PUC

ISSN 0103-9741

Monografias em Ciência da Computação
n° 45/03

Extensão do Arcabouço para a Automação dos Testes de Programas Redigidos em C

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 45/03
Editor: Carlos J. P. Lucena Novembro, 2003

Extensão do Arcabouço para a Automação dos Testes de Programas Redigidos em C

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

Extensão do Arcabouço para a Automação dos Testes de Programas Redigidos em C

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

Abstract. A module test automation framework is presented. This framework is specifically geared towards modules written in C. This paper extends chapter *Instrumentação* in Staa, A.v.; *Programação Modular*; Campus 2000 [Staa 2000]; and also *Arcabouço para Automação de Testes de Programas Redigidos em C* [Staa 2003]. The extensions described here comprise a module designed to perform passage counting. This instrument may be used to control the quality of the test suite. Another extension involves the control of data spaces accessed by means of pointers. Using this instrument the programmer is able to verify the type correctness of dynamic data spaces and may also observe the occurrence of buffer overflows and memory leaks.

Keywords: C program testing framework; Dynamic data space access control; Incremental development; Module testing; Software Engineering; Test automation; Test coverage control.

Resumo. É apresentado um arcabouço (*framework*) simples para a automação de testes de módulos redigidos na linguagem C. O artigo estende o capítulo *Instrumentação* em Staa, A.v.; *Programação Modular*; Campus 2000; e também a monografia Staa, A.v.; *Arcabouço para Automação de Testes de Programas Redigidos em C*; MCC 09/03; Departamento de Informática, PUC-Rio; 2003. A primeira extensão introduzida aborda o controle da qualidade das massas de teste em si. Esse controle é realizado utilizando técnicas de contagem do número de passagens por determinado ponto no código. Outro instrumento disponibilizado auxilia o desenvolvedor a identificar falhas de uso de ponteiros. Através desse instrumento é possível verificar a corretude de tipos entre o tipo implícito no ponteiro e o tipo específico do espaço por ele apontado. Esse instrumento permite, ainda, verificar extravasamento de espaços alocados, bem como vazamento de memória.

Palavras-chave: Arcabouço de testes visando C; Automação do teste; Controle da cobertura dos testes; Controle do acesso a espaços de memória dinâmicos; Desenvolvimento incremental; Engenharia de software; Teste de módulos.

Sumário

1 Introdução

Este documento tem por objetivo discutir e ilustrar instrumentos de automação dos testes de módulos redigidos em C. Embora a instrumentação seja especificamente voltada para testes de programas redigidos em C, os conceitos aqui apresentados podem ser facilmente transferidos para programas C++ e, muitos deles também para Java. Na realidade os instrumentos aqui ilustrados nasceram como instrumentos C++ e que, posteriormente, foram convertidos para C.

O arcabouço aqui descrito amplia o descrito na monografia *Arcabouço para Automação de Testes de Programas Redigidos em C* [Staa 2003] incorporada ao pacote que acompanha esse documento, além de complementar os capítulos de *Instrumentação*, e *Teste de Módulos* contidos no livro *Programação Modular* [Staa 2000]. Assume-se que o leitor tenha lido estes dois documentos.

O arquivo de distribuição `ArcaboucoTeste-C-2003.zip` contém os diversos arquivos que implementam e ilustram o uso do arcabouço (*framework*) de teste para programas escritos em C apresentado nesse documento. Contém ainda o código fonte, ferramentas de apoio, documentação e exemplos de diretivas de teste. Para não ser inutilmente extenso e repetitivo, os detalhes do uso do arcabouço encontram-se nos comentários dos módulos 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-C-LeiaMe-2003.pdf`.

No presente documento serão descritas a arquitetura do arcabouço e os instrumentos adicionais que podem ser utilizados para controlar os testes de programas. Os instrumentos disponibilizados são: controle da completeza dos testes e controle de acesso a espaços de memória dinâmica. Entretanto, o presente documento não apresenta todos os detalhes do arcabouço. Esses podem ser encontrados nos arquivos de código fonte que constituem o arquivo de distribuição.

1.1 Motivação

Testar artefatos é uma das várias técnicas de controle da qualidade de programas Ao testar um artefato este é submetido a várias massas de teste cada qual composta por vários casos de teste. Cada caso de teste é composto por um conjunto de dados e comandos e pela descrição dos resultados esperados. Ao executar um caso de teste, os resultados obtidos são comparados com os respectivos resultados esperados. Caso a comparação mostre a existência de uma discrepância, terá sido identificada uma falha ou um defeito[1].

Numa forma de automatizar testes, a massa de teste passa a ser um programa que pode ser escrito na linguagem do próprio módulo a testar [JUnit, CPPUnit]. Em outra forma a massa de teste é redigida em uma linguagem de diretivas (*script*) [Staa 2003, FG 1999]. A massa de teste conterá agora diretivas para exercitar o módulo em teste e para comparar os

[1] ***Falhas*** são comportamentos do programa diferentes do esperado e que comprometem 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.

resultados desses exercícios. A massa de teste pode conter também diretivas que monitorem o próprio processo de teste. Essas diretivas são processadas pela instrumentação.

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. A cada incremento adicionam-se ao conjunto de artefatos já aceitos alguns poucos novos, idealmente um único, módulos ou componentes. O construto resultante deve ser cuidadosamente testado, verificando-se todas as propriedades adicionadas com relação ao construto anterior já testado. Uma vez aprovados os artefatos adicionados, estes são incorporados ao conjunto de artefatos aceitos e progride-se para o desenvolvimento do próximo módulo.

A remoção das faltas contidas em um artefato, bem como a evolução deste, conduzem a repetidas alterações nesse artefato. Muitas delas são estruturais e não somente o acerto de algumas linhas de código, tornando necessária uma reorganização de todo o código (*refactoring*) [Beck 2000, Fowler 2000]. Após cada alteração, independentemente de quão simples ou complexa, deve-se realizar um ***teste de regressão*** com o intuito de verificar se os problemas foram integralmente resolvidos e nenhum novo foi introduzido. Portanto, testes de regressão são realizados um número grande de vezes.

Uma vez encontrada uma falha deve-se diagnosticá-la, localizando *todos* os pontos no código que contribuam para ela, somente depois disso o código deverá ser corrigido. A diagnose[2] dos problemas 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 fatorados, compondo módulos a serem incorporados ao arcabouço de teste automatizado.

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 que exercita essa funcionalidade. Essa arquitetura viabiliza tanto o uso de *scripts* de teste como o de funções de teste. Além disso, reduz o trabalho de instrumentar os módulos a testar.

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 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 em-

[2] ***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.

prego 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.

1.2 Organização

Na seção 2 apresentaremos um breve resumo dos conceitos de teste necessários para o entendimento de instrumentação e automação dos testes. Na seção 3 descrevemos a arquitetura do arcabouço. Primeiro é mostrada a inter-relação entre os módulos em teste e o conjunto já aceito. Depois é discutida a organização do arcabouço. Na seção 4 discutimos o instrumento de apoio ao controle da completeza dos testes. Um dos problemas comuns ao testar software é determinar quando os módulos foram suficientemente testados. São necessários, então, critérios de término objetivos e que sirvam como um indicador mensurável da qualidade dos testes realizados. Na seção 5 descrevemos um instrumento que visa controlar o acesso a espaços de memória dinâmica. Programas em C utilizam espaços de dados alocados dinamicamente e uma profusão de ponteiros manipuláveis diretamente pelo usuário. Ponteiros oferecem sérios riscos de integridade a programas. Além disso, erros de uso de ponteiros podem ser muito difíceis de diagnosticar devido à dificuldade de se estabelecer as relações de causa (uso incorreto de determinado ponteiro) e efeito (destruição de código ou estruturas de dados devido ao uso incorreto de ponteiros). Na seção 6 descrevemos, sucintamente, um processo de desenvolvimento incremental de módulos utilizando o arcabouço de teste aqui apresentado. O objetivo é auxiliar o aprendiz a organizar o trabalho de desenvolvimento. Finalmente, a seção 7 apresenta um breve fecho do artigo.

2 Teste de módulos, resumo

Nesta seção apresentaremos, de forma bastante resumida, os conceitos de teste necessários para o entendimento de instrumentação e automação dos testes. Uma descrição mais detalhada pode ser encontrada em [Staa 2000] e [Staa 2003].

Para assegurar a corretude de um programa composto por módulos é necessário, entre outras coisas, o teste rigoroso de cada módulo [FO 2000]. Integrar programas compostos por módulos de qualidade duvidosa obviamente aumenta o risco do programa vir a falhar, possivelmente gerando danos de grande envergadura. Além disso, a diagnose a partir de um relatório de problema encontrado pelo usuário de um programa composto por vários módulos tende a ser muito custosa e desgastante para o fornecedor desse 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 de uma especificação incorreta. Em alguns casos também é interessante verificar as conseqüências do uso incorreto do módulo, com vistas a determinar se o módulo é suficientemente robusto[3]. Em todos esses casos cria-se um módulo de controle do teste (*driver*) cuja finalidade é possibilitar o exercício do módulo a testar. Para assegurar que o

[3] Um programa (módulo) ***robusto*** percebe 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.

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 do teste, o módulo testado não deveria mostrar mudanças de comportamento.

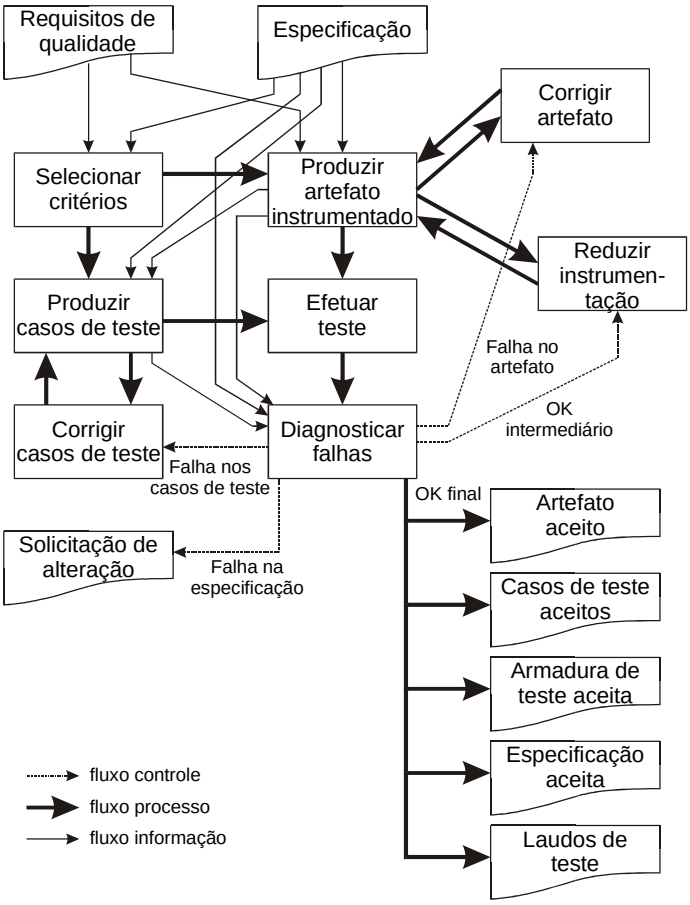

Figura 1. Visão macroscópica do processo de teste de artefatos.

Quando for desejado testar em detalhe as funções internas ao módulo, pode tornar-se necessário criar uma interface adicional especificamente projetada para fins de teste. A interface específica para teste [Staa 2003] permite interagir com a instrumentação [Staa 2000] cuja finalidade é monitorar a execução, a completeza dos testes, o correto uso de espaços de dados dinâmicos[4] e a confiabilidade da instrumentação. 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 1. 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 (`#ifdef`, `#else`, `#endif`).

Conseqüentemente, o teste deverá ser realizado em pelo menos duas etapas, uma em que a instrumentação deverá estar presente e, a outra, sem utilizá-la. Isto é necessário para verificar se a remoção da instrumentação introduziu ou não problemas no módulo em teste.

[4] ***Espaços de dados dinâmicos*** são áreas de memória real alocadas com funções do tipo `malloc`.

Conseqüentemente, são necessários dois construtos 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.

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. Infelizmente isto nem sempre será verdade.

Tampouco basta realizar vários testes para adquirir confiança no módulo, é necessário também verificar se são suficientes para servirem como um indicador confiável da qualidade dos módulos testados. 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]

A confiabilidade de um teste depende da qualidade da massa de teste utilizada. Uma massa de teste pouco rigorosa 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[5].

Conforme ilustrado na Figura 1, 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 pode levar à descoberta de problemas antes desconhecidos.
- 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, pode levar a ter-se que modificar componentes do arcabouço de teste.
- 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.
- a especificação aceita[6]. 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 executável em que cada caso de teste constitui um exemplo da funcionalidade esperada.

---

[5] 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.

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

- o conjunto de laudos produzidos ao testar, sendo que o mais recente deles estabelece a ausência de faltas não toleradas.[7]

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, eles são muito sensíveis à escolha. Ou seja, diferentes escolhas podem levar à identificação de diferentes faltas, ou mesmo não acusar a existência de faltas contidas no código. Em virtude disso, segundo Dijkstra [DDH 1972], testes podem evidenciar a existência de faltas, mas jamais a ausência delas. Esta deficiência dos testes é tão maior quanto menos criteriosa tiver sido a escolha dos casos de teste.

3 Arquitetura do arcabouço

Nessa seção descrevemos a arquitetura do arcabouço. Primeiro é mostrada a inter-relação entre os módulos em teste e o conjunto já aceito. Depois é discutida a organização do arcabouço.

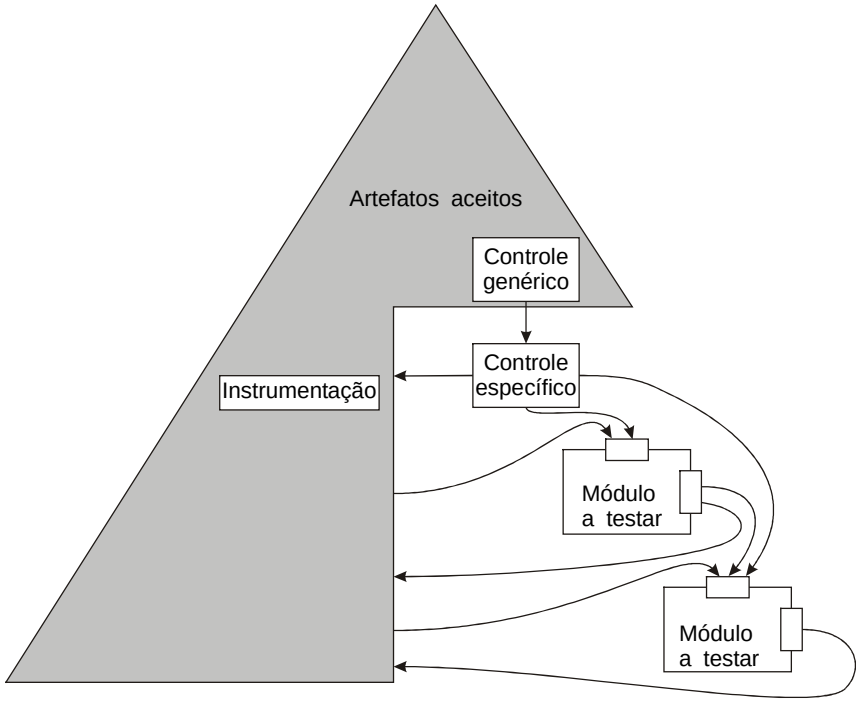

Figura 2. Arcabouço de teste.

Ao desenvolver um programa de forma incremental é conveniente criar uma estrutura de código que simplifique a evolução dos construtos. Para tal podem ser utilizados instrumentos padronizados que fazem parte do arcabouço de teste automatizado. A Figura 2 ilustra a arquitetura abstrata utilizada ao desenvolver incrementalmente programas [Staa 2003]. Nessa figura a parte cinza corresponde ao conjunto de módulos já aceitos, bem como aos

[7] Em algumas situações faltas conhecidas podem permanecer no código, 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 a uma posterior nova liberação do módulo.

módulos que perfazem o arcabouço de teste. A parte fora da área cinza corresponde aos módulos a serem testados e o módulo de controle específico que interpretará as diretivas de teste especificamente desenvolvidas para testar esses módulos. O conjunto perfaz o construto de teste. Através de *scripts* de `MAKE` os diversos construtos podem ser reconstruídos a qualquer momento, permitindo assim o teste de regressão de qualquer um deles.

Essa arquitetura não exige 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 o ponto em que se ativa o módulo de controle genérico. Pode-se, por exemplo, ativá-lo diretamente a partir de um programa principal genérico, ou pode-se incluir um controle de ativação na barra de menu da aplicação, ou, finalmente, pode-se associar um atalho (*hot key*) à sua chamada. Outra particularidade dessa arquitetura é ela 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[8] específicas para cada construto.

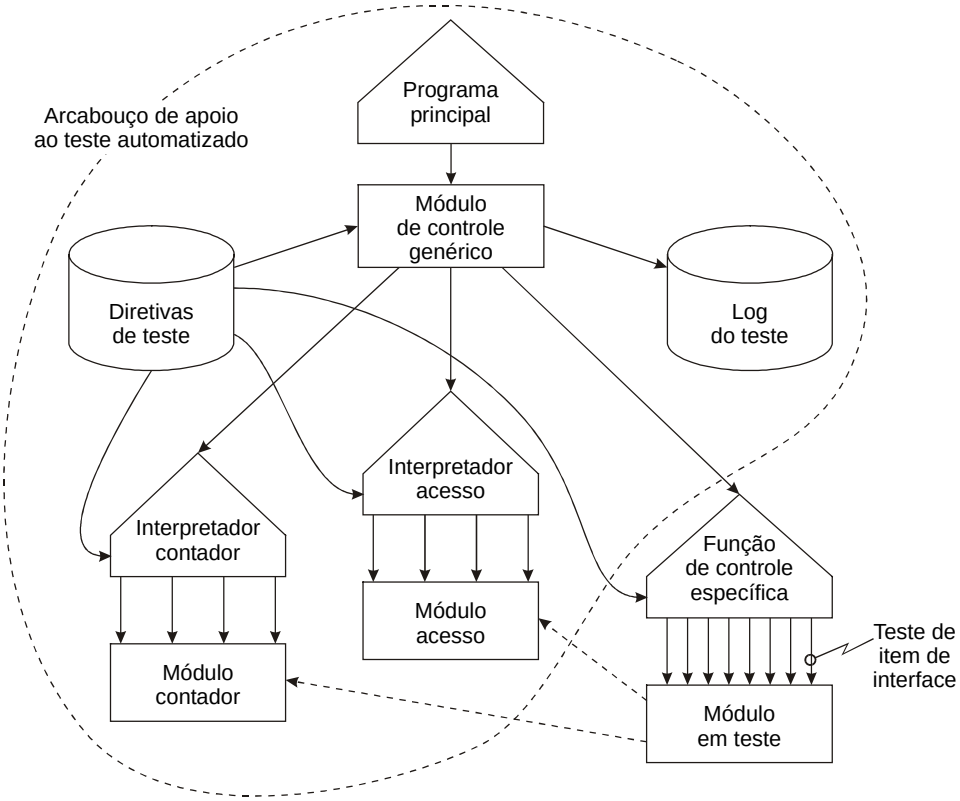

Figura 3. Arquitetura do arcabouço de apoio ao teste automatizado

Na Figura 3 é ilustrada a arquitetura completa do arcabouço. Os retângulos correspondem a módulos ou instrumentos. Os retângulos com o lado superior quebrado correspondem a módulos interpretadores de comandos de teste.

---

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

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 `InterpretarComandos`. Alterando-se essa função, podem ser adicionados novos instrumentos e correspondentes interpretadores de comandos de teste.

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 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. A seguir é apresentado um exemplo de interpretador de um determinado comando de teste específico.

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

   else if ( strcmp( ComandoTeste , INS_ESQ_CMD ) == 0 )
   {
      NumLidos = sscanf( TST_Buffer , "%*s %d %c %i" ,
                         &inxArv , &ValorDado , &CondRetEsperada ) ;
      if ( ( NumLidos != 3 )
        || !EhInxValido( inxArv , TRUE ))
      {
         return TST_CondRetParm ;
      } /* if */
      CondRetObtido = ARV_InserirEsquerda( VetArv[ 1 ] , ValorDado ) ;
      return TST_CompararInt( CondRetEsperada , CondRetObtido ,
                  "Retorno errado ao inserir à esquerda." );
   } /* fim: Testar ARV Adicionar filho à esquerda */
```

4 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 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

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.

4.1 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.

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 pode-se incluir no nome do contador o número da linha de código em que é usado como argumento da função de contagem.

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_COUNT( contador )`[9]:

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

[9] 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.

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

static void DestruirArvore( tpNo * pNo )
{
   #ifdef _DEBUG
      CNT_COUNT( "Static DestruirArvore, início" ) ;
   #endif
   if ( pNo == NULL )
   {
      #ifdef _DEBUG
         CNT_COUNT( "Static DestruirArvore nó não existente" ) ;
      #endif
      return ;
   }
   #ifdef _DEBUG
      CNT_COUNT( "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*[10] [Staa 2000] deve-se inserir um contador no início de cada bloco subordinado a uma estrutura de controle da execução (ex.: `if`, `else`, `while`, `for`, `catch`), conforme ilustrado no exemplo

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

acima. No caso de não existir `else` ou `default`, estes controles devem ser adicionados e o correspondente código conterá somente uma contagem, veja a 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 `=verificarcontadores`). 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 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 exemplo. Evidentemente é um critério fraco, uma vez que não assegura que todo o código da função tenha sido executado.
- *cobertura de retorno*, neste caso antecede-se cada comando `return` (e por extensão: os comandos `throw`, e `exit( i )`) de um contador. A contagem resultante informa quantas vezes a correspondente saída foi executada.
- *cobertura de chamada*, neste caso antecede-se cada chamada de função de um contador.

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 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 todos 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 `=inicializar-contadores` ou `=gravarcontadores`).

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 (veja o módulo `COUNTERS`). Agora, sempre que a função de contagem encontrar um contador com esse valor é 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 (veja o módulo `COUNTERS`).

4.2 Integração com teste automatizado

O módulo `COUNTERS` contém as funções de manipulação de contadores. Já o módulo `INTCNT` 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.

A separação do módulo interpretador `INTCNT` e do módulo de processamento `COUNTERS` tem por objetivo permitir o uso direto do módulo de processamento no código cliente. Isso permite realizar testes utilizando funções de teste ao invés de ser forçado a usar sempre um interpretador de diretivas de teste.

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
=verificarcontadores    0

== Terminar contagem
=terminarcontadores
```

4.3 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 `=resetallcounters`). 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.

```
COUNT( "Início" ) ;
while ( ( Arq_A.Buffer.chave < MAX )
     && ( Arq_B.Buffer.chave < MAX ))
{
   COUNT( "Repete" ) ;
   if ( Arq_A.Buffer.chave == Arq_B.Buffer.chave )
   {
      COUNT( "Igual" ) ;
      TransferirRegistro( &Arq_A , Arq_S ) ;
      TransferirRegistro( &Arq_B , Arq_S ) ;
   } else if ( Arq_A.Buffer.chave < Arq_B.Buffer.chave )
   {
      COUNT( "chave Arq_A menor" ) ;
      TransferirRegistro( &Arq_A , Arq_S ) ;
   } else
   {
      COUNT( "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
=resetallcounters
=intercala Vazio Vazio Vazio Vazio   // A  B  S  D

=valorcontador   "Início"  1
=valorcontador   "Repete"  0
=valorcontador   "Igual"   0
=valorcontador   "chave Arq_A menor"  0
=valorcontador   "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

=valorcontador   "Início"  1
=valorcontador   "Repete"  2
=valorcontador   "Igual"   1
=valorcontador   "chave Arq_A menor"  1
=valorcontador   "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. O nome de cada arquivo identifica as chaves dos registros que contém, simplificando a compreensão dos casos de teste.

5 Controle do acessos a espaços de memória dinâmica

Nessa seção discutiremos um instrumento a ser utilizado para monitorar o acesso a espaços de dados dinâmicos. Programas em C utilizam espaços de dados alocados dinamicamente 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 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.
- 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 tenha sido alterado.

Através do módulo `CNTESPAC` 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 `CNTESPAC.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 `CNTESPAC.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 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 não são afetados. Enquanto nenhuma alocação venha a utilizar parte ou todo esse espaço, o programa tem a im-

pressão que o espaço ainda está disponível e que contém dados corretos. Este tipo de problema tende a levar a programas com 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 mal funcionamento no programa. Em geral essa falha é reportada 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` introduz dependências de plataforma, possivelmente tornando não portáteis os diversos construtos contendo o módulo.

O instrumento de controle de acesso permite também controlar a igualdade do tipo do espaço e o tipo esperado pelo ponteiro. Para isso é necessário definir o tipo do espaço (comando `=definirtipoespaco`) e, depois, a cada vez que o espaço for acessado, comparar o tipo esperado pelo ponteiro que fará o acesso (comando `=obtertipoespaco`). Cabe salientar que esses dois comandos usualmente estarão contidos no módulo a testar, portanto devem ser controlados por `#ifdef _DEBUG`.

O módulo ARVORE 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.

6 Processo de desenvolvimento

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

Em processos ágeis de desenvolvimento [Beck 2000, 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 adicione mais funções. As massas de teste crescem junto com o módulo. Ao final do desenvolvimento estarão completos e aceitos 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 (`make`) utilizado para o teste do módulo completo. O roteiro a seguir auxilia desenvolver programas utilizando esta abordagem:

- Especifique a interface do módulo a ser testado (módulo de definição)
  - também pode ser feito incrementalmente, adicionando-se funções à medida que o módulo for sendo desenvolvido.
- Crie o módulo de implementação enchimento (*dummy*)
  - o módulo implementa cada função da interface. As funções devem fazer nada (o corpo da função é vazio).

- se a função deve retornar alguma coisa, retorna algum valor neutro (por exemplo: 0, NULL etc.) e sempre o mesmo.

- 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.
  - 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 as cabeças das árvores definidas. Para manipular uma dessas árvores, o interpretador extrai o respectivo índice do comando de teste sendo interpretado.
- Redija um *script* de teste contendo a lista dos casos de teste a serem realizados
  - cada caso de teste contém somente o comando título
  - 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 programa (construto)
  - corrija o programa até que não existam mais mensagens de erro *nem* mensagens de advertência.
- Teste o programa
  - este teste tem por objetivo assegurar que toda a estrutura de compilação e de execução automatizada dos testes esteja operando corretamente.
  - este teste deve acusar somente falhas de caso de teste vazio.
  - 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. Por exemplo `CriarArvore` deve ser testado antes de usar essa árvore.
    - funções que criam valores são mais relevantes do que as que manipulam, extraem 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.
- Redija o código dos casos de teste correspondentes às funções desenvolvidas.

- se necessário redija os casos de teste complementares que verificarão o contexto, os contadores e os espaços alocados.

- Teste e elimine todos os problemas encontrados. Lembre-se que o problema encontrado pode estar na especificação do módulo em teste, no código do módulo em teste, no módulo de teste específico, ou na massa de testes (arquivo de diretivas).
- Adicione casos de teste sempre que observar ou julgar que determinada condição ou seqüência de execução não tenha sido adequadamente testada.
  - casos de teste devem ser adicionados caso algum contador reportar zero após a execução de todas as massas de teste.
  - casos de teste também devem ser adicionados caso sejam encontrados problemas difíceis de serem diagnosticados.
- 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 (sem `_DEBUG` definido)
- Teste e elimine os problemas encontrados. Repita tudo, inclusive o teste do módulo instrumentado, até não encontrar mais problemas.
  - a completeza da massa de teste de produção também pode ser medida utilizando contadores.
  - primeiro teste o módulo com a instrumentação ativa utilizando o *script* de produção.
  - depois recompile o módulo para produção e desative os comandos de teste que tenham interface com a instrumentação.
- Caso o programa passe tanto pelo teste instrumentado como pelo teste de produção, o módulo em teste poderá ser aceito.

7 Epílogo

O teste automatizado de módulos instrumentados viabiliza o controle da qualidade da massa de teste utilizada e, também, auxilia na diagnose dos problemas encontrados. Contribui, assim, para um aumento da produtividade e da qualidade dos módulos a serem desenvolvidos. Facilita muito a realização de testes de regressão. Conseqüentemente, facilita, até mesmo auxilia, no 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 também contribui para uma manutenção mais rápida e correta.

Referências bibliográficas

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

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

[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

[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

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

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

[GG 1975] Goodenough, J.B.; Gerhart, S.L.; "Toward a Theory of Test Data Selection"; *IEEE Transactions on Software Engineering* 1(2); 1975; 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

[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

[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

[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;

[Staa 2003] Staa, A.v.; *Arcabouço para Automação de Testes de Programas Redigidos em C*; Monografias em Ciência da Computação 09/03; Departamento de Informática, PUC-Rio; 2003.