desenvolvimento. Especificações também evoluem em virtude da observação de novas necessidades que vão surgindo em conseqüência do próprio desenvolvimento ou uso do artefato [Lehman 1998]. Propriedades inicialmente não observadas, ou não previstas, de repente tornam-se importantes. Isto é particularmente o caso quando se adota uma postura de desenvolvimento sem generalização precoce, na qual se procura evitar despesas inúteis com características que possivelmente jamais serão utilizadas [Beck 2000, HT 2001, Cockburn 2002].

Durante o desenvolvimento podem ser cometidos ***Erros de desenvolvimento***. Estes são cometidos por desenvolvedores, por ferramentas, e/ou instrumentos de apoio ao desenvolvimento. Erros de desenvolvimento introduzem ***faltas*** ou ***deficiências*** no artefato. Uma ***falta*** é uma inadequação latente que, se *exercitada de uma determinada forma*, conduz a um ***erro de processamento***. Erros de processamento podem ou não ser observados de alguma forma. Quando observados passam a ser ***falhas*** [ALRL 2004, Romanowsky 2005]. De forma similar, ***deficiências*** observadas passam a ser ***defeitos***. O grande problema reside no fato que muitos erros de processamento permanecem não observados por longos períodos, levando a um potencialmente enorme acúmulo de danos. Um dos principais objetivos dos testes é identificar o maior número possível de faltas e deficiências existentes em um construto. Uma vez identificadas elas poderão ser eliminadas através da depuração.

Uma vez encontrada uma falha deve-se diagnosticá-la, localizando *todos* os pontos no código (faltas) que contribuam para ela, somente depois disso o código deverá ser corrigido. A diagnose[11] dos problemas encontrados é freqüentemente um trabalho demorado, incompleto e incorreto. Quando incompleto, uma parte do problema terá sido resolvida, possibilitando a sua manifestação em condições diferentes das exercitadas nos testes. Quando incorreto, a própria remoção do problema introduz novos antes inexistentes. Para agilizar a diagnose e reduzir a freqüência dos erros cometidos, pode-se utilizar instrumentação incorporada ao código do módulo [Staa 2000]. Diversos desses instrumentos podem ser encapsulados em módulos próprios a serem incorporados ao arcabouço de apoio ao teste automatizado.

A remoção das faltas contidas em um artefato bem como a evolução deste, conduzem a repetidas alterações no artefato. Muitas destas alterações são estruturais e não somente o acerto de algumas linhas de código (*refactoring*) [Beck 2000, Fowler 2000]. Após cada alteração torna-se necessário testar de novo o artefato com o intuito de verificar se os problemas[12] foram resolvidos e nenhum novo foi introduzido. Chama-se este tipo de teste de ***teste de regressão***. Este procura verificar se as alterações realizadas em um módulo não

---

[11] ***Diagnose*** é o processo de localizar todas as faltas causadoras dos problemas observados durante o teste ou uso de um artefato. ***Depuração*** (*debugging*) é o processo de *remoção integral* destas faltas, idealmente sem acrescentar novas.

[12] Entendemos por ***problema*** coletivamente os relatos de falhas ou defeitos, bem como as solicitações de melhorias ou de adaptação, e, ainda, as solicitações de desenvolvimento de novos módulos, componentes ou mesmo programas.

afetaram mais do que o estritamente esperado em virtude dessa alteração, levando em conta todos os construtos que utilizem o módulo alterado.

O mesmo problema ocorre quando se desenvolve um programa, ou mesmo um módulo, de forma incremental [Beck 2000, Cockburn 2002]. Neste caso, a cada vez que se incrementa um módulo ou componente que já havia sido aprovado, torna-se necessário refazer todos os testes de todos os construtos que contenham este módulo ou componente. Conclui-se disto que os diversos testes são realizados repetidas vezes.

Reexecutar de forma *manual* todos os casos de testes, que podem ser vários milhares, é uma tarefa tediosa, cara e, por isso, freqüentemente não realizada. A conseqüência é a criação de programas não confiáveis. Torna-se desejável, então, automatizar os testes de modo que possam ser reexecutados várias vezes e isso a um custo baixo, assegurando a completeza necessária para que se possa considerar fidedigno[13] o construto. Evidentemente, a automação de testes adiciona custos [FG 1999] e não se aplica a todas as situações encontradas ao desenvolver software.

Os custos adicionais decorrem do desenvolvimento da instrumentação que realiza e avalia a execução dos testes. Além disso, é evidente que um teste automatizado depende da especificação. Portanto, se esta muda, os testes e os programas afetados devem ser mudados também. Se mudanças forem muito freqüentes, teremos uma adição significativa de custo [FG 1999] para manter as várias massas de teste.

Para se poder confiar nos testes, é necessário que as massas de teste sejam de boa qualidade. Pode-se verificar a qualidade das massas de teste instrumentando-se os módulos em teste. Dessa forma obtém-se diversas medidas quanto à realização dos testes. Apesar das inerentes limitações dessa técnica, ela apresenta resultados satisfatórios e é de baixo custo. No presente arcabouço o instrumento de controle da completeza dos testes é disponibilizado através de um módulo que implementa as operações de medição e outro módulo que implementa o interpretador de comandos de teste específicos para este controle.

Muitas linguagens oferecem recursos de elevado risco e que tendem a dificultar a diagnose dos problemas encontrados. Em particular, linguagens tais como C e C++ permitem a alocação dinâmica de espaços de dados, a manipulação explícita de ponteiros em expressões complexas, a tipagem dinâmica (*type cast*) dos espaços apontados por ponteiros e, finalmente, não controlam o acesso a dados fora dos limites dos correspondentes espaços de dados. Se por um lado essas características dão uma grande flexibilidade aos programas, por outro lado, são causa de inúmeros problemas de difícil diagnose. O mau uso de ponteiros é a fonte de inúmeras dores de cabeça para os desenvolvedores e usuários de software. Entretanto, nem sempre um programador estará consciente do fato de que está fazendo mau uso de determinado ponteiro. Mais uma vez pode-se auxiliá-lo através do emprego de instrumentos. Nesse caso a instrumentação inserida no código perfaz o controle do acesso a espaços dinâmicos. A arquitetura usada é similar à utilizada para os instrumentos de apoio à medição da completeza dos testes.

1.1 Organização deste documento

No capítulo 2 apresentamos em linhas gerais como instalar e utilizar o arcabouço. No capítulo 3 apresentamos um resumo dos principais conceitos de teste de programas. Estes

[13] Um software é dito ***fidedigno*** (inglês: *dependable*) quando se pode *justificavelmente* depender dele assumindo riscos de danos compatíveis com o serviço por ele prestado.

são necessários para a compreensão do presente texto. Detalhes podem ser encontrados no livro Programação Modular [Staa 2000]. O capítulo 4 apresenta o exemplo utilizado neste documento. O capítulo 5 discute uma forma tradicional de teste manual. O objetivo é identificar as características que um teste automatizado deve satisfazer. O capítulo 6 é discutida uma forma de criar testes automatizados utilizando um módulo de teste específico em que os casos de teste são redigidos na linguagem de programação utilizada pelo módulo a testar. Esta forma de teste é adaptada do que é proposto em [JUnit]. No capítulo 7 é apresentada a forma de teste automatizado dirigida por uma linguagem simples de diretivas de teste. Neste capítulo são explorados os conceitos básicos de teste dirigido por diretivas. No capítulo 8 é apresentada a arquitetura completa do arcabouço de teste `AutoTest`. No capítulo 9 é discutida uma forma de se medir a completeza dos testes. No capítulo 10 é apresentado um módulo que monitora o acesso a memória dinâmica e que é capaz de simular falta de memória. Este módulo pode ser entendido como uma forma de "*mock object*" [HT 2003]. No capítulo 11 é apresentada uma proposta de processo de desenvolvimento de módulos e programas compostos por módulos em que se utilize teste automatizado e desenvolvimento incremental, tanto do conjunto de módulos, como dos módulos em si. Finalmente no capítulo 12 é apresentado o fecho deste documento.

2 Uso do arcabouço

Acompanha este documento o arcabouço de apoio ao teste `AutoTest`. Este se encontra no arquivo `ArcaboucoTeste-2-00.zip`. Para instalá-lo, crie um diretório (pasta) e *dezipe* o arquivo neste diretório. O arcabouço contém 3 exemplos:

`.\Simples` ilustra o uso básico do arcabouço de apoio ao teste automatizado.

`.\Arcabouc` ilustra a criação e o teste do arcabouço completo. O arcabouço completo contém, além dos módulos básicos de apoio ao teste automatizado, ainda um módulo de controle da completeza dos testes e outro de controle do acesso a memória dinâmica. Os exemplos contidos em `\Arcabouc` e `\Instrum` ilustram também o uso de uma estrutura de diretórios em que cada diretório contém o arquivos de um determinado tipo.

`.\Instrum` ilustra o uso do arcabouço completo.

`.\Tabela` ilustra o uso do arcabouço para o desenvolvimento e teste de um programa facilmente localizável. Chama-se de localização de um programa a tradução de suas mensagens e interfaces para diferentes culturas. O exemplo ilustra como tornar um programa localizável em culturas que utilizem o alfabeto latino. Consegue isto através de uma tabela contendo todas as mensagens geradas pelo programa.

Para que se possa utilizar a biblioteca, é necessário que ela tenha sido criada com a mesma versão do compilador que a que será utilizada para desenvolver os módulos do projeto. As instruções a respeito encontram-se no documento leia.me.

Para verificar se a instalação está completa e correta, torne o diretório `.\Tabela` o diretório corrente e ative o *batch file* `Tudo.bat`. É provável que seja necessário acertar parâmetros de ambiente (*set*) requeridos pelo compilador. É possível também ser necessário ajustar os parâmetros dos arquivos (`.comp` e `.parm`) usados pelo gerador de *scripts* de `make GMAKE`. O

material distribuído contém o GMAKE e a respectiva documentação. O arcabouço vem inicializado para operar corretamente com o compilador C/C++ do Visual Studio da Microsoft.

Para utilizar o arcabouço em projetos próprios, crie o diretório do projeto, copie para este diretório (ou sub-diretório):

- a biblioteca `.\obj\ArcaboucoTeste.lib`
- de `.\fontes` os módulos de definição
  - `CESPDIN.H`
  - `CONTA.H`
  - `GENERICO.H`
  - `LERPARM.H`
  - `TST_ESPC.H`
- Se for utilizar `GMAKE`, copie de `.\Ferramnt` os arquivos
  - `compilebanner.exe`
  - `exbestat.exe`
  - `gmake.exe`
  - `gmake.parm` e ajuste as referências a arquivos nele contido para ficarem consistentes com o padrão de estrutura de diretórios do projeto.

Ao compilar, assegure-se que a biblioteca `ArcaboucoTeste.lib` esteja incluída na lista de arquivos fornecidos ao ligador `LINK`. Note que o programa principal faz parte da biblioteca.

3 Teste de módulos, resumo

Nesta seção apresentaremos, de forma bastante resumida, os conceitos de teste necessários para o entendimento da automação dos testes. Uma descrição mais detalhada pode ser encontrada em [Staa 2000].

Qualquer teste depende de um padrão com o qual se deve comparar o comportamento do construto. No caso de testes de corretude este padrão é a especificação do módulo a testar. O objetivo do teste passa a ser: exercitar o módulo a testar de modo que seja encontrado o maior número possível (idealmente todos[14]) de problemas que o módulo em teste poderia apresentar.

Para avaliar a corretude de um programa composto por módulos evidentemente é necessário, entre outras coisas, o teste rigoroso de cada módulo [FO 2000]. Integrar programas compostos por módulos de qualidade duvidosa, aumenta muito o risco do programa vir a falhar, possivelmente gerando danos de grande envergadura. Além disso a identificação das faltas (diagnose) a partir de relatórios de problemas encontrados em um programa composto por vários módulos tende a ser muito custosa e desgastante para o

[14] Segundo Dijkstra [DDH 1972] testes somente são capazes de mostrar a *presença* de faltas, mas *não a ausência* delas. Ou seja, usando exclusivamente testes, nunca poderemos assegurar que o programa esteja 100% correto (para todos os efeitos práticos não existe teste exaustivo!). No entanto, através de vários instrumentos, entre eles testes, podemos chegar bem perto. Uma parte significativa da indústria consegue produzir programas com, em média, cerca de 1 linha de código em erro para mais de 10.000 linhas de código puro entregues. ***Código puro***: sem contar as linhas em branco e as de comentários.

fornecedor do programa [BB 2001], uma vez que precisa identificar exatamente os módulos faltosos e, dentro desses, todas as faltas responsáveis pelo problema. A dificuldade é agravada por essas descrições serem, quase sempre, relatos vagos e/ou confusos do problema observado [Kaner 2000].

Durante o teste de um módulo procura-se encontrar problemas decorrentes de uma implementação ou especificação incorreta. Em alguns casos também é interessante verificar as conseqüências do uso incorreto do módulo. Isto permite verificar se o módulo é robusto[15]. Em todos esses casos cria-se um módulo de controle do teste (*driver*) cuja finalidade é exercitar o módulo em teste, ver Figura 1. Para assegurar que o controlador de teste não interfira no módulo a testar, o exercício deste deve ser realizado estritamente através da interface do módulo a testar. Desta forma, ao retirar o módulo de controle dos testes, o módulo testado não sofre alterações.

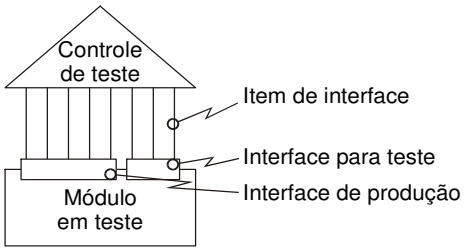

Figura 1. Interação entre o módulo de controle e do módulo em teste

São exemplos de itens da interface em um programa C:

- funções globais exportadas;
- dados globais exportados;
- arquivos manipulados;
- mensagens transmitidas ou recebidas;
- interface com o usuário (recepção de dados e comandos, exibição de resultados, mensagens e auxílio interativo);
- estados manipulados. Por exemplo, ao empilhar um elemento em uma pilha, muda-se o seu estado de modo que o novo elemento esteja no topo da pilha. Isto será assim independentemente da implementação e do fato da estrutura de dados da pilha ser ou não encapsulada. Um outro exemplo, uma árvore pode não existir, pode estar vazia, determinado nó pode ou não conter um ponteiro para a esquerda ou para a direita.

Ao testar um módulo deve-se verificar, para cada função da interface, se esta se comporta conforme especificado e, para cada dado externado através da interface, se este contém valores conforme esperado. Também deve ser verificado se o estado corrente do módulo é condizente com o que se espera. O estado do módulo é caracterizado pelos dados que contém e pelas estruturas de dados que implementa. Mais difícil tendem a ser os controles de arquivos e bases de dados gerados ou alterados pelo módulo. Neste caso será necessário

[15] Um programa (módulo) ***robusto*** é capaz de observar que está operando ou sendo usado de forma incorreta, interceptando a execução de forma a impedir que o programa gere ou propague danos vultosos. Um programa ***tolerante a falhas*** é capaz de corrigir as falhas para depois retomar a execução normal de forma confiável, mesmo que restringindo as classes, ou o volume, ou a qualidade dos serviços capaz de prestar.

desenvolver módulos ou funções capazes de examinar o conteúdo destes artefatos com o intuito de verificar se é o esperado ou se satisfaz as suas assertivas estruturais. Interfaces gráficas requerem um exame visual cuidadoso na busca de discrepâncias com relação ao que deveriam conter. Isto é particularmente complicado quando estiverem sendo exibidas figuras ou desenhos. Finalmente, mensagens requerem que se verifique se o que está sendo enviado e recebido é o que deveria ser.

Em algumas situações torna-se necessário testar em detalhe características internas ao módulo. Neste caso necessita-se de uma interface adicional especificamente projetada para fins de teste. De maneira geral, a interface de teste permite interagir com a instrumentação [Staa 2000] cuja finalidade é monitorar a execução, a completeza dos testes e a confiabilidade da instrumentação. Ou seja, a interface de teste não deve prover mecanismos que possam interferir na funcionalidade do módulo, exceto quando se utiliza uma técnica baseada em mutantes de código [DMM 2001] ou de dados [Staa 2000]. Deve sempre ser possível remover essa interface sem que isso afete o comportamento da porção útil do módulo, veja a atividade *Reduzir instrumentação* na Figura 2. Em virtude da possibilidade de ter-se que realizar testes de regressão, a interface deve ser capaz de ser reintroduzida antes desses testes. Ou seja, a interface de teste não deve prover mecanismos que possam interferir na funcionalidade do módulo quando posto em produção, tampouco deve ser removida fisicamente do código. Consegue-se isso através do uso da compilação condicional do código de instrumentação (`#if`, `#ifdef`, `#else`, `#endif`).

Conseqüentemente, o teste deverá ser realizado em pelo menos duas etapas, uma em que a instrumentação deverá estar presente no módulo a testar e, a outra, em que não faz parte deste. O teste em duas etapas é necessário para verificar se a remoção da instrumentação introduziu ou não problemas no módulo em teste. Conseqüentemente, são necessários dois construtos, dois controladores e duas massas de teste, a primeira testa completamente o módulo inclusive a sua instrumentação, enquanto que a segunda testa o módulo estritamente através de sua interface de produção sem requerer acesso à instrumentação.

Não basta criar uns tantos casos de teste sem seguir os ditames de um método de seleção de casos de teste. Ao agir desta forma, criam-se testes que verificam determinadas situações, mas deixam de examinar uma quantidade em geral grande de outras possíveis situações. É necessário então usar algum critério para selecionar os casos de teste.

Uma massa de teste é um conjunto de casos de teste e tem por objetivo testar completamente o módulo segundo uma determinada perspectiva. Para criar uma massa de teste utilizam-se critérios de seleção de casos de teste. Existe uma grande variedade de critérios de seleção de casos de teste [FG 1999, KFN 1988, Nguyen 2001, SJ 2001, Staa 2000], cada qual com capacidades específicas de identificar problemas porventura contidos no módulo sendo testado. Todos esses critérios visam definir massas de teste que, supostamente, testam integralmente o módulo segundo uma determinada perspectiva.

Mesmo utilizando critérios, não basta realizar os testes para adquirir confiança no módulo, é necessário verificar se o conjunto de casos de teste é suficiente para servir como um indicador confiável da qualidade do módulo testado [WV 2000]. Evidentemente, pode-se procurar verificar essa suficiência através de inspeções [Laitenberger 2002, SRB 2000], tanto do processo utilizado para gerar as massa de teste, como da massa de teste em si. Infelizmente, inspeções, por serem realizadas por seres humanos, têm uma boa chance de não serem realizadas, ou o serem de forma incompleta. Mesmo assim inspeções têm-se mostrado bastante eficazes, uma vez que, quando praticadas, identificam entre 60% e 75%

do total de faltas encontradas no decorrer do projeto [BB 2001]. Surge então a necessidade de se utilizar alguma forma de medição para verificar a suficiência dos testes.

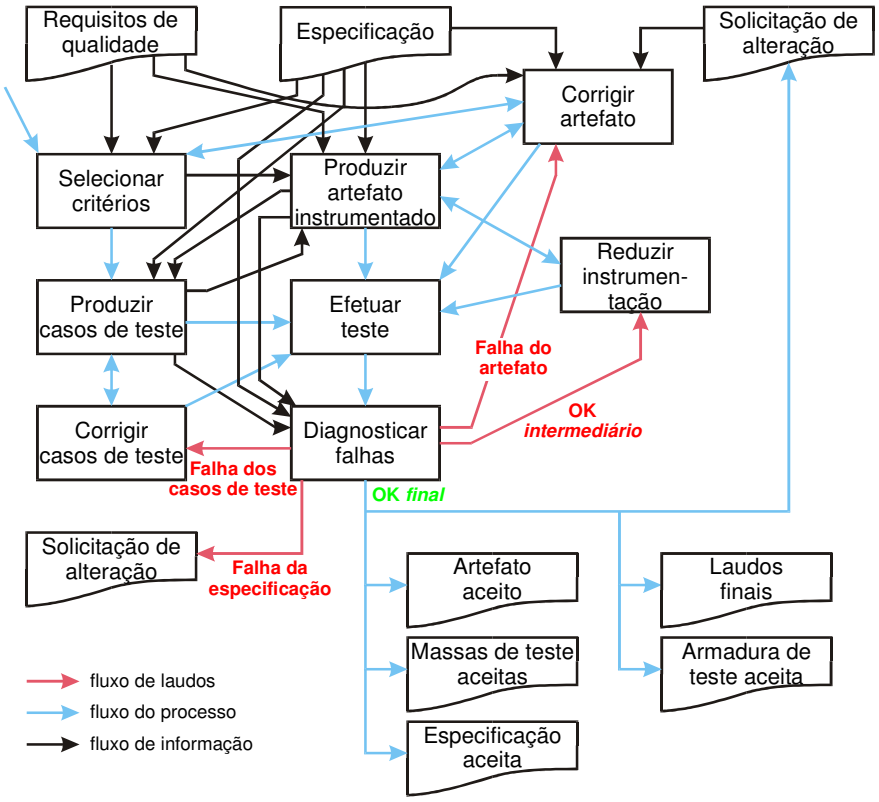

Figura 2. Visão macroscópica do processo de teste de artefatos.

A confiabilidade de um teste depende da qualidade da massa de teste utilizada. Uma massa de teste pouco rigorosa requer pouco esforço para ser criada e realizada, infelizmente conduz a uma baixa confiança na qualidade do artefato testado. Já uma massa de teste muito rigorosa pode conduzir a uma elevada confiança nessa qualidade, embora às custas de um esforço significativamente maior. Infelizmente, conforme apontou Dijkstra [DDH 1972], confiança total (certeza da ausência de faltas) raras vezes pode ser alcançada[16].

Conforme ilustrado na Figura 2, ao concluir a atividade *Diagnosticar falhas* do teste automatizado de um módulo teremos:

- o ***módulo aceito*** *segundo os testes realizados*. A aceitação será *segundo os testes*, uma vez que uma outra escolha de dados e comandos de teste poderia levar à descoberta de problemas antes desconhecidos.
- a ***massa de teste aceita***. Essa massa de teste deve ser cuidadosamente armazenada para que possa ser reutilizada em testes de regressão futuros e, ainda, para que sirva de ponto de partida ao gerar a massa de teste a ser usada com uma evolução do módulo.

[16] Em [GY 1976] é defendida a tese que, apesar de suas restrições, testes são necessários, uma vez que a prova formal da corretude, além de muito custosa, também não é suficientemente confiável.

- a ***especificação aceita***[17]. Ao final do processo de teste deve-se ser capaz de demonstrar que o módulo corresponde exatamente [18]à sua especificação. Conseqüentemente, se tiverem ocorrido mudanças na especificação estas devem estar devidamente incorporadas à especificação entregue (usualmente o módulo de definição). Ao utilizar métodos de desenvolvimento ágeis, as massas de teste devem ser desenvolvidas antes de se desenvolver os módulos [Beck 2000, Cockburn 2002]. Dessa forma a massa de teste passa a ser uma especificação (parcial) executável em que cada caso de teste constitui um exemplo da funcionalidade esperada.
- o ***arcabouço de teste aceito*** para a realização *desses testes*. Outra escolha de dados pode requerer novos cenários de teste, o que, por sua vez, poderia levar a ter-se que modificar componentes do arcabouço de teste.
- o ***conjunto de laudos*** produzidos ao testar, sendo que o mais recente deles estabelece a ausência de faltas não toleradas.[19] Possivelmente, alguns dos laudos conterão evidências de falhas. Para que o construto possa ser aceito é necessário que essas falhas sejam transformadas em solicitações de alteração. Algumas destas serão, na realidade relatórios de problemas pendentes.
- Armadura de teste aceita. Uma armadura de teste é um conjunto de artefatos, possivelmente contendo hardware, cuja finalidade é simular o contexto no qual reside o módulo ou componente sendo testado. Por exemplo, ao testar um módulo que se comunica com um *browser*, pode-se criar um outro módulo que simula [HT 2003] o comportamento do *browser* e que é capaz de analisar o que foi recebido com o intuito de verificar se equivale o que se esperava receber. Este mesmo módulo pode simular a edição de dados e transmiti-los de volta ao servidor.

É dito por Kaner e outros em [KFN 1988]:

> *Realistic test planning is dominated by the need to select a few test cases from a huge set of possibilities. No matter how hard you try, you will miss important tests. No matter how careful and thorough a job you do, you will never find the last bug in a program, or if you do, you will not know it.*

Infelizmente, todos estes resultados não são indicação da suficiência dos testes. Ou seja, é perfeitamente possível que todos os itens da interface sejam dados como satisfatoriamente implementados. Como testes se baseiam na escolha de alguns (muito poucos) exemplos do conjunto de todos os exemplos possíveis (possivelmente infinitos), eles são muito sensíveis à escolha. Ou seja, diferentes escolhas podem levar à identificação de diferentes faltas, ou mesmo a não acusar a existência de faltas efetivamente contidas no código. Esta deficiência dos testes será tão maior quanto menos criteriosa tiver sido a escolha dos casos de teste.

4 Apresentação do módulo exemplo

Nesta seção apresentaremos um módulo singelo que servirá de exemplo para ilustrar os conceitos discutidos no restante do artigo. O módulo exemplo tem por objetivo criar, alterar,

---

[17] Na verdade não necessariamente estará aceita toda a especificação, mas, sim, somente os itens da especificação do módulo que conduzem a algum código executável.

[18] Implementa tudo o que foi especificada e nada além do que foi especificado.

[19] Em algumas situações faltas conhecidas podem permanecer no código – as ***faltas toleradas*** –, desde que devidamente documentadas e/ou encapsuladas em código preventivo. Por exemplo, uma opção de menu que exercite uma falta conhecida, poderia ser desativada até uma nova liberação do módulo ser concluída.

explorar e destruir árvores binárias. A Figura 3 apresenta o modelo da estrutura e dois exemplos de árvores binárias [Staa 2000].

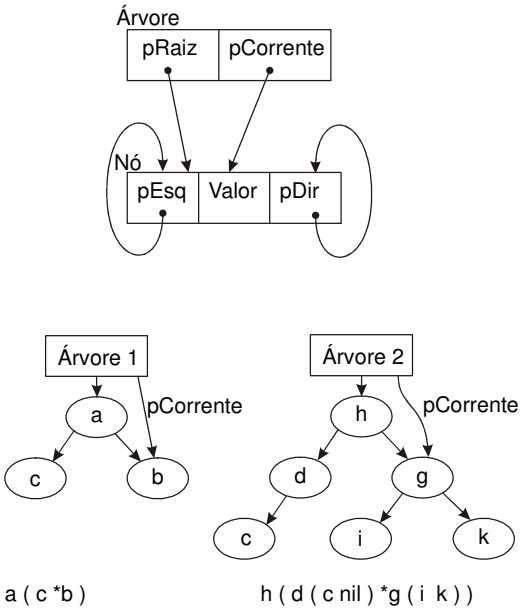

Figura 3. Modelo e exemplos de árvores usadas neste texto

O código fonte do arcabouço e de exemplos de seu uso é disponibilizado. Este código contém duas implementações do módulo árvore, uma simples e outra contendo instrumentação de apoio ao teste e capaz de manipular simultaneamente várias árvores binárias.. Nesta seção discutiremos a forma simples. O código a seguir corresponde ao módulo de definição[20] C (*header file*, módulo de declaração). Conforme estabelecido em [Staa 2000] o módulo de definição contém tanto a especificação como a declaração de todos itens de interface.

```
#if ! defined( ARVORE_ )
#define ARVORE_
/***************************************************************************
*
*  $MCD Módulo de definição: Módulo árvore
*
*  Arquivo gerado:              ARVORE.H
*  Letras identificadoras:      ARV
*
*  Projeto: Disciplinas INF 1628 / 1301
*  Gestor:  DI/PUC-Rio
*  Autores: avs - Arndt von Staa
*
*  $ED Descrição do módulo
*     Este módulo implementa um conjunto simples de funções para criar e
*     explorar uma árvore binária.
*     A árvore possui uma cabeça que contém uma referência para a raíz da
*     árvore e outra para um nó corrente da árvore.
```

[20] Na linguagem C os módulos devem ser compostos por dois arquivos de código fonte. Um deles, o ***módulo de definição*** (`.h`), declara todos os tipos, constantes, dados e funções que estão na *interface* do módulo. Contém também a documentação desta interface destinada ao programador cliente do módulo bem como ao desenvolvedor. O segundo arquivo, o ***módulo de implementação*** (`.c`), contém as declarações dos tipos, constantes, dados e funções *encapsuladas*. Contém ainda o código de implementação de todas as funções, encapsuladas ou não. Contém também a documentação encapsulada destinada ao desenvolvedor. Finalmente, contém as inclusões (`#include`) necessárias para que possa ser submetido com sucesso ao compilador.

```
*     A árvore poderá estar vazia. Neste caso a raiz e o nó corrente
*     serão nulos, embora a cabeça esteja definida.
*     O nó corrente será nulo se e somente se a árvore estiver vazia.
*
***************************************************************************/

#if defined( ARVORE_OWN )
   #define ARVORE_EXT
#else
   #define ARVORE_EXT extern
#endif

/***************************************************************************
*
*  $TC Tipo de dados: ARV Condições de retorno
*
***************************************************************************/

   typedef enum {
         ARV_CondRetOK ,
               /* 0 - Executou correto */
         ARV_CondRetNaoRaiz ,
               /* 1 - Não criou nó raiz */
         ARV_CondRetEstrutura ,
               /* 2 - Estrutura da árvore está errada */
         ARV_CondRetNaoFolha ,
               /* 3 - Nó não é folha relativa à inserção desejada */
         ARV_CondRetNaoArvore ,
               /* 4 - Árvore não existe */
         ARV_CondRetNaoCorr ,
               /* 5 - Árvore está vazia */
         ARV_CondRetEhRaiz ,
               /* 6 - Nó corrente é raiz */
         ARV_CondRetNaoFilho ,
               /* 7 - Nó corrente não possui filho na direção desejada */
         ARV_CondRetMemoria
               /* 8 - Faltou memória ao alocar dados */
   } ARV_tpCondRet ;

/***************************************************************************
*
*  $FC Função: ARV Criar árvore
*
*  $ED Descrição da função
*     Cria uma nova árvore vazia.
*     Caso já exista uma árvore, esta será destruída.
*
***************************************************************************/

   ARV_tpCondRet ARV_CriarArvore( void ) ;

/***************************************************************************
*
*  $FC Função: ARV Destruir árvore
*
*  $ED Descrição da função
*     Destrói o corpo e a cabeça da árvore, anulando a árvore corrente.
*     Faz nada caso a árvore corrente não exista.
*
***************************************************************************/

   void ARV_DestruirArvore( void ) ;

/***************************************************************************
*
*  $FC Função: ARV Adicionar filho à esquerda
*
*  $EP Parâmetros
*     $P ValorParm - valor a ser inserido no novo nó.
```

```
*
*  $FV Valor retornado
*     ARV_CondRetFolha     - caso não seja folha para a esquerda
*
***************************************************************************/

   ARV_tpCondRet ARV_InserirEsquerda( char ValorParm ) ;

/***************************************************************************
*
*  $FC Função: ARV Adicionar filho à direita
*
*  $EP Parâmetros
*     $P ValorParm - valor a ser inserido no novo nó
*
*  $FV Valor retornado
*     ARV_CondRetFolha     - caso não seja folha para a direita
*
***************************************************************************/

   ARV_tpCondRet ARV_InserirDireita( char ValorParm ) ;

/***************************************************************************
*
*  $FC Função: ARV Ir para nó pai
*
***************************************************************************/

   ARV_tpCondRet ARV_IrPai( void ) ;

/***************************************************************************
*
*  $FC Função: ARV Ir para nó à esquerda
*
***************************************************************************/

   ARV_tpCondRet ARV_IrNoEsquerda( void ) ;

/***************************************************************************
*
*  $FC Função: ARV Ir para nó à direita
*
*  $FV Valor retornado
*     ARV_CondRetNaoFilho   - nó corrente não possui filho à direita
*
***************************************************************************/

   ARV_tpCondRet ARV_IrNoDireita( void ) ;

/***************************************************************************
*
*  $FC Função: ARV Obter valor corrente
*
*  $EP Parâmetros
*     $P ValorParm - é o parâmetro que receberá o valor contido no nó.
*                    Este parâmetro é passado por referência.
*
***************************************************************************/

   ARV_tpCondRet ARV_ObterValorCorr( char * ValorParm ) ;

#undef ARVORE_EXT

/********** Fim do módulo de definição: Módulo árvore **********/

#else
#endif
```

5 Teste manual de módulos

Nesta seção discutiremos resumidamente como se processa o teste manual de módulos. O objetivo é identificar as ações passíveis de automação.

No teste manual de módulos, o módulo de controle específico contém usualmente um menu para a seleção do item da interface a ser testado. Uma vez selecionada uma função a testar, são solicitados os dados necessários para ativar a função. A seguir a função é executada com estes dados e o resultado é exibido de alguma forma, de preferência de modo que seja facilmente entendida pelo testador[21]. Caso seja selecionada uma interação com um dado global, o menu de teste deverá permitir a seleção de ações que acessam e exibem ou alteram estes dados. Após ter realizado uma ação de teste, repete-se a escolha de novas ações, até que o testador selecione a ação de término do teste. A seguir ilustramos um possível menu de teste:

```
 1 Criar árvore
 2 Destruir árvore
 3 Inserir nó à esquerda
 4 Inserir nó à direita
 5 Ir para nó filho à esquerda
 6 Ir para nó filho à direita
 7 Ir para nó pai
 8 Obter valor do nó corrente
 9 Exibir a árvore em formato parentetizado
99 Terminar

Escolha a opção:
```

Escolhendo, por exemplo, a opção 3, deve ser fornecido o valor a ser inserido no novo nó, a função é executada, a condição de retorno é exibida e volta-se ao início do menu de seleção. O testador deve verificar se o valor retornado corresponde ao que era esperado. Para saber se o dado foi corretamente inserido é necessário efetuar pelo menos a ação 8 com relação ao novo nó corrente. Na realidade ele deveria explorar os nós na vizinhança do nó inserido, pois de outra forma não poderá justificar que a inserção tenha se dado corretamente. Desnecessário dizer que o testador precisa manter um desenho do estado da árvore para que possa acompanhar o desenrolar do teste. Ao terminar uma ação pode-se também exibir uma forma parentetizada da árvore. A seguir ilustramos uma possibilidade de fazer isto, considerando os exemplos da Figura 3:

```
a ( b *c )
h ( d ( c nil ) *g ( i k ))
```

Nos exemplos: `nil` significa que o nó não existe na árvore e o asterisco ('`*`') precede o nó corrente.

Uma variante mais bonita, porém ***muito*** mais custosa, é desenvolver uma interface gráfica (GUI – Windows) na qual se seleciona a ação, fornecem-se os dados necessários e exibem-se os resultados obtidos. Entretanto, de maneira geral não vale a pena enfeitar os controladores de teste, uma vez que isto somente acarreta mais custos sem trazer benefícios compatíveis.

O teste manual oferece várias vantagens:

---

[21] ***Testador*** é a pessoa ou equipe responsável pela realização dos testes. O testador pode ser o próprio desenvolvedor ou uma pessoa ou equipe não relacionada como o desenvolvimento do artefato a testar..

- É relativamente simples e barato de programar.
- É virtualmente irrestrito quanto à interface com as funções a testar e com relação aos resultados apresentados. Por exemplo, o aspecto de interfaces gráficas, tais como posicionamento dos *widgets*[22], seleção de cores, *rendering* (geração da imagem gráfica) e truncamentos gráficos, são usualmente mais fáceis de controlar visualmente do que por programas;
- É mais fácil verificar a corretude caso esta se baseie em valores aproximados. Evidentemente, isto requer que o testador saiba determinar quais as aproximações aceitáveis.

Por outro lado possui grandes desvantagens:

- O confronto visual de resultado esperado com resultado obtido é altamente sujeito a erros humanos. Freqüentemente o testador *acha* que os resultados são iguais aos esperados, ou aproximadamente iguais, ou satisfatórios, quando um exame mais cuidadoso mostra que não são.
- Massas de teste envolvendo muitos casos de teste são enfadonhas e caras de realizar manualmente. Conseqüentemente acabam não sendo executadas de forma completa ou com o rigor necessário. Como já foi dito, isto tende a levar a programas de baixa qualidade.
- Como o testador fornece os dados durante uma sessão de teste, é comum que trabalhe com dados *inventados na hora*, ao invés de criar uma massa de teste baseada em algum critério de seleção. Mais uma vez, isto contribui para programas de baixa qualidade além de encarecer os testes, uma vez que massas de teste criadas sem cuidado tendem a retestar várias vezes uma mesma condição, contribuindo nada para a observação de uma nova falha cuja causa (falta) seja diferente das anteriormente identificadas.
- O teste de regressão manual tende a examinar somente as funções que foram modificadas, sem examinar se estas modificações introduziram anomalias em porções do programa que pela lógica não deveriam ser afetadas por essas modificações. A conseqüência é um programa de baixa qualidade e que representa um desafio de diagnose para o testador quando falhas forem reportadas pelo usuário (pessoa) do programa em uso produtivo.

Em virtude dos problemas acima identificados, o *teste manual* ***nunca*** *poderá ser utilizado como um atestado de qualidade*. Mesmo que esteja definido um roteiro de teste cuidadoso, com casos de teste escolhidos segundo o melhor dos critérios, como o teste depende da ação humana, está-se sujeito a erros humanos. Como o processo de teste não é confiável, não se pode dizer nada quanto à esperança do programa estar correto. Tampouco adianta auditar a documentação dos testes, pois não se sabe se a massa de teste (o roteiro de teste) foi obedecida rigorosamente.

---

[22] ***Widgets*** são os elementos componentes de uma janela. Exemplos: barras de rolagem, menus, botões, ícones, janelas de edição, janelas de seleção, textos.

6 Teste de módulos usando funções de teste

Nesta seção mostraremos como automatizar os testes utilizando funções de teste especificamente desenvolvidas para este fim. Identificaremos também parte da composição do módulo de controle genérico que poderá ser reutilizado nos vários testes.

Conforme proposto por Beck [Beck 2000, JUnit, CPPUnit] pode-se automatizar os testes através de uma função ou de um módulo de teste específico, contendo código de chamada para ações de teste e código para estabelecer o contexto de teste. Beck [Beck 2000, Wake 2001] inclusive recomenda que primeiro se redija as funções de teste e, depois, o módulo a testar. Segundo ele, isto tem as vantagens:

- ajuda o programador a entender o que é desejado que ele programe antes de perder horas resolvendo o problema errado.
- não vicia o programador a assumir o programa dele como sendo a versão correta da especificação. Uma das falhas humanas mais comuns é *achar* que alguma coisa está definida de determinada maneira, quando na realidade não está. Através de uma cuidadosa leitura da especificação estes erros seriam facilmente sanados. Porém, de tanto lidar com ela, o programador acaba decorando incorretamente a especificação. Quando isto acontece, ele acabará produzindo casos de teste em conformidade com o que ele *programou* e não com o que ele *deveria ter programado*.
- serve como instrumento de verificação da completeza do programa. Enquanto o arcabouço de teste acusar erros de omissão, o programador sabe que ainda tem coisas a resolver e quais são.

Um exemplo inicial de função de teste pode ser:

```
ContaCaso ++ ;
if (CriarArvore( ) != ARV_CondRetOK )
{
   printf( "\nErro ao criar árvore" ) ;
   ContaFalhas ++ ;
}
ContaCaso ++ ;
if (InserirEsquerda( 'a' ) != ARV_CondRetOK )
{
   printf( "\nErro ao inserir nó raiz da árvore" ) ;
   ContaFalhas ++ ;
}
ContaCaso ++ ;
if (IrPai( ) != ARV_CondRetEhRaiz )
{
   printf( "\nErro ao ir para pai de nó raiz" ) ;
   ContaFalhas ++ ;
}
ContaCaso ++ ;
if ( ObterValor( ) != 'a' )
{
   printf( "\nNó corrente não contém valor esperado" ) ;
   ContaFalhas ++ ;
}
. . .
```

Este código tende a ser extenso e a conter um sem-número de linhas de código repetidas. Uma solução para isto é desenvolver uma biblioteca de funções de comparação e incorporá-la ao módulo de teste genérico. Desta forma estas funções de comparação ou apoio passam a fazer parte do arcabouço, podendo ser reutilizadas em todos os construtos e todos os

programas desenvolvidos na organização. Por exemplo, podem ser desenvolvidas diversas funções similares a:

```
void CompararInt( int ValorObtido ,
                  int ValorEsperado ,
                  int NumeroLinha ,
                  char * Mensagem )
```

Na qual:

ValorObtido é o valor retornado pela função sendo testada. Ou é o valor que deveria estar contido em alguma variável, elemento de vetor ou de estrutura;

ValorEsperado é o valor esperado neste caso de teste;

NumeroLinha é o número da linha de código da função de teste. É obtido com a "constante" padrão de C: `__LINE__` O objetivo de expor a linha do código fonte é facilitar a localização do fragmento de código que identificou a falha. Desta forma pode-se determinar com mais rapidez a causa desta falha.

Mensagem é a mensagem de erro que deve ser exibida caso os valores não sejam iguais.

A função conta o número de casos de teste, adicionando 1 a cada vez que for chamada. A função conta, também, o número de falhas observadas, adicionando 1 sempre que o resultado esperado for diferente do obtido. A mensagem será impressa sempre que os resultados esperado e obtido sejam discrepantes. Um exemplo de formato de saída é:

```
>>> Linha <numeroLinha> Falha <contaFalha> <Mensagem>
          Valor é: <ValorObtido> Deveria ser <ValorEsperado>
```

Desta forma somente precisam ser impressos os casos de teste que tenham identificado uma falha, reduzindo o trabalho de separar os acertos dos erros. Tem-se, ainda, a vantagem de saber a linha do código de teste onde ocorreu o problema e o resultado obtido. Muitas vezes o problema encontrado não está no módulo a ser testado e sim no módulo de teste. Além disso, saber o resultado obtido é sempre uma ajuda para determinar a causa (falta) da falha observada.

Tendo-se desenvolvido tais funções o fragmento da função de teste de criar o nó raiz de uma árvore passa a ser:

```
VerificarInt( CriarArvore( ), ARV_CondRetOK , __LINE__ ,
              "Retorno errado ao criar árvore.") ;
VerificarInt( InserirEsquerda('a' ), ARV_CondRetOK , __LINE__ ,
              "Retorno errado ao inserir raiz.") ;
VerificarInt( IrPai( ), ARV_CondRetEhRaiz, __LINE__ ,
              "Retorno errado ao caminhar para pai de raiz.") ;
VerificarChar( ObterValor( ) , 'a' , __LINE__ ,
              "Valor do nó raiz está errado." ) ;
. . .
```

Ao terminar o teste pode-se chamar uma função de biblioteca `ExibirResumo( )` que exibe o número de casos de teste e o número de falhas identificadas. Todas as funções descritas até aqui podem ser redigidas de tal forma que os resultados exibidos possam ser dirigidos ou para a console (tela ou janela) ou para um arquivo *log*. Caso os resultados sejam enviados para um *log*, pode-se evidenciar, por meios mecânicos, que os testes foram integralmente efetuados e quais os resultados do teste. O *log* corresponde a um ***laudo do teste***, nele estarão

registradas todas as falhas encontradas ao testar. Através da inspeção do módulo de teste específico pode-se verificar se está completo e se obedece rigorosamente aos critérios de seleção de casos de teste indicados. Ou seja, esta forma de realizar os testes pode servir como um atestado da qualidade do teste.

Um efeito colateral desta forma de testar é o teste implacavelmente apresentar ao desenvolvedor todas as falhas observadas, bem como onde, no código do módulo de teste específico, estas falhas foram encontradas. Isto facilita ao desenvolvedor diagnosticar e depurar, isto é, localizar as faltas, removê-las e refazer os testes com vistas a confirmar a completa remoção da falta, sem a introdução de novas faltas. Esta forma de teste é, então, um excelente *aliado* para o desenvolvedor, ao invés de ser um dos inúmeros *obstáculos* a serem vencidos ao desenvolver programas. Esta forma de teste também contribui para uma melhor avaliação da confiabilidade, uma vez que facilita a procura pró-ativa por falhas e a eliminação das correspondentes faltas [WV 2000].

Além das ações de teste, pode tornar-se necessário estabelecer um contexto. Por exemplo, caso se queira testar dados compostos (`struct`), pode ser melhor copiar o valor destes dados para uma área de trabalho e só então verificar, campo a campo, se os resultados obtidos são iguais aos esperados. Similarmente, quando se testa vários dados interrelacionados, pode ser melhor copiar a estrutura e depois verificar se está correta. Finalmente, pode ser necessário estabelecer um cenário para poder realizar os testes. Por exemplo, ao testar a função `MatricularEmDisciplina` precisa-se saber o histórico do aluno e o horário de oferecimento de disciplinas. Para verificar a corretude do tratamento das diversas formas de dados em erro, é necessário forjar históricos e horários correspondentes a estas situações de erro. Em outras ocasiões deseja-se verificar o que acontece se o sistema gerente da base de dados acusa determinado erro ao processar uma transação. É necessário, então, ser capaz de provocar erros deliberados na base de dados. Em geral utilizam-se módulos, ou mesmo componentes, especialmente projetados para facilitar a simulação destes comportamentos anômalos. Tais módulos ou componentes, quando desenvolvidos em programação orientada a objetos, são chamados de ***mock objects*** [HT 2003, capítulo *Mock Objects*].

O uso de funções de teste específicas estendendo um arcabouço de apoio ao teste apresenta várias vantagens:

- É possível gerar um código de teste tão extenso quanto necessário para que se disponha de uma massa de teste completa e confiável [GG 1977]. Na maioria dos casos o teste pode ser realizado com suficiente profundidade para que se possa ter elevada confiança na corretude do módulo em teste.
- É possível utilizar código redigido na linguagem de programação utilizada para desenvolver o módulo a testar. Além de tornar desnecessário aprender uma nova linguagem, isto também reduz a dificuldade de se estabelecer interfaces confiáveis. Além disso viabiliza a redação de funções auxiliares, repetições, seleções e tudo o mais que uma linguagem de programação oferece.

A técnica oferece também algumas desvantagens, entre elas:

- A função de teste específica tende a ser muito extensa e mal documentada, tornando difícil verificar a completeza e a eficácia do teste.
- As funções de teste específicas muitas vezes não seguem padrões, conseqüentemente cada uma delas tem um estilo próprio. Isto tende a dificultar a manutenção.

- As funções `VerificarXxx` acusam problemas, mesmo quando estes são esperados. Por exemplo, ao testar a instrumentação pode ser necessário deturpar[23] [Staa 2000] a estrutura controlada pela instrumentação de modo que se possa verificar se esta é suficientemente eficaz em determinar falhas estruturais. Quando se deturpa uma estrutura de dados é claro que as funções que controlam a sua validade vão acusar falhas. Entretanto, falha seria não ter acusado um problema. Conseqüentemente é necessário esperar que seja acusada a falha e a seguir ser capaz de recuperar o indicador de falhas para OK.
- Em virtude da sua extensão, os módulos de teste podem se tornar difíceis de manter, uma vez que misturam o assunto[24] (*concerns*) [OT 2001, TOHS 1999] de criar e manter o contexto (cenário do teste), com o de realizar um número suficiente de casos de teste.
- É difícil assegurar que as funções de teste não interfiram com o código em teste, ou mesmo o código já aprovado. Como as funções de teste têm acesso a todos os módulos, como são muito extensas, e como o controle de completeza e eficácia dos teste será sempre realizado de forma visual, pode-se tornar difícil observar a existência de uma ação implementada de forma perniciosa.

7 Teste de módulos utilizando arquivos de diretivas

Nesta seção discutiremos uma forma alternativa de automação de testes. Ao invés de se redigir um extenso módulo de controle de teste específico, redige-se um módulo específico que estabelece somente a interface com o módulo a testar. Esse módulo de controle interpreta as diretivas de teste que se encontram em um arquivo de diretivas (*script*). Para facilitar a compreensão, examinaremos nesta seção somente uma parte do arcabouço, mais adiante será descrita a arquitetura completa do arcabouço `AutoTest` de apoio ao teste automatizado. O arquivo de distribuição `TesteAutomatizado.zip`, diretório: `Simples`, contém os diversos arquivos que implementam o que está descrito neste capítulo.

7.1 Arquitetura simplificada

A arquitetura simplificada do arcabouço de apoio ao teste automatizado `AutoTest` é ilustrada na Figura 4. Ela combina alguns aspectos do teste manual com outros do teste automatizado utilizando um módulo de teste específico.

O *programa principal* é uma função (`main`) que faz parte do módulo `PRINCIP`. O programa principal lê os parâmetros de linha de comando e coordena a execução dos testes. Caso o desenvolvimento se dê incorporando o módulo de teste a um construto mais abrangente, pode-se chamar diretamente a função `TST_ControlarSessaoTeste` que faz parte do módulo de controle de teste `GENERICO`. Esta função poderá ser chamada de qualquer ponto no construto.

---

[23] Esta forma de teste é similar à baseada em ***mutantes*** [Delamaro 1997, DMM 2001], só que, ao invés de adulterar o código, adulteram-se os dados armazenados.

[24] Uma das preocupações da programação modular ou programação baseada em componentes é a ***separação de assuntos*** (*separation of concerns*). Idealmente cada módulo deve tratar de um único assunto. A composição de assuntos que formam um problema a resolver, é conseguida através da composição de módulos.

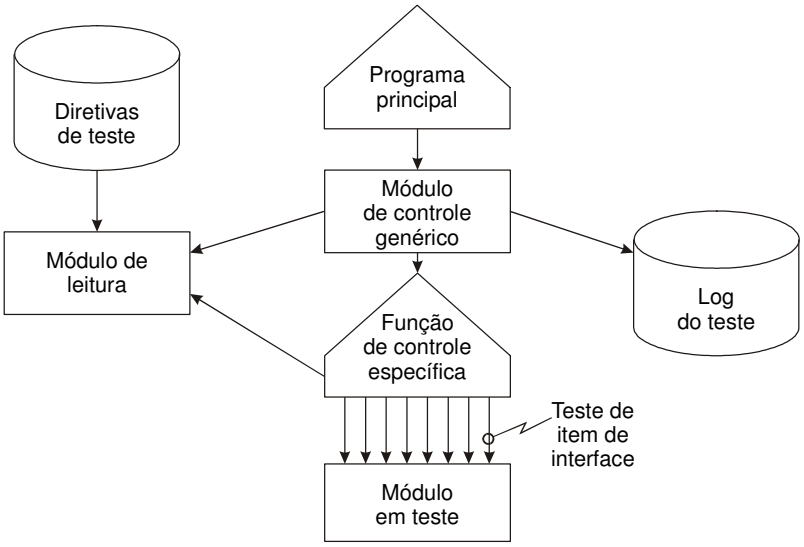

Figura 4. Arquitetura simplificada para teste utilizando arquivos de diretivas

O *módulo de leitura*, módulo `LERPARM`, disponibiliza uma série de funções para a leitura e análise sintática dos arquivos de diretivas de teste. Em particular disponibiliza a função `LER_LerParametros` similar à `scanf`, mas que permite a leitura de dados simbólicos (variáveis declaradas) e/ou literais, sendo que estes possuem uma sintaxe específica para cada tipo.

O *módulo de controle genérico*, módulo `GENERICO`, coordena a realização dos testes, interpreta os comandos de teste genéricos e disponibiliza uma série de funções de comparação dos resultados esperado e obtido. Ela é responsável por estabelecer uma interface uniforme com o testador. A descrição da interface está disponível no módulo de definição `GENERICO.H`.

A *função de controle específica*, contida no módulo `TESTXXX`, interpreta os comandos de teste que têm por objetivo exercitar o módulo a testar `XXX`. Na realidade esta função seria uma das funções do módulo de controle genérico, mas foi fatorada e incorporada em outro módulo para maior encapsulamento e conforto no desenvolvimento do módulo de controle específico.

Ao desenvolver módulos utilizando o arcabouço de teste automatizado, o programador precisa, obviamente, redigir o módulo a ser testado e o módulo de teste específico que contém a função de controle específica. As demais funcionalidades são disponibilizadas pelo arcabouço de teste. O módulo de definição (`.h`) de qualquer módulo de teste específico será sempre o módulo `TST_ESPC.H`. Isto garante a correta integração entre o módulo `GENERICO` e o módulo de teste especifico `TESTXXX`.

A combinação do módulo de controle genérico com o módulo de controle específico forma um *interpretador do arquivo de diretivas*. Desta forma a massa de teste passa a ser este arquivo de diretivas – o arquivo *script* de teste –, ao invés de estar contida em um código, como acontece no que foi visto no capítulo anterior. Cabe salientar que o presente arcabouço de teste também permite criar funções de teste similares ao descrito no capítulo anterior. Isto decorre do fato que cada item do interpretador poder ser tão complexo quanto se queira. No entanto, ao fazer isto perde-se uma série das vantagens de se utilizar um interpretador de comandos de teste.

Para cada item da interface do módulo a testar redige-se um pequeno código de controle que estabelece a interface com este item. Este código estará contido na função específica de controle do teste. O código de controle de cada item de interface é, em última análise, um interpretador simples que lê de um arquivo de diretivas os comandos de teste, os dados, ou parâmetros, e os resultados esperados. Após lidos os dados, a interface a testar é exercitada e o resultado obtido é extraído e comparado com o resultado esperado. Caso sejam discrepantes, é emitido uma indicação de falha. As funções de comparação são padronizadas e fazem parte do módulo de controle genérico. A seguir ilustramos o código de controle do teste de uma função disponibilizada na interface do módulo `ARVORE`.

```
/* Testar ARV Adicionar filho à esquerda */

    else if ( strcmp( ComandoTeste , INS_ESQ_CMD ) == 0 )
    {
       NumLidos = LER_LerParametros( "ci" ,
                         &ValorDado , &CondRetEsperada ) ;
       if ( NumLidos != 2 )
       {
          return TST_CondRetParm ;
       } /* if */

       return TST_CompararInt( CondRetEsperada ,
                 ARV_InserirEsquerda( ValorDado ) ,
                 "Retorno errado ao inserir à esquerda." );

    } /* fim ativa: Testar ARV Adicionar filho à esquerda */
```

7.2 Interpretador do arquivo de diretivas de teste

O arquivo de diretivas de teste é, na realidade, um programa de teste, redigido em uma linguagem de programação *ad hoc* especialmente projetada e desenvolvida para testar o módulo a testar. A sintaxe desta linguagem é similar para todas as implementações do módulo de teste específico. A sintaxe lembra fortemente uma linguagem do tipo *assembler* em que os parâmetros possuem tipo. A escolha de uma linguagem com esta sintaxe tem por finalidade reduzir os custos de desenvolvimento do interpretador bem como do *script* de teste.

A seguir ilustramos parte de um arquivo de diretivas para o teste utilizando todos os comandos com valores literais. O arquivo completo é o arquivo `TesteArvore.script`:

```
== Verificar ObterValorCorr relativo a uma árvore inexistente
=obter  '!'   4         retorno 4 -> não existe árvore

== Criar árvore
=criar      0   0
=irdir      5           retorno 5 -> árvore está vazia

== Inserir à direita
=insdir     'a'   0     retorno 0 -> operação bem sucedida

== Obter o valor inserido
=obter      'a'   0
```

A seguir ilustramos o mesmo *script* usando dados simbólicos. O arquivo completo é o arquivo `TesteArvoreSimb.script`.

```
== Declarar as condições de retorno
=declararparm  OK           int 0
=declararparm  NaoArvore    int 4
=declararparm  NaoCorr      int 5
```

```
== Declarar os conteúdos dos nós
=declararparm  CharErro     char  '!'
=declararparm  CharA        char  'a'

== Verificar ObterValorCorr relativo a árvore inexistente
=obter      CharErro   NaoArvore

== Criar árvore
=criar      OK
=irdir      NaoCorr

== Inserir à direita
=insdir     CharA   OK

== Obter o valor inserido
=obter      CharA   OK
```

Recomenda-se a leitura dos scripts de teste `TesteBasicoSimb.script` e `TesteBasico.script` que ilustram casos de teste em que se verifica se os ponteiros foram corretamente criados ao inserir nós na árvore. Isto é conseguido explorando-se as adjacências do nó inserido. O script `TesteArvoreSimb.script` verifica somente se a operação concluiu corretamente. Evidentemente ter concluído corretamente não é um indicador de que todas as atribuições também foram realizadas corretamente. Torna-se então necessário verificar se todos os campos do "`struct`" criado estão com um valor correto.

Com alguns cuidados de projeto do interpretador, a mesma massa de testes pode ser utilizada tanto para testar o módulo instrumentado, como para testar o módulo compilado para produção (otimizado). Isto assegura que o construto de produção possa ser verificado antes de ser aprovado, o que é recomendável, uma vez que a remoção de instrumentação[25] pode introduzir mal funcionamento.

O código a seguir ilustra parte do módulo de teste específico `TESTARV` utilizado para testar o módulo árvore do nosso exemplo.

```
#include    "TST_ESPC.H"
#include    "generico.h"
#include    "lerparm.h"
#include    "arvore.h"

/* Tabela dos nomes dos comandos de teste específicos */

#define     CRIAR_ARV_CMD       "=criar"
#define     INS_DIR_CMD         "=insdir"
#define     INS_ESQ_CMD         "=insesq"
#define     IR_PAI_CMD          "=irpai"
#define     IR_ESQ_CMD          "=iresq"
#define     IR_DIR_CMD          "=irdir"
#define     OBTER_VAL_CMD       "=obter"
#define     DESTROI_CMD         "=destruir"

/***************************************************************************
*
*  $FC Função: TARV Efetuar operações de teste específicas para árvore
*
*  $EP Parâmetros
*     $P ComandoTeste - String contendo o comando
*
*  $FV Valor retornado
*     Ver TST_tpCondRet definido em TST_ESPC.H
*
***************************************************************************/
```

[25] O código de instrumentação deve estar envolto em um controle de compilação condicional, por exemplo `#ifdef _DEBUG … #endif`. Ver capítulo 14 de [Staa 2000].

```
TST_tpCondRet TST_EfetuarComando( char * ComandoTeste )
{

   ARV_tpCondRet CondRetEsperada = ARV_CondRetMemoria ;

   char ValorEsperado = '?'  ;
   char ValorObtido   = '!'  ;
   char ValorDado     = '\0' ;

   int  NumLidos = -1 ;

   TST_tpCondRet Ret ;

   /* Testar ARV Criar árvore */

      if ( strcmp( ComandoTeste , CRIAR_ARV_CMD ) == 0 )
      {

         NumLidos = LER_LerParametros( "i" ,
                           &CondRetEsperada ) ;
         if ( NumLidos != 1 )
         {
            return TST_CondRetParm ;
         } /* if */

         return TST_CompararInt( CondRetEsperada , ARV_CriarArvore( ) ,
                                 "Retorno errado ao criar árvore." );
      } /* fim ativa: Testar ARV Criar árvore */

   /* Testar ARV Adicionar filho à direita */

      else if ( strcmp( ComandoTeste , INS_DIR_CMD ) == 0 )
      {

         NumLidos = LER_LerParametros( "ci" ,
                           &ValorDado , &CondRetEsperada ) ;
         if ( NumLidos != 2 )
         {
            return TST_CondRetParm ;
         } /* if */

         return TST_CompararInt( CondRetEsperada ,
                                 ARV_InserirDireita( ValorDado ) ,
                                 "Retorno errado inserir à direita." );
      } /* fim ativa: Testar ARV Adicionar filho à direita */

      else if ...
```

Cada fragmento do interpretador (bloco "`if`") deve obedecer à seguinte organização:

- seleção do comando
- leitura dos parâmetros, através da função `LER_LerParametros( ...);`
- verificação da corretude dos parâmetros, número de parâmetros lidos e valor lido;
- realização da ação a ser testada;
- confronto do resultado obtido ao realizar a ação com o esperado.

Cada bloco `if` corresponde ao interpretador de um determinado comando de teste. A análise do comando corrente é realizada pela função `LER_LerParametros`. O primeiro parâmetro é um *string* que informa o número e o tipo de cada parâmetro do comando de teste sendo analisado. Os demais parâmetros indicam onde deve ser armazenado o valor. Estes parâmetros são passados por referência.

São interpretados os tipos:

i – inteiro. Um literal inteiro é um número

f – flutuante (`double`). Um literal flutuante é um número vírgula flutuante em conformidade com o formato "f" da função `scanf`.

c – caractere. Um literal; caractere é uma letra contida entre aspas simples. Podem ser fornecidos caracteres especiais usando o caractere de escape '\'. Ex.: `'x','\n','\\'`

s – *string*. Um literal *string* é uma seqüência de caracteres contida entre aspas duplos. Podem ser utilizados caracteres especiais. Ex.: `"abc"`, `"a\"b\n"`

A função LER_LerParametros pode ler literais, conforme definidos acima ou nomes declarados. Ao declarar um nome, define-se o nome, o seu tipo e o seu valor, este através de um literal com tipo correspondente à declaração. Uma vez declarado um nome pode-se utilizá-lo em qualquer lugar em que seja solicitado um valor do tipo deste nome. Exemplos:

```
=declararparm  OK          int     0
=declararparm  NaoCorr     int     5
=declararparm  CharA       char    'a'
=declararparm  Letras      string  "abcdefghijklmnopqrstuvwxyz"
=declararparm  Tolerancia  double  1.E-7
```

7.3 Linguagem de diretivas

Nesta seção descrevemos a estrutura geral da linguagem de diretivas. Também serão descritas as diretivas genéricas.

O arquivo de diretivas de teste corresponde à massa de testes. O arquivo é formado por uma seqüência de casos de teste. Cada caso de teste contém um ou mais comandos de teste. Cada comando pode:

- exercitar um dos itens de interface do módulo a testar;
- executar alguma função do próprio módulo de controle específico e que tem por objetivo estabelecer ou modificar algum contexto necessário para realizar os testes;
- efetuar alguma operação genérica.

Cada comando ocupa uma linha de texto do arquivo. O padrão usado determina que todos os comandos de teste comecem na primeira coluna da linha e comecem com o caractere *igual* (`"="`).

Os comandos genéricos são:

- `linha em branco`
  Faz nada
- `// <Comentário>`
  Os comentários são exibidos no texto de saída (laudo).
- `== <comentário de início de caso de teste>`
  Inicia um case de teste. Cada case de teste deve testar um único aspecto e verificar se este aspecto foi corretamente realizado. Para tal pode ser formado por diversos

comandos de teste. Os comentários de início de caso de teste são exibidos no texto de saída. Estes comentários devem refletir exatamente o objetivo do caso de teste. Cada início de caso de teste incrementa o contador de casos de teste. Caso seja encontrada uma falha em um caso de teste, todas as diretivas a seguir serão ignoradas, até que se chegue a um novo início de caso de teste, quando a interpretação das diretivas é retomada. Sempre que ocorrer uma falha, a diretiva a seguir será examinada, ver `=recuperar` a seguir. Para evitar que acidentalmente se tente interpretar um início de caso de teste como uma diretiva `=recuperar`, recomenda-se preceder cada início de caso de teste por uma linha em branco. Isto também contribui para uma melhoria da legibilidade.

- `=recuperar`
  Em alguns casos deseja-se realizar um teste em que o código de interpretação da diretiva retornará uma identificação de falha. Em geral isto é feito com o intuito de verificar se o interpretador foi corretamente implementado. Mas como o resultado esperado é agora uma falha, se nada for feito o arcabouço acusará incorretamente que o teste resultou em algum problema. O comando `=recuperar` tem por finalidade sinalizar que se estava esperando uma indicação falha e, conseqüentemente, desconsiderar o problema reportado. No entanto, caso não seja reportada uma falha no comando de teste antecessor, o comando `=recuperar` reportará falha, indicando desta forma que o código de teste não operou como esperado.

- `=AceitaFalhasAcumuladas` <*número de falhas a aceitar*>
  Em alguns casos podem ser criados casos de testes que propositadamente geram diversas falhas neste caso o comando `=recuperar` não serve, pois somente recupera uma única falha. São exemplos de tais situações controles de cobertura de testes em que diversos contadores não foram ainda exercitados. Este assunto será visto mais adiante neste documento.

- `=bkpt`
  Freqüentemente é necessário utilizar um depurador para que se possa determinar a causa de um problema. O comando `=bkpt` permite a introdução de um *breakpoint* no *script* de teste a partir do qual o depurador poderá ser utilizado. Isto permite a execução acelerada do teste até um ponto pouco antes do que reporta o problema crítico. Para estabelecer o *breakpoint* que ativará o depurador é necessário marcar o código que interpreta o comando `=bkpt`.

- =<*comando de teste*> <*parâmetro*$_1$> <*parâmetro*$_2$> ...
  São os <comandos de teste> desenvolvidos pelo programador do módulo de teste específico. Cada parâmetro pode requerer zero ou mais parâmetros. Todos eles devem ser colocados na mesma linha que o *comando de teste*.

A linguagem utilizada é muito simples. É claro que se poderia utilizar linguagens mais complexas tal como ocorre em muitas ferramentas de teste [FG 1999], ou até mesmo XML para assegurar portatilidade. O efeito disso será somente uma complexidade maior do interpretador sem ganhos expressivos em sua funcionalidade. No caso de XML, se por um lado ganha-se em flexibilidade e portatilidade, por outro gasta-se muito mais esforço para redigir os *scripts* de teste. Como o teste de módulo é dirigido para testar um determinado módulo em determinada plataforma, é remota a chance de reúso dos *scripts* para uma grande variedade de módulos ou plataformas.

Os símbolos dos comandos disponibilizados pelo módulo de teste específico são definidos em uma tabela contida no módulo de controle específico `TEST_XXX.C` que contém a função de controle específico. Isto permite ao usuário selecionar a terminologia mais apropriada para as diretivas de teste de um determinado módulo a ser testado, além de documentar quais os comandos disponíveis. No nosso caso:

```
#define     CRIAR_ARV_CMD       "=criar"
#define     INS_DIR_CMD         "=insdir"
#define     INS_ESQ_CMD         "=insesq"
#define     IR_PAI_CMD          "=irpai"
#define     IR_ESQ_CMD          "=iresq"
#define     IR_DIR_CMD          "=irdir"
#define     OBTER_VAL_CMD       "=obter"
#define     DESTROI_CMD         "=destruir"
#define     ESPAC_CMD           "=espaco"
```

7.4 Vantagens e desvantagens

O teste automatizado utilizando uma linguagem de diretivas oferece algumas vantagens.

- O esforço de redação da função de teste específica é pequeno, sendo virtualmente igual ao esforço de redação de um controlador de teste manual.
- Os interpretadores de cada diretiva são simples, permitindo que sejam facilmente inspecionados quanto ao correto uso da interface.
- Através da linguagem de diretivas de teste pode-se realizar testes tão complexos e detalhados quanto se queira.
- Através da geração de um arquivo de *log* pode-se documentar os laudos dos testes realizados, assegurando a existência das histórias de evolução de cada construto.
- Caso os módulos não tenham dependência temporal (ex. não são *multi-threading*) os problemas encontrados são sempre repetíveis.
- Permite estabelecer com precisão (ex. linha de comando) onde o *debugger* deve ser ativado, reduzindo em muito o tempo gasto usando o *debugger*.

O teste automatizado utilizando uma linguagem de diretivas oferece também algumas desvantagens.

- O arquivo de diretivas é na realidade um programa. Ao encontrar um problema é necessário determinar se é decorrente de uma falta na programação do teste, uma falta no interpretador de diretivas, ou uma falta no módulo em teste.
- A linguagem *ad hoc* utilizada não permite a redação de subprogramas. Entretanto, subprogramas podem ser codificados no módulo de teste específico e disparados por um comando.

8 Arquitetura completa do arcabouço

Ao desenvolver um programa de forma incremental é conveniente criar uma estrutura de código que simplifique a evolução dos construtos. Para tal pode-se utilizar instrumentação

padronizada que é desenvolvida cedo no processo de desenvolvimento. É recomendado também utilizar um arcabouço[26] (*framework*) padrão de teste.

A Figura 5 ilustra a arquitetura abstrata do arcabouço de teste que pode ser utilizado ao desenvolver incrementalmente programas. Nessa figura a parte cinza corresponde ao conjunto de módulos já aceitos, bem como aos módulos que perfazem o arcabouço de teste. A parte fora da área cinza corresponde aos módulos a serem testados e o módulo de teste específico que interpretará as diretivas de teste especificamente desenvolvidas para testar os módulos a testar. O conjunto perfaz o construto de teste. O construto é compilado utilizando-se um *scripts* de `MAKE` especificamente projetado para ele. Este *script* de `MAKE` deve ser devidamente arquivado de modo que se possa, a qualquer momento, reconstruir o construto, permitindo assim o seu teste de regressão. Ao utilizar um ambiente integrado de desenvolvimento, cada construto corresponde a um projeto.

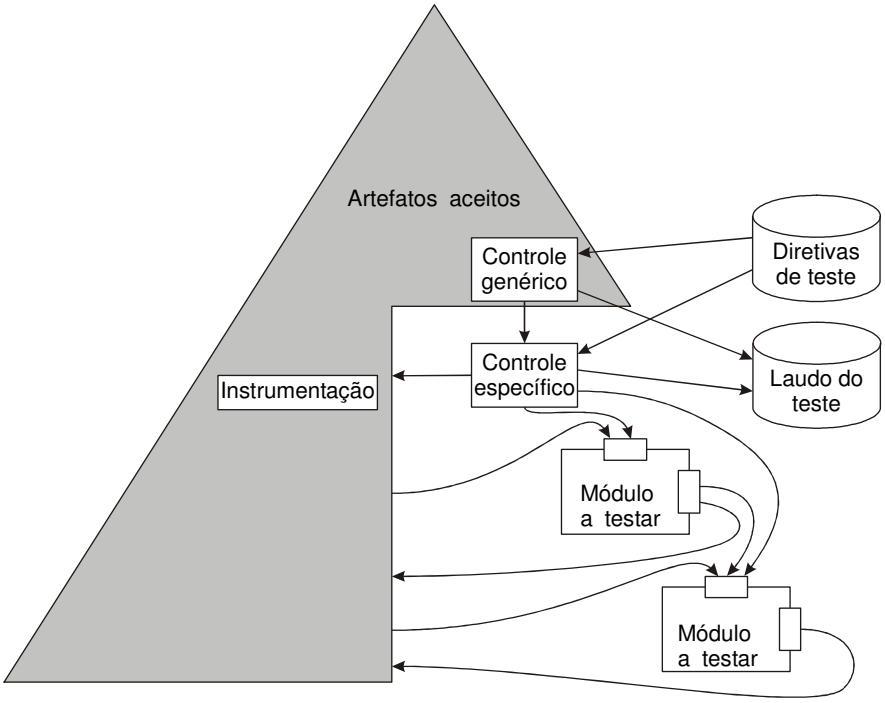

Figura 5. Arcabouço de teste.

À medida que os módulos e componentes forem sendo desenvolvidos e aprovados, eles são registrados como artefatos aceitos. Entre os artefatos aceitos encontra-se também a instrumentação a ser utilizada para realizar ou monitorar os testes [Staa 2000]. O módulo de controle genérico realiza as tarefas de teste comuns a todos os módulos ou artefatos a testar. O controle específico contém as funções que exercitam especificamente os módulos a testar.

Para testar determinado módulo será necessário compor um construto específico para este módulo. Este construto conterá o arcabouço, o módulo de teste específico, o módulo a ser testado, os módulos já aceitos requeridos pelo módulo a ser testado e, possivelmente módulos de enchimento (*stubs*, módulos ainda não desenvolvidos) requeridos pelo módulo a ser testado. Uma vez aprovados o módulo a testar, ele migra para o conjunto de artefatos

[26] Um ***arcabouço*** (***framework***) é uma solução incompleta para um problema genérico. Para resolver um problema específico o arcabouço deverá ser adequadamente instanciado.

aceitos e inicia-se o desenvolvimento de um novo módulo construto. Em algumas situações pode ser interessante desenvolver e testar mais de um módulo. Isto pode ocorrer caso sejam fortemente interdependentes.

O presente arcabouço não impõe uma estratégia de desenvolvimento específica. Ou seja, pode-se desenvolver de forma descendente (*top down*), ascendente (*bottom up*) ou qualquer outra, sem que isto implique uma mudança de organização do arcabouço. A única mudança observada é a forma usada para ativar o módulo de controle genérico. Pode-se, por exemplo, ativá-lo diretamente a partir de um programa principal genérico incorporado ao arcabouço, como é o presente caso, ou pode-se incluir um controle de ativação na barra de menu, ou, finalmente, pode-se associar um atalho (*hot key*) à sua chamada. Outra particularidade do arcabouço é ele permitir, tanto aos módulos em teste, quanto aos módulos de controle, a utilização dos módulos que já se encontram aprovados. Isto reduz em muito o trabalho de desenvolvimento das armaduras de teste[27] específicas para cada construto.

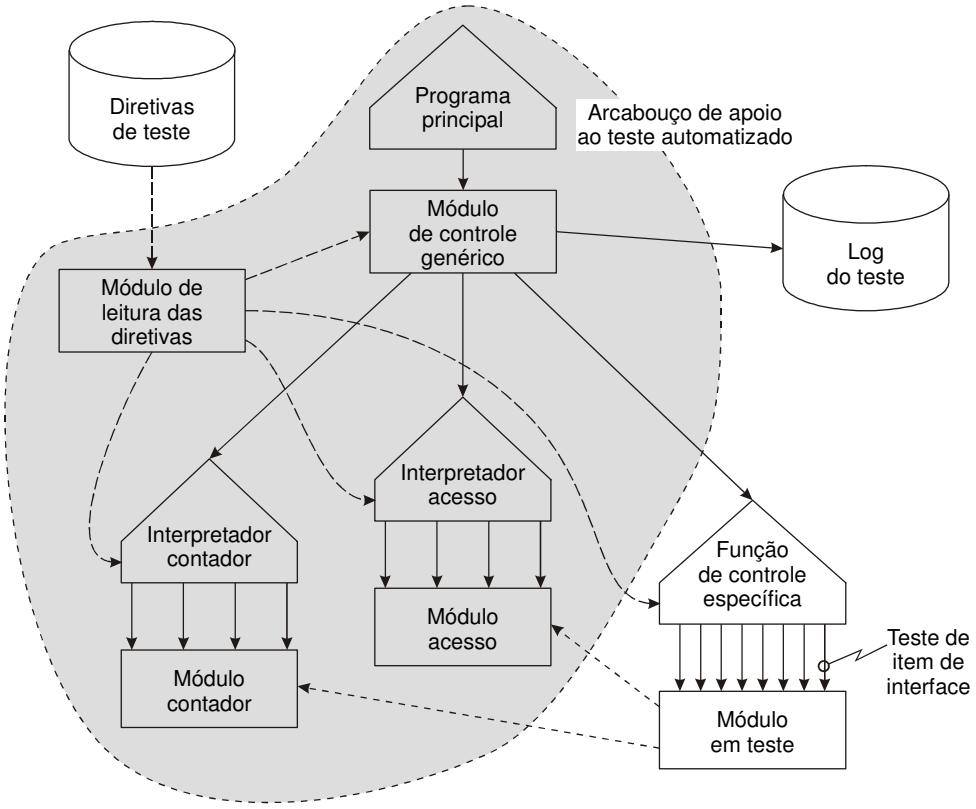

Figura 6. Arquitetura do arcabouço de apoio ao teste automatizado

Na Figura 6 é ilustrada a arquitetura completa do arcabouço. A parte sombreada corresponde ao arcabouço. O correspondente código encontra-se na biblioteca `ARCABOUCOTESTE.LIB`. Os retângulos correspondem a módulos ou instrumentos. Os retângulos com o lado superior quebrado correspondem a módulos interpretadores de comandos de teste. O módulo de leitura lê e analisa diretivas de teste. As linhas com

[27] Uma ***armadura de teste*** é formada por módulos de apoio ao teste. São exemplos: módulos de controle (*drivers*), módulos de enchimento (*stubs*), geradores de dados, verificadores de resultados, verificadores de estruturas, exibidores de estruturas (*data structure display functions*).

tracejado longo mostram a transferência de elementos das diretivas para os módulos interpretadores.

O módulo de controle genérico realiza as tarefas de teste comuns a todos os módulos ou artefatos a testar. Por exemplo, todas as funções de comparação são padronizadas e fazem parte do módulo de controle genérico. Além disso esse módulo interpreta os comandos de teste genéricos. Finalmente, cabe a ele distribuir os comandos para os diversos módulos interpretadores. A distribuição é realizada pela função encapsulada `Interpretar_Comandos`. Alterando-se essa função, podem ser adicionados novos instrumentos e correspondentes interpretadores de comandos de teste. Entretanto, esta função não deve ser alterada caso seja meramente adicionado um módulo de teste específico, uma vez que este já está previsto na sua implementação.

O controle específico contém as funções que exercitam especificamente o módulo a testar. Será necessário redigir um módulo de controle específico para cada construto. Para cada item de interface do módulo a testar redige-se um pequeno código de controle tal como descrito no capítulo Teste de módulos utilizando arquivos de diretivas que estabelece a interface entre os comandos de teste e os correspondentes itens a testar. Após ler e verificar os dados o item da interface a testar é exercitada e o resultado obtido é comparado com o resultado esperado. Caso sejam discrepantes, é emitida uma indicação de falha.

O módulo a testar pode conter chamadas a funções dos módulos do arcabouço, representadas como linhas com tracejado curto. Deve-se tomar cuidado, no entanto, de assegurar que estas chamadas não façam parte do módulo quando compilado para produção. Em outras palavras, todas as chamadas a funções de módulos do arcabouço devem estar contidas em um controle de compilação condicional (`#ifdef` ... `#endif`).

9 Controle de completeza dos testes

Um dos problemas comuns ao testar software é determinar quando os módulos foram suficientemente testados. Infelizmente, é freqüente que os testes sejam dados por concluídos quando se esgotou o prazo para a entrega do artefato, ou quando se exauriram os recursos para o seu desenvolvimento [Marick 1997, BB 2001]. É claro que critérios de término como estes tendem a levar a software de baixa qualidade. Portanto, são necessários critérios de término mais objetivos e que sirvam como indicadores *mensuráveis* da qualidade dos testes realizados.

Uma das formas de responder essa pergunta é definir critérios para a seleção de casos de teste de tal forma que testem completamente o módulo segundo algum critério de completeza. Na realidade faz parte de um bom critério de seleção a existência de um critério de completeza [GG 1975, GY 1976; Staa 2000]. Infelizmente o objetivo de completeza freqüentemente deixa de ser alcançado, ou por erro na formulação da massa de teste, ou por esse não ser o objetivo explícito do critério de seleção de casos de teste utilizado. Temos então que encontrar mecanismos que monitorem a execução dos testes de modo que se possa determinar com acurácia[28] a porção do módulo que foi efetivamente testada.

Métodos de programação baseados em redigir testes antes de iniciar a codificação [Beck 2000, Cockburn 2002] não eliminam a necessidade da realização de testes completos e

[28] ***Acurácia***: Propriedade de uma medida de uma grandeza física que foi obtida por instrumentos e processos isentos de erros sistemáticos. [Aurélio Eletrônico; Positivo Informática]

rigorosos. Os métodos ágeis têm a virtude de se basearem em especificações mecanicamente verificáveis, impedindo, assim, uma quantidade significativa de erros de programação, o que, por sua vez, conduz a código com menos faltas. Evidentemente, as que sobram precisam ser encontradas, o que requer massas de teste completas, confiáveis e válidas [GG 1975, GY 1976, Staa 2000]. Outra conseqüência dessa forma de conduzir o desenvolvimento é os testes de um módulo requererem duas abordagens. Na primeira, o teste de unidade, o módulo deve ser exercitado o mais completamente possível. Na segunda, o teste funcional, a interface do módulo deve ser exercitada de modo a demonstrar que o módulo se comporta tal como especificado atendendo os desejos do usuário. Evidentemente, além dessas podem ser desenvolvidas ainda mais massas de teste, cada qual visando objetivos de teste específicos, como por exemplo, testes de desempenho, testes de segurança, etc.

No restante desta seção discutiremos como medir automaticamente a cobertura dos testes. Estes instrumentos medem a porção do artefato que foi exercitada ao testar e, dentro de algumas limitações, podem informar também se determinadas formas de execução dos testes foram efetivamente realizadas.

9.1 Controle da cobertura

Controladores de cobertura têm a finalidade de obter estatísticas relativas à execução do módulo. Em linhas gerais, o controle de cobertura é realizado através da contagem do número de vezes que a execução passa por determinado ponto no programa. O resultado da contagem é um perfil da execução do programa. Existem várias formas de fazer isto [TH 2002]. Nós utilizaremos uma forma simples, mas pouco eficiente em consume de tempo de processamento.

Cada ***ponto de passagem*** a ser controlado é associado a um ***contador*** identificado por um nome simbólico. A contagem é realizada através da chamada da ***função contadora*** que tem como argumento o *nome* do contador. Cada vez que essa função for chamada o correspondente contador é incrementado de um. Os nomes dos contadores devem facilitar o entendimento do seu significado e a sua localização no código fonte. Para isso basta assegurar que cada o nome seja utilizado em um único local. Desta forma uma pesquisa pelo nome imediatamente haverá de localizá-lo.

O código a ser controlado é ***marcado*** com diversas chamadas da função de contagem, cada chamada utilizando um contador específico e que nenhuma outra chamada referencie. Desta forma cada contador estará associado a exatamente um ponto no código. A marcação pode ser realizada de forma manual ou pode ser realizada por alguma programa que leve em conta a estratégia de contagem a ser utilizada. Neste artigo assumimos a que a marcação seja feita de forma manual. A seguir ilustramos um fragmento de código marcado para contagem utilizando a macro `CNT_CONTAR( contador )`[29]:

```
void ARV_DestruirArvore( void ** refArvore )
{
   tpNoArvore * pArvore = ( tpNoArvore * ) ( *refArvore ) ;
   #ifdef _DEBUG
      CNT_CONTAR( "ARV_DestruirArvore, início" ) ;
   #endif
   if ( pArvore != NULL )
   {
      #ifdef _DEBUG
```

[29] Esta macro chama a função `CNT_Count( contador, número-da-linha-fonte)` fornecendo automaticamente o número da linha do código fonte onde se encontra a chamada.

```
         CNT_CONTAR( "ARV_DestruirArvore existente" ) ;
      #endif
      if ( pArvore->pNoRaiz != NULL )
      {
         #ifdef _DEBUG
            CNT_CONTAR( "ARV_DestruirArvore não vazia" ) ;
         #endif
         DestruirArvore( pArvore->pNoRaiz ) ;
      #ifdef _DEBUG
         } else
         {
            CNT_CONTAR( "ARV_DestruirArvore vazia" ) ;
      #endif
      } /* if */
      free( pArvore ) ;
      *refArvore = NULL ;
   #ifdef _DEBUG
      } else
      {
         CNT_CONTAR( "ARV_DestruirArvore não existente" ) ;
   #endif
   } /* if */
} /* Fim função: ARV Destruir árvore */

static void DestruirArvore( tpNo * pNo )
{
   #ifdef _DEBUG
      CNT_CONTAR( "Static DestruirArvore, início" ) ;
   #endif
   if ( pNo == NULL )
   {
      #ifdef _DEBUG
         CNT_CONTAR( "Static DestruirArvore nó não existente" ) ;
      #endif
      return ;
   }
   #ifdef _DEBUG
      CNT_CONTAR( "Static DestruirArvore nó existente" ) ;
   #endif
   DestruirArvore( pNo->pEsq ) ;
   DestruirArvore( pNo->pDir ) ;
} /* Fim função static: Destruir árvore */
```

No exemplo acima os pontos de contagem estão contidos em um fragmento de compilação condicional controlado pelo nome `_DEBUG`. Isso permite que o código de contagem possa permanecer no código do programa mesmo quando esse for compilado para produção (nome `_DEBUG` não definido). Ao compilar visando a contagem, deve-se definir o nome `_DEBUG`, caso contrário todos os comandos de contagem serão ignorados ao compilar.

O exemplo mostra outra particularidade que é o nome de cada contador corresponder ao nome da correspondente pseudo-instrução [Staa 2000]. Dessa forma a marcação servirá não só para auxiliar na medição dos testes, mas também servirá de mecanismo de documentação do módulo.

A estratégia de marcação depende do objetivo da medição. Por exemplo, para verificar a cobertura de uma massa de testes criada com o critério de completeza *cobertura de arestas*[30] [Staa 2000] deve-se inserir um contador no início de cada bloco de código subordinado a uma estrutura de controle da execução (ex.: `if`, `else`, `while`, `for`, `catch`), conforme ilustrado no exemplo acima. No caso de não existir `else` ou `default`, estes controles devem ser adicionados e o correspondente código conterá somente uma contagem, veja a

[30] O critério ***cobertura de arestas*** corresponde a inserir um contador no início de cada aresta do *fluxograma* correspondente ao código da função sendo controlada.

condição *destruir árvore nula* no exemplo acima. Cabe salientar que ambos os `else` da primeira função não figuram no código de produção. Na segunda função não é necessário criar um `else` vazio, uma vez que a parte `then` termina com um `return`. No entanto, é necessário inserir o contador após o término do `if`.

Ao final da execução do programa, exibe-se o conteúdo dos contadores (comando `=exibircontadores`), ou, então, verifica-se se existe algum contador cujo valor seja zero (comando `=verificarcontagens`). Todos aqueles que contenham zero, correspondem a porções nunca exercitadas do código. Deve-se, então, criar novos casos de teste de modo que todos os contadores passem a conter um valor diferente de zero ao final dos testes.

Muitos outros critérios de completeza podem ser controlados com contadores, entre eles:

- *cobertura de funções*, neste caso insere-se um contador somente no início de cada função. A contagem resultante informa quantas vezes cada função foi ativada no decorrer da execução do programa. Esse critério é o utilizado no código exemplo distribuído. Evidentemente é um critério fraco, uma vez que não assegura que todo o código de cada função tenha sido executado. O módulo `ARVORE` contido no diretório `instrum\fontes` ilustra o uso do controle de cobertura de funções.
- *cobertura de retorno*, neste caso antecede-se um contador a cada comando `return` (e por extensão: os comandos `throw`, e `exit( i )`). A contagem resultante informa quantas vezes a correspondente saída foi executada.
- *cobertura de chamada*, neste caso antecede-se um contador a cada chamada de função.

Para poder verificar se um contador foi ou não incrementado, é necessário saber de sua existência antes de iniciar a execução do teste. Portanto, é necessário criar uma tabela contendo todos os nomes de contadores que poderão ser usados no módulo. Isto pode ser conseguido através da leitura de um arquivo contendo todos os contadores utilizados no programa (comandos `=inicializarcontadores` ou `=lercontadores`).

A seguir ilustramos o conteúdo do arquivo de nomes de contadores correspondente ao exemplo acima.

```
ARV_DestruirArvore, início
ARV_DestruirArvore existente
ARV_DestruirArvore não existente
ARV_DestruirArvore vazia
ARV_DestruirArvore não vazia
Static DestruirArvore, início
Static DestruirArvore nó não existente
Static DestruirArvore nó existente
```

Como já foi mencionado, o teste terá sido completo, segundo uma estratégia de contagem, se, no *conjunto de todas as massas de testes*, todos os contadores contiverem um valor diferente de zero. Dependendo do módulo a testar, pode ser necessário particionar a massa de testes em diversos subconjuntos de casos de teste. Quando isto acontece é necessário poder-se acumular as contagens considerando as diversas "sub-massas" de teste. Portanto, é necessário que o conjunto de contadores e respectivas contagens possa ser gravado em um arquivo e, posteriormente, lido para que se possa efetuar a acumulação (comandos `=inicializarcontadores` ou `=gravarcontadores`). O comando inicializar requer um parâmetro *string* com o nome do *arquivo de acumulação*. Se este *string* for igual a `"."` não será utilizado arquivo de acumulação. Caso, ao iniciar, o arquivo de acumulação não exista, não ocorrerá problema desde que sejam lidos os arquivos de nomes de contadores. Ao final da

execução, quando o módulo de contagem for terminado, será gravado o arquivo de acumulação.

Programas podem conter fragmentos que, no processamento normal, jamais deveriam ser executados. Por exemplo, em seleções múltiplas é recomendado que cada seleção seja realizada explicitamente. Conseqüentemente, o `else` (ou `default`) final capturará condições ilegais. Portanto, esses fragmentos jamais deveriam ser executados no conjunto de todos os casos de teste. Para conseguir isso, o contador correspondente pode ser inicializado para um valor especial, no caso –2 (ex.: "`nome contador\=-2`", veja o módulo `CONTA.H`). Sempre que a função de contagem encontrar um contador com esse valor será gerada uma mensagem de falha de execução.

Finalmente, pode ser conveniente manter no código um comando de contagem, apesar de ser irrelevante, segundo o critério usado, se o controle passa ou não por esse ponto. Da mesma forma como contadores ilegais, pode-se inicializar um contador de modo que indique ser opcional, no caso o valor de inicialização é –1 (ex.: "`nome contador\=-1`", veja o módulo `CONTA.H`).

9.2 Integração com teste automatizado

O módulo `COUNTA` contém as funções de manipulação de contadores. Já o módulo `INTRPCNT` contém o interpretador de comandos de teste especificamente voltados para contadores. Ambos os módulos devem ser incorporados a um construto que deverá realizar o controle da cobertura de testes. O módulo `TESTCNT` interpreta comandos de teste do módulo contador que não estão disponíveis no uso normal do arcabouço.

A separação do módulo interpretador `INTCNT` e do módulo de processamento `CONTA` tem por objetivo permitir o uso das funções do módulo `CONTA` no código do módulo de teste específico.

O exemplo a seguir mostra a organização típica de um *script* de teste envolvendo contagem.

```
== Iniciar contagem
=inicializarcontadores  "."
=lercontadores          TesteContador-arv
=iniciarcontagem

== Criar árvore
=criar      0   0
=irdir      5

== Inserir à direita
=insdir     a   0

// Mais casos de teste

== Terminar controlar contadores
=pararcontagem
=verificarcontagens    0

== Terminar contagem
=terminarcontadores
```

9.3 Casos especiais

Os módulos de controle de cobertura podem ser utilizados também para verificar a corretude de um caso de teste específico. Por exemplo, pode ser desejado que cada caso de

teste percorra um determinado caminho no código do módulo [Staa 2000]. Na prática, ao testar exclusivamente através da interface, consegue-se somente examinar o resultado da execução, sendo virtualmente impossível verificar se o caminho foi executado exatamente conforme desejado.

A verificação da execução de caminhos específicos pode ser realizada com o apoio de contadores. Da mesma forma como no controle da completeza de um teste, o código deve ser marcado com comandos de contagem. Esses comandos devem ser inseridos nos pontos de interesse dos casos de teste a controlar. Antes de cada caso de teste os contadores afetados (ou todos eles) devem ser zerados (comando `=zerartodoscontadores`). Após a execução do caso de teste em questão verifica-se se cada contador afetado pelo caso de teste contém a contagem esperada. O valor esperado pode ser calculado com base nos dados escolhidos para o caso de teste. A seguir apresentamos um exemplo de código (intercalação de seqüências ordenadas) devidamente marcado.

```
CNT_CONTAR( "Início" ) ;
while ( ( Arq_A.Buffer.chave < MAX )
     && ( Arq_B.Buffer.chave < MAX ))
{
   CNT_CONTAR( "Repete" ) ;
   if ( Arq_A.Buffer.chave == Arq_B.Buffer.chave )
   {
      CNT_CONTAR( "Igual" ) ;
      TransferirRegistro( &Arq_A , Arq_S ) ;
      TransferirRegistro( &Arq_B , Arq_S ) ;
   } else if ( Arq_A.Buffer.chave < Arq_B.Buffer.chave )
   {
      CNT_CONTAR( "chave Arq_A menor" ) ;
      TransferirRegistro( &Arq_A , Arq_S ) ;
   } else
   {
      CNT_CONTAR( "chave Arq_B menor" ) ;
      TransferirRegistro( &Arq_B , Arq_S ) ;
   }
}
```

No exemplo a seguir mostramos como controlar a execução de dois casos de teste voltados para o código acima. Esses casos são parte de uma massa criada segundo o critério cobertura de caminhos [Staa 2000].

```
== Teste intercalar arquivos vazios
=zerartodoscontadores
=intercala "Vazio" "Vazio" "Vazio" "Vazio"   // A  B  S  D

=contagemcontador   "Início"  1
=contagemcontador   "Repete"  0
=contagemcontador   "Igual"   0
=contagemcontador   "chave Arq_A menor"  0
=contagemcontador   "chave Arq_B menor"  0

== Teste intercalar A com dois B com 1, 1o. A == 1o. B
=resetallcounters
=intercala  "Regs-1-5"  "Regs-1"  "Regs-5"  "Regs-par-1"

=contagemcontador   "Início"  1
=contagemcontador   "Repete"  2
=contagemcontador   "Igual"   1
=contagemcontador   "chave Arq_A menor"  1
=contagemcontador   "chave Arq_B menor"  0
```

No exemplo acima o comando `=intercala`, interpretado pelo módulo de teste específico, recebe 4 arquivos, os dois de entrada, o de saída normal e o de saída contendo duplicatas. Os arquivos de saída gerados ao intercalar devem ser comparados com os arquivos

identificados no comando de teste. O nome de cada arquivo identifica as chaves dos registros que contém, simplificando a compreensão dos casos de teste.

9.4 Comentários finais

É claro que a marcação do código com contadores é um trabalho tedioso, além de poluir visualmente o código. Idealmente deveria estar disponível um pequeno programa que fosse capaz de gerar uma versão marcada do código fonte. Ao selecionar esta versão ao compilar, passa-se a controlar a completeza do teste. Ao selecionar a versão não marcada, testa-se a versão de produção.

É claro também que as chamadas da função `CNT_CONTAR` consomem tempo significativo. Conseqüentemente é fortemente desaconselhado utilizar um módulo marcado em produção. Existem alternativas mais eficientes mas que requerem alterações em um nível bem mais baixo do que o da linguagem C [TH 2002].

10 Controle do acessos a espaços de memória dinâmica

Nessa seção discutiremos um instrumento capaz de monitorar o acesso a espaços de dados dinâmicos. Programas em C utilizam espaços de dados alocados dinamicamente (`malloc`) e uma profusão de ponteiros manipuláveis diretamente pelo usuário. A possibilidade de utilizar ponteiros oferece um significativo ganho de flexibilidade. Infelizmente, porém, ponteiros oferecem também sérios riscos de integridade a programas. Além disso, erros de uso de ponteiros são muitas vezes difíceis de diagnosticar devido à dificuldade de se estabelecer as relações de causa (uso incorreto de determinado ponteiro) e efeito (destruição de código ou das estruturas de dados devido ao uso incorreto de ponteiros).

Entre os diversos possíveis usos incorretos de ponteiros temos:

- acesso a um espaço de dados contendo dados de um tipo diferente do esperado pelo ponteiro.
- acesso a um espaço de dados que já foi liberado (`free`).
- acesso a dados fora dos limites do espaço alocado (extravasão, *buffer overflow*) .
- destruição de todos os ponteiros que apontam para um determinado espaço de dados, contudo sem liberar esse espaço (vazamento de memória).
- alteração do conteúdo de um espaço de dados a partir de um determinado ponteiro, quando outros ponteiros esperam que não seja alterado.

Através do módulo `CESPDIN` que controla o acesso a espaços dinamicamente alocados pode-se verificar a ocorrência de uma parte desses erros. Para utilizar este instrumento, basta incluir o módulo de definição `CESPDIN.H`. Esse módulo redefine as funções `malloc` e `free`, de modo que todos os espaços dinamicamente alocados estejam registrados em uma *lista de espaços alocados*. Cada espaço, além de conter vários controles, ver capítulo *Instrumentação* em [Staa 2000], também identifica o local do código fonte em que se encontra a correspondente chamada para a função `malloc`. Para evitar que os programas em produção mantenham essa lista, a inclusão de `CESPDIN.H` deve estar contida em um controle de compilação condicional `#ifdef _DEBUG`.

Uma boa implementação de uma estrutura de dados, mesmo a mais complexa, deve possuir operadores que a capacitem a liberar todos os espaços alocados. O comando `=obternumeroespacosalocados` permite verificar se o número de espaços ainda alocados corresponde ao esperado. Caso não corresponda é provável que tenha ocorrido vazamento de memória. O comando `=exibirtodosespacos` lista todos os espaços ainda alocados, indicando o arquivo fonte e a linha de código que continha o comando `malloc` utilizado para alocar o espaço. Com essa lista é facilitada a localização da falta que levou ao vazamento de memória.

Ao liberar um espaço (`free`), o gerente de memória dinâmica meramente ajusta alguns dados de controle, entretanto, os valores contidos no espaço desalocado não são afetados. Enquanto nenhuma alocação venha a utilizar parte ou todo esse espaço, o programa terá a impressão de que o espaço ainda está disponível e que contém dados corretos. Este tipo de problema tende a levar programas a terem um comportamento errático, ora funcionam ora não, sem uma explicação racional para isso. O instrumento de controle de memória dinâmica sempre borra os espaços liberados. Dessa forma qualquer uso de um espaço já liberado provocará funcionamento incorreto no programa, usualmente na forma de um cancelamento. Em geral esse tipo de falha é reportado por um erro de acesso a memória. Essa mesma característica do instrumento impede a múltipla liberação de um mesmo espaço de dados.

O módulo de controle de acesso a espaços de dados inclui controles de extravasão de uso dos espaços. Através desses controles é possível determinar se o módulo em teste gravou dados fora do domínio do espaço. O controle não é 100% preciso, uma vez que a função `sizeof` pode reportar um valor maior do que o efetivamente requerido. Utilizando o comando de pré-processamento `#pragma pack (1)` é possível ajustar as regras de alocação utilizados pelo compilador. Cuidado, `#pragma pack` introduz dependências de plataforma, possivelmente tornando não portáteis os diversos construtos contendo o módulo.

O instrumento de controle de acesso permite ainda controlar a igualdade do tipo do espaço e o tipo esperado pelo ponteiro. Para isso é necessário definir o tipo do espaço (função `CED_DefinirTipoEspaco`) e, depois, a cada vez que o espaço for acessado, comparar o tipo esperado pelo ponteiro que fará o acesso (função `CED_ObterTipoEspaco`). Cabe salientar que estas duas funções devem ser usadas somente quando o módulo for compilado para depuração (`_DEBUG` ligado).

Finalmente o módulo de controle de espaço dinâmico permite simular memória extremamente exígua. Desta forma podem ser forçadas situações nas quais o módulo reporta falta de memória. Pode simular a ocorrência de falta de memória com uma determinada freqüência.

O módulo `ARVORE` contido no diretório `instrum\fontes` ilustra o uso do controle de acesso a espaços dinâmicos. Para isso foram desenvolvidas funções de verificação da integridade estrutural de árvores, e também foi desenvolvida uma função de deturpação a ser utilizada para testar os verificadores. Ver capítulo *Instrumentação* em [Staa 2000] para mais detalhes com relação a essas classes de funções.

11 Processo de desenvolvimento

Nesta seção descrevemos, em linhas gerais, como desenvolver incrementalmente módulos utilizando o arcabouço de teste aqui apresentado. O objetivo é auxiliar o aprendiz a organizar o trabalho de desenvolvimento.

Em processos ágeis de desenvolvimento [Cockburn 2002] é proposto que até os módulos sejam desenvolvidos incrementalmente. Neste caso desenvolvem-se algumas funções do módulo que são testadas antes que se adicionem mais funções. Conseqüentemente as massas de teste crescem junto com o módulo. Ao final do desenvolvimento estarão completos e aprovados o módulo completamente implementado, a correspondente massa de teste, o módulo de controle específico de teste do módulo e as diretivas de reconstrução do construto (*script* do `make`), ver Figura 2.

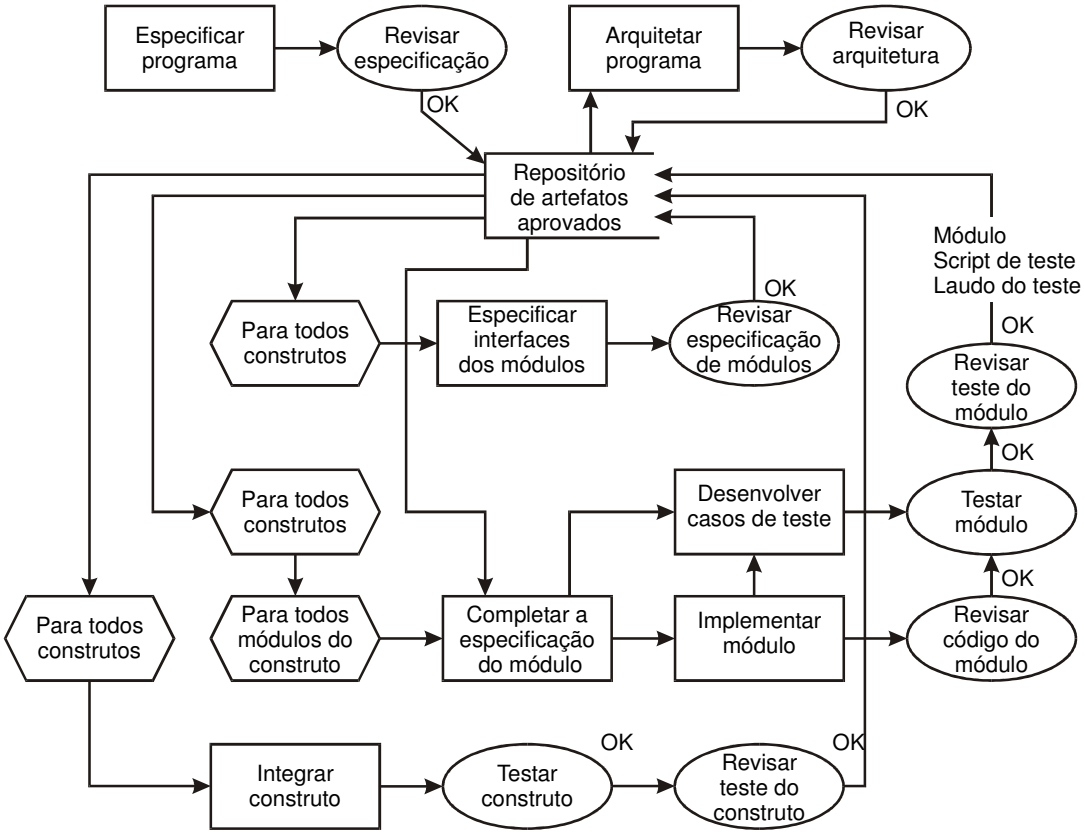

Figura 7 Processo de programação

A Figura 7 ilustra o processo de desenvolvimento de um programa composto por diversos módulos. A seguir detalharemos as tarefas *Implementar módulo*, *Desenvolver casos de teste*, *Testar módulo*. Siga o seguinte roteiro ao desenvolver programas utilizando esta abordagem de teste:

- Especifique a interface do módulo a ser testado (módulo de definição `.h`);
  - também pode ser feito incrementalmente, adicionando-se funções à medida que o módulo for sendo desenvolvido;
- Crie o módulo de implementação enchimento (`.c` *dummy*);
  - cada função faz nada;

- se a função deve retornar alguma coisa, retorna algum valor neutro, por exemplo: inteiro zero, ponteiro `null`, *string* nulo `""`;

- Redija a função de teste específico;
  - para cada função e atributo da interface deve existir um fragmento interpretando um comando de teste específico e que exercita esse item da interface. Inicialmente estes fragmentos podem fazer nada. Neste caso recomenda-se que retornem a condição `TST_CondRetNaoImplementado`.
  - pode ser necessário também desenvolver comandos de teste para estabelecer, verificar e destruir o contexto do teste. Projete o contexto e implemente o código necessário para manipular o contexto. O contexto de teste é usualmente formado por variáveis globais encapsuladas no módulo de teste específico. Por exemplo, o módulo de teste específico pode encapsular um vetor contendo ponteiros para cabeças de árvores definidas (veja o código no exemplo `\instrum`). Para manipular uma dessas árvores, o interpretador extrai o respectivo índice do comando de teste sendo interpretado. Este índice identificará a árvore a utilizar.

- Redija um *script* de teste contendo a lista de *todos* os casos de teste a serem realizados;
  - cada caso de teste contém somente o comando título seguido de uma linha em branco;
  - o *script* neste momento é somente uma lista de casos (roteiro de teste);
  - acrescente ao *script* os comandos dos "casos de teste" cuja finalidade é estabelecer, verificar ou destruir o contexto e o uso dos instrumentos. Na realidade, estes casos de teste nada testam, somente inicializam, criam ou destroem o contexto.

- Compile o construto;
  - corrija o programa até que não existam mais mensagens de erro, nem mensagens de advertência.

- Teste o construto
  - este teste tem por objetivo assegurar que toda a estrutura de compilação e de execução automatizada dos testes esteja operando corretamente.
  - este teste acusará falhas correspondentes aos casos de teste vazios.
  - corrija o *script*, o módulo de teste específico e o módulo de enchimento (módulo a ser testado) até que persistam somente falhas de caso de teste vazio.

- Codifique algumas (poucas) funções do módulo a ser testado
  - baseie-se em regras de precedência e de relevância para escolher quais as funções a implementar
    - funções que estabelecem um contexto devem ser implementadas antes das que utilizam este contexto.
    - funções que criam e extraem valores são mais relevantes do que as que manipulam ou alteram estes valores.
  - se contiver `malloc`, `free` ou ponteiros para espaços dinâmicos, instrumente o programa de modo que faça uso da instrumentação de controle de acesso a espaços dinâmicos.
  - instrumente o programa de modo que possa monitorar o teste segundo as estratégias de medição a serem usadas.

    - reveja – leia com atenção – o código procurando encontrar eventuais erros de projeto ou codificação. Esta forma de verificação de topo de mesa (*desk top checking*) é extremamente eficaz e gasta muito menos tempo do que a depuração do código usando um depurador!
- Redija os comandos dos casos de teste correspondentes às funções desenvolvidas.
- Teste e elimine todos os problemas encontrados. Lembre-se que o problema encontrado pode estar na especificação do módulo a testar, no código do módulo a testar, no módulo de teste específico, ou na massa de testes (arquivo de diretivas).
- Adicione casos de teste sempre que julgar que determinada condição ou seqüência de execução não tenha sido adequadamente testada.
- Repita até o que o módulo instrumentado esteja completamente implementado e testado.
- Produza uma massa de teste de produção.
    - Idealmente deve-se criar uma nova massa de teste independentemente da utilizada para o teste detalhado.
    - Freqüentemente será possível redigir a massa de teste de produção antes (ou durante) redigir o módulo a testar.
- Recompile o programa para produção. Destrua todos os módulos objeto (`.obj`) e recompile sem `_DEBUG` definido.
- Teste e elimine os problemas encontrados. Repita tudo, inclusive o teste do módulo instrumentado, até não encontrar mais problemas.
- Caso o programa passe tanto pelo teste instrumentado como pelo teste de produção, o módulo em teste poderá ser aceito.

12 Conclusão

O teste automatizado contribui para um aumento significativo da produtividade[31] e da qualidade dos módulos a serem desenvolvidos. Facilita muito a realização de testes de regressão. Conseqüentemente facilita, até mesmo auxilia, o desenvolvimento incremental de módulos e programas. Como a manutenção de um módulo pode ser vista como uma forma de incremento, o uso de teste automatizado contribui para uma manutenção mais rápida e correta. Mais especificamente, o teste automatizado oferece várias vantagens:

- exige rigor ao escrever as especificações das interfaces de um módulo ou componente.
    - embora alguns não acreditem, este rigor é sempre vantagem!
- facilita o desenvolvimento incremental do módulo.
- assegura que, a cada incremento, o módulo tenha sido completamente retestado.
- a função de teste específico serve como exemplo de uso do módulo.
- o *script* de teste serve como especificação operacional do módulo.

---

[31] ***Produtividade*** é medida em termos de uma dimensão (ex. número de ***linhas de código corretas***, ou número de pontos de função) implementada por unidade de tempo ou unidade de custo.

    - apesar de ser uma especificação incompleta e baseada em exemplos, freqüentemente é mais precisa do que especificações textuais.
- os problemas encontrados são repetíveis, facilitando a depuração.
- reduz significativamente o esforço de teste quando se leva em conta a necessidade de reteste após corrigir ou evoluir o módulo.
- reduz o estresse do desenvolvedor uma vez que passa a ser possível particionar o desenvolvimento em diversas atividades cada qual culminando com um módulo parcial porém corretamente implementado (segundo o teste).

Por outro lado o teste automatizado oferece algumas desvantagens:

- ao alterar um módulo obriga a evoluir
    - o módulo em teste.
    - a função de teste específico.
    - os casos de teste. Se os casos de teste forem mal documentados isto pode tornar-se um problema maior.
- ao encontrar um problema torna-se necessário determinar se a causa é:
    - o módulo em teste.
    - o módulo de controle genérico.
    - o arquivo de diretivas de teste.
- nem tudo é passível de teste automatizado, interfaces gráficas são exemplos.

Referências bibliográficas

[ALRL 2004] Avizienis, A.; Laprie, J-C.; Randell, B.; Landwehr, C.; "Basic Concepts and Taxonomy of Dependable and Secure Computing" ; IEEE Transactions on Dependable and Secure Computing 1(1); Los Alamitos, CA: IEEE Computer Society; 2004; pags 11-33

[BB 2001] Boehm, B.; Basili, V.R.; "Software Defect Reduction Top 10 List"; *IEEE Computer* 34(1); janeiro 2001; pags 135-137

[Beck 2000] Beck, K.; *Extreme Programming Explained*; New York: Addison Wesley; 2000

[Cockburn 2002] Cockburn, A.; *Agile Software Development*; Boston: Addison-Wesley; 2002

[CPPUnit] CPPUnit– arcabouço (*framework*) de apoio ao teste automatizado em C++. Procure a versão mais recente na rede. URL: http://sourceforge.net/ projects/cppunit

[DDH 1972] Dahl, O.-J.; Dijkstra, E.W.; Hoare, A.A.R.; *Structured Programming*; London: Academic Press; 1972

[Delamaro 1997] Delamaro, M.E.; *Mutação de Interface: Um Critério de Adequação Interprocedimental para o Teste de Integração*. Tese de Doutorado, IFSC/USP, São Carlos, SP, Junho, 1997.

[DMM 2001] Delamaro, M.E.; Maldonado, J.C.; Mathur, A.P.; "Interface Mutation: An Approach for Integration Testing"; *IEEE Transactions on Software Engineering* 27(3); Los Alamitos, CA: IEEE Computer Society; 2001; pags 228-247

[FG 1999] Fewster, M.; Graham, D.; *Software Test Automation*; New York: Addison-Wesley; 1999

[FO 2000] Fenton, N.E.; Ohlson, N.; "Quantitative Analysis of Faults and Failures in a Complex System"; *IEEE Transactions on Software Engineering* -26(8); 2000; pags 797-814

[Fowler 2000] Fowler, M.; *Refactoring: Improving the Design of Existing Code*; New York: Addison Wesley; 2000

[GG 1977] Goodenough, J.B.; Gerhart, S.L.; "Toward a Theory of Test Data Selection"; IEEE Transactions on Software Engineering 1(2); Los Alamitos, CA: IEEE Computer Society; 1977; pags 156-173

[GY 1976] Gerhart, S.L.; Yelowitz, L. "Observations of fallibility in applications of modern programming methodologies"; *IEEE Transactions on Software Engineering* 2(9); 1976; pp 195-207

[HT 2001] Hunt, A. ; Thomas, D.; *The Pragmatic Programmer*; Addison Wesley; New York; 2001

[HT 2003] Hunt, A.; Thomas, D.; "Mock Objects"; in: [HT 2003] Hunt, A.; Thomas, D.; eds.; *Pragmatic Unit Test in Java with JUnit*; Sebastopol, CA: O'Reilly; 2003

[HT 2003] Hunt, A.; Thomas, D.; *Pragmatic Unit Test in Java with JUnit*; Sebastopol, CA: O'Reilly; 2003

[JUnit] JUnit – arcabouço de apoio ao teste automatizado em Java. Procure a versão mais recente na rede. URL: http://junit.org/index.htm

[Kaner 2000] Kaner, C.; *Bug Advocay*; URL: http://http://www.testingcraft.com/bug-advocacy-kaner.pdf; baixado outubro 2003

[KFN 1988] Kaner, C.; Falk, J.; Nguyen, H.Q.; *Testing Computer Software*; Second Edition; New York: Thomson; 1988

[Laitenberger 2002] Laitenberger, O.; A Survey of Software Inspection Technologies; in Chang, S.K. ed.; *Handbook on Software Engineering*, vol. 2; 2002; pags 517-556

[Lehman 1998] Lehman, M.M.; "Software Future: Managing Evolution"; *IEEE Software* 15(1); IEEE Computer Society; 1998; pags. 40-44.

[Marick 1997] Marick, B.; *Classic Testing Mistakes*; STAR; 1997; Baixado 1/9/2003, URL: http://www.visibleworkings.com/papers/mistakes.pdf

[Nguyen 2001] Nguyen, H.Q.; *Testing Applications on the Web*; New York: Wiley; 2001

[OT 2001] Osher, H.; Tarr, P.; Using Multidimensional Separation of Concerns to (Re)shape Evolving Software; *Communications of the ACM* 44(10); pags 43-50

[Romanowsky 2005] Romanowsky, A.; *Software Dependability*; *Key Note Address*; 19o. Simpósio Brasileiro de Engenharia de Software (SBES'2005); Uberlândia, MG, Brasil; 2005

[SJ 2001] Splaine, S.; Jaskiel, S.P.; *The Web Testing Handbook*; Orange Park, Fa: STQE Publishing; 2001

[SRB 2000] Shull, F.; Rus, I.; Basili, V.R.; How Perspective-Based Reading Can Improve Requirements Inspections; *IEEE Software*; julho 2000; pags 73-79

[Staa 2000] Staa, A.v.; *Programação Modular*; Rio de Janeiro: Campus; 2000;

[TH 2002] Tikir, M.M.; Hollingsworth, J.K.; "Efficient Instrumentation for Code Coverage Testing"; Proceedings of the *2002 ACM SIGSOFT International Symposium on Software Testing and Analysis*, Rome, Italy, 2002; New York, NY: ACM Association for Computing Machinery; 2002; pags 86-96

[TH 2004] Thomas, D.; Hunt, A.; "Nurturing Requirements"; *IEEE Software* 21(2); Los Alamitos, CA: IEEE Computer Society; 2004; pags 13-15

[TOHS 1999] Tarr, P.; Ossher, H.; Harrison, W.; Sutton, S.M.; "N Degrees of Separation: Multi-Dimensional Separation of Concerns"; *Proceedings 21st International Conference on Software Engineering*; 1999; pags 107–119

[Wake 2001] Wake, W.C.; *Extreme Programming Explored*; Addison-Wesley; New York; 2001

[WV 2000] Whittaker, J.A.; Voas, J.; "Toward a More Reliable Theory of Reliability"; *IEEE Computer* 33(12); Los Alamitos, CA: IEEE Computer Society; 2000; pags 36-42