PUC

ISSN 0103-9741

Monografias em Ciência da Computação
n° 09/03

Arcabouço para Automação de 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 XX/AA
Editor: Carlos J. P. Lucena Março, 2003

Arcabouço para Automação de 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

Arcabouço para Automação de Testes de Programas Redigidos em C

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. 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 can be instantiated to test a specific module. The instantiated test control module implements a test script interpreter. In this approach a test script specifically geared to sufficiently test the given module defines the test suite. The article also emphasizes incremental development of programs as well as of modules.

Keywords: C program testing framework; Incremental development; Module testing; Software Engineering; Test automation.

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 compreensão do texto. Descreve duas modalidades de automação de testes. A primeira baseada em um módulo utilizando funções de teste especificamente implementadas para testar um determinado módulo. Nesta abordagem a massa de teste é formada pelo código fonte do módulo de controle do teste. A segunda abordagem utiliza um arcabouço para a instanciação de um interpretador de diretivas de teste voltado para um determinado módulo a ser testado. Nesta segunda modalidade, a massa de teste é formada por um arquivo de diretivas contendo diversos casos de teste e a respectiva documentação. O artigo enfatiza o desenvolvimento incremental tanto de programas como de módulos.

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

1 Introdução

Este documento tem por objetivo discutir e ilustrar técnicas de automatização dos testes de módulos. Embora seja especificamente voltado para testes de programas redigidos em C, os conceitos aqui apresentados podem ser facilmente transferidos para programas C++ e Java. Na realidade as ferramentas aqui ilustradas nasceram como ferramentas C++ que foram convertidas para C. Este documento complementa os capítulos de *Instrumentação*, *Teste de Módulos* e *Integração* contidos no livro [Staa 2000].

Testar programas é uma das várias técnicas de controle da qualidade de programas Ao testar um artefato[1] 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 interface com o usuário que confunde ou dificulta o trabalho deste;
- um arquivo, ou base de dados, cujo conteúdo e estrutura não corresponde ao esperado.

Testes podem ser realizados com diferentes pontos de vista. Por exemplo, o ***teste de corretude***[4] procura encontrar diferenças entre o especificado e o implementado. O artefato estará correto caso esteja em conformidade exata com sua especificação. No ***teste de interface*** a preocupação é identificar se a interface do componente[5] ou módulo[6] 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[7] resolve os problemas que o usuário[8] espera ver resolvidos. No ***teste de utilizabili-***

---

1 ***Artefato*** é algum resultado tangível do desenvolvimento de um programa. São exemplos: documentos 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 para o usuário; arquivos contendo texto de auxílio (*help*); etc.

2 ***Massa de teste*** é o conjunto de todos os dados e comandos correspondentes aos ***casos de teste*** e que será utilizada em uma sessão ou execução de teste.

3 ***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.

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.

5 ***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*). Exemplos são arquivos `.jar`, `.dll`, `.so`, ou `.lib`.

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

7 ***Construto*** é um programa operacional e que implementa uma parte da funcionalidade que o programa almejado deverá vir a implementar. No ***desenvolvimento incremental***, cada construto incremental implementará mais funções do

dade 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[9] 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.

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 poucos módulos ou componentes, idealmente um, devidamente aprovados ao conjunto de elementos que forma o incremento antecessor. 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. 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].

A remoção das faltas[10] contidas em um artefato bem como a evolução deste, conduzem a repetidas alterações no artefato. Muitas delas 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[11] foram resolvidos e

---

programa final que o construto antecessor, culminando com o construto final correspondendo ao programa plenamente desenvolvido.

8 ***Usuário*** de um módulo é tomado no sentido lato. Um ***módulo cliente*** utiliza um outro módulo; o ***programador cliente*** reutiliza o módulo em algum programa; o ***programador desenvolvedor*** desenvolve ou mantém o módulo; a pessoa, ***usuário*** no *sentido estrito*, que utiliza o programa contendo o módulo.

9 ***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.

10 ***Erros*** são cometidos por desenvolvedores e/ou ferramentas. Erros introduzem faltas ou deficiências no artefato. Uma ***falta*** é uma inadequação latente que, se exercitada de uma determinada forma, conduz a uma falha. De forma similar, ***deficiências*** conduzem a defeitos.

11 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.

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 afetem 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 altera 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 a um custo baixo, assegurando a completeza necessária para que se possa continuar a confiar no 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.

Na seção 2 apresentaremos, de forma bastante resumida, os conceitos de teste necessários para o entendimento da automação dos testes. Na seção 3 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, explorar e destruir árvores binárias. Na seção 4 discutiremos como se processa o teste manual de módulos. O objetivo é identificar as ações passíveis de automação. Na seção 5 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. Na seção 6 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 dado arquivo de diretivas (*script*). Na seção 7 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. Na seção 8 concluiremos descrevendo as principais vantagens e desvantagens advindas do uso de teste automatizado. Na seção 9 descreveremos a composição do arquivo `ZIP` contendo o exemplo de uso do arcabouço e o ambiente de desenvolvimento utilizado.

2 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[12]) de problemas que o módulo em teste poderia apresentar.

Para avaliar a corretude de um programa composto por módulos é necessário, entre outras coisas, o teste rigoroso de cada módulo. 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[13]) a partir de relatórios de problemas encontrados em um programa composto por vários módulos tende a ser muito custosa e desgastante para o fornecedor do programa, uma vez que precisa identificar exatamente os módulos faltosos.

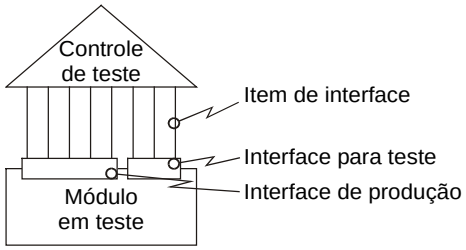

Figura 1. Interação entre o módulo de controle e do módulo em teste

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[14]. Em ambos os 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. Caso seja desejado testar em detalhe funções ou algoritmos internos ao módulo, torna-se necessário criar 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.

O teste deverá ser realizado em duas etapas, uma incluindo a instrumentação e a outra sem inclui-la. Isto é necessário para verificar se a remoção da instrumentação não introduz pro-

---

12 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 (teste exaustivo não existe!). No entanto, através de vários instrumentos, entre eles testes, podemos chegar bem perto. Uma parte significativa da indústria consegue produzir programas com 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.

13 ***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.

14 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.

blemas no módulo a testar. Conseqüentemente são necessários dois construtos e duas massas de teste.

São exemplos de itens de 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 pilha ser ou não encapsulada.

Ao testar um módulo deve-se verificar, para cada função de interface, se esta se comporta conforme especificado e para cada dado externado se contém valores conforme especificado. 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 desenvolver módulos ou funções capazes de examinar o conteúdo destes artefatos com o intuito de verificar se é o esperado. 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.

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[15] (*framework*) padrão de teste. A Figura 2 ilustra a arquitetura abstrata de um arcabouço de teste que pode ser utilizado ao desenvolver incrementalmente programas.

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. Desta forma será necessário redigir um módulo de controle específico para cada construto. Como veremos mais adiante, esta tarefa tem um custo modesto. Uma vez os módulos a testar tendo sido aprovados, eles migram para o conjunto de artefatos aceitos e inicia-se o desenvolvimento de um novo construto.

---

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

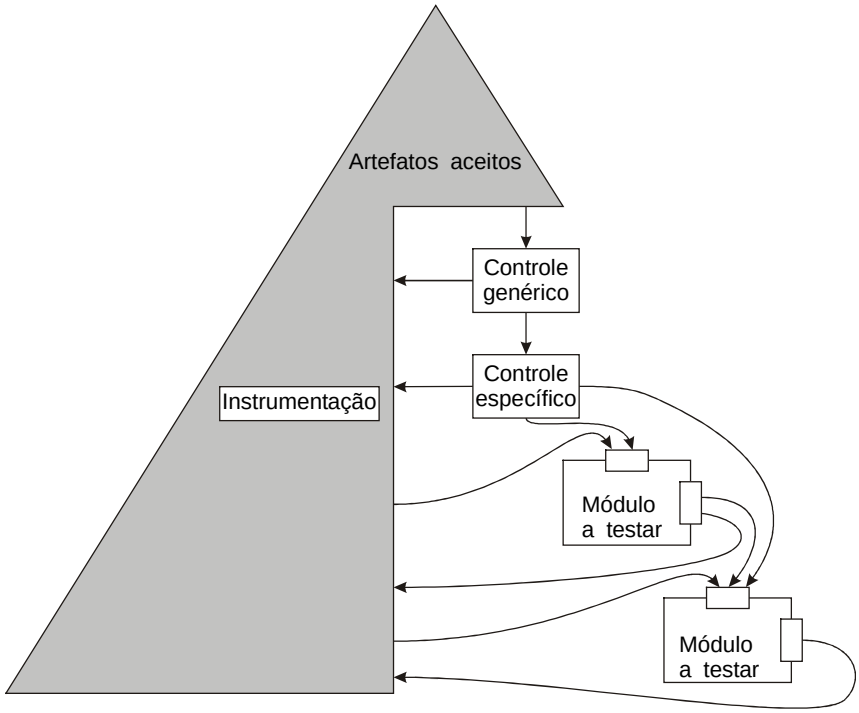

Figura 2. Arcabouço de teste.

O presente arcabouço 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, 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[16] específicas para cada construto.

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

[16] 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*).

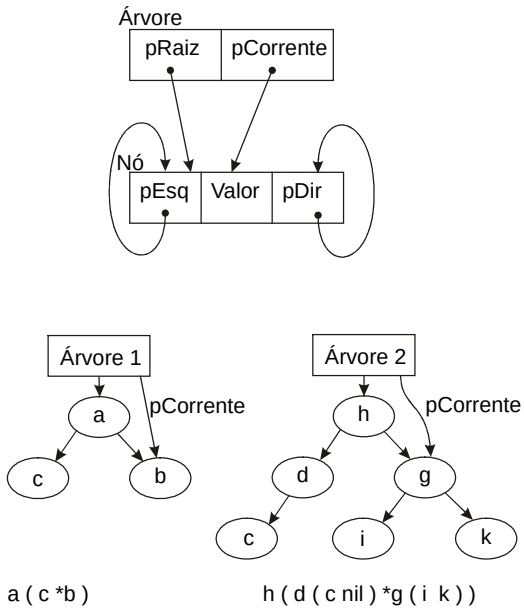

Figura 3. Modelo e exemplos de árvores usadas neste texto

O código a seguir corresponde ao módulo de definição[17] C (*header file*, módulo de declaração). A redação está em acordo com os padrões de programação estabelecidos em [Staa 2000]. Ou seja, o módulo de definição contém também a especificação 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
*
*  $HA Histórico de evolução:
*     Versão  Autor    Data     Observações
*       3.00   avs   28/02/2003 Uniformização da interface das funções e
*                               de todas as condições de retorno.
*       2.00   avs   03/08/2002 Eliminação de código duplicado, reestruturação
*       1.00   avs   15/08/2001 Início do desenvolvimento
*
*  $ED Descrição do módulo
*     Este módulo implementa um conjunto simples de fun‡”es para criar e
*     explorar árvores binárias.
*     A árvore possui uma cabeça que contém uma referência para a raíz da
*     árvore e outra para um nó corrente da árvore.
*     A cada momento o módulo admite no máximo uma única árvore.
*     Ao iniciar a execução do programa não existe árvore.
```

[17] Na linguagem C todos os módulos devem ser compostos por dois arquivos de código fonte. Um deles, o ***módulo de definição***, declara todos os tipos, constantes, dados e funções que estão na *interface* do módulo. Contém também a documentação desta interface. O segundo arquivo, o ***módulo de implementação***, contém as declarações dos tipos, constantes, dados e funções *encapsuladas*. Contém ainda o código de implementação de todas as funções, encapsuladas ou não. Finalmente contém a documentação encapsulada destinada ao desenvolvedor.

```
*     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.
*
*  $FV Valor retornado
*     ARV_CondRetOK
*     ARV_CondRetMemoria
*
***************************************************************************/

   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_CondRetOK
*     ARV_CondRetEstrutura
*     ARV_CondRetMemoria
*     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_CondRetOK
*     ARV_CondRetEstrutura
*     ARV_CondRetMemoria
*     ARV_CondRetFolha     - caso não seja folha para a direita
*
***************************************************************************/

   ARV_tpCondRet ARV_InserirDireita( char ValorParm ) ;


/***************************************************************************
*
*  $FC Função: ARV Ir para nó pai
*
*  $FV Valor retornado
*     ARV_CondRetOK
*     ARV_CondRetNaoArvore
*     ARV_CondRetNaoCorr
*     ARV_CondRetEhRaiz
*
***************************************************************************/

   ARV_tpCondRet ARV_IrPai( void ) ;


/***************************************************************************
*
*  $FC Função: ARV Ir para nó à esquerda
*
*  $FV Valor retornado
*     ARV_CondRetOK
*     ARV_CondRetNaoArvore
*     ARV_CondRetNaoCorr
```

```
*     ARV_CondRetNaoFilho   - nó corrente não possui filho à esquerda
*
*****************************************************************/

   ARV_tpCondRet ARV_IrNoEsquerda( void ) ;


/*****************************************************************
*
*  $FC Função: ARV Ir para nó à direita
*
*  $FV Valor retornado
*     ARV_CondRetOK
*     ARV_CondRetNaoArvore
*     ARV_CondRetNaoCorr
*     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.
*
*  $FV Valor retornado
*     ARV_CondRetOK
*     ARV_CondRetNaoArvore ;
*     ARV_CondRetNaoCorr ;
*
*****************************************************************/

   ARV_tpCondRet ARV_ObterValorCorr( char * ValorParm ) ;

#undef ARVORE_EXT

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

#else
#endif
```

4 Teste manual de módulos

Nesta seção discutiremos 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 de 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[18]. 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:

[18] ***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..

```
 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, é solicitado 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 a ação 8 com relação ao novo nó corrente. 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[19], é desenvolver uma interface gráfica (GUI – Windows) na qual se seleciona a ação, fornece os dados necessários e exibe os resultados obtidos.

O teste manual oferece várias vantagens:

- 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*[20], seleção de cores 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 que são aceitáveis.

Por outro lado possui grandes desvantagens:

---

[19] De maneira geral não vale à pena enfeitar os controladores de teste, uma vez que isto acarreta custos que não trazem benefícios.

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

- O confronto visual de resultado esperado com resultado obtido é altamente sujeito a erros humanos. Freqüentemente o testador *acha* que os resultados são iguais ou aproximadamente iguais, 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 realizados 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. Conseqüentemente, os casos de teste tendem a ser criados sem seguir um critério de seleção de casos de teste estabelecido. 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 falha cuja causa (falta) seja diferente das anteriormente identificadas.
- O teste de regressão tende a examinar somente as funções que foram modificadas, sem examinar se estas modificações introduzem anomalias em porções do programa aparentemente não afetadas pelas 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. Para que se possa ter confiança em um programa é necessário que a implementação corresponda *exatamente*[21] à especificação. 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 foi obedecida rigorosamente.

5 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 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 defi-

[21] Correspondência ***exata***: o programa implementa tudo o que está especificado e nada além do que está especificado.

nida 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 VerificarInt( 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__`

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 <contaLinha> 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, e o resultado obtido. Muitas vezes o problema encontrado não está no módulo a ser testado e sim no próprio *script* 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 a função de teste passa a ser:

```
VerificarInt( CriarArvore( ), ARV_CondRetOK , __LINE__ ,
              "Erro ao criar árvore.") ;
VerificarInt( InserirEsquerda('a' ), ARV_CondRetOK , __LINE__ ,
              "Erro ao inserir raiz.") ;
VerificarInt( IrPai( ), ARV_CondRetEhRaiz, __LINE__ ,
              "Erro ao caminhar para pai de raiz.") ;
VerificarChar( ObterValor( ) , 'a' , __LINE__ ,
              "Valor do nó 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 que foram observadas, bem como onde, no código de teste, estas falhas foram encontradas. Isto facilita ao desenvolvedor localizar as faltas, removê-las e refazer os testes com vistas a confirmar a completa remoção da falta. 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.

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.

Esta forma de testar apresenta várias vantagens:

- É possível gerar um código tão extenso quanto necessário para que se disponha de uma massa de teste confiável. Exceto em algumas situações 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 em teste. Isto reduz a dificuldade de se estabelecer interfaces confiáveis. Viabiliza, também, a redação de funções auxiliares, repetições, seleções e tudo o mais que uma linguagem de programação oferece.
- Ao utilizar linguagens e bibliotecas de desenvolvimento para janelas que permitam consultar o conteúdo dos *widgets*, pode-se até automatizar uma boa parte do teste da interface gráfica [FG 1999].

A técnica oferece também algumas desvantagens

- A função de teste específica tende a ser muito extensa e mal documentada, tornando difícil verificar a completeza e 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[22] [Staa 2000] a estrutura controlada pela instrumentação de modo que se possa verificar se esta é suficientemente eficaz em determinar falhas estruturais.
- Em virtude da extensão os módulos de teste podem se tornar difíceis de manter, uma vez que misturam o assunto[23] (*concerns*) [OT 2001, TOHS 1999] de criar e manter o contexto (cenário do teste), com o de realizar um suficiente número 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 em princípio têm acesso a todos os módulos e como são muito extensas, pode-se tornar difícil observar se existe uma ação implementada de forma perniciosa.

6 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

---

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

[23] 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.

diretivas de teste que se encontram em um dado arquivo de diretivas (*script*). O arquivo de distribuição `Teste-Automatizado.zip` contém os diversos arquivos que implementam o que está descrito aqui.

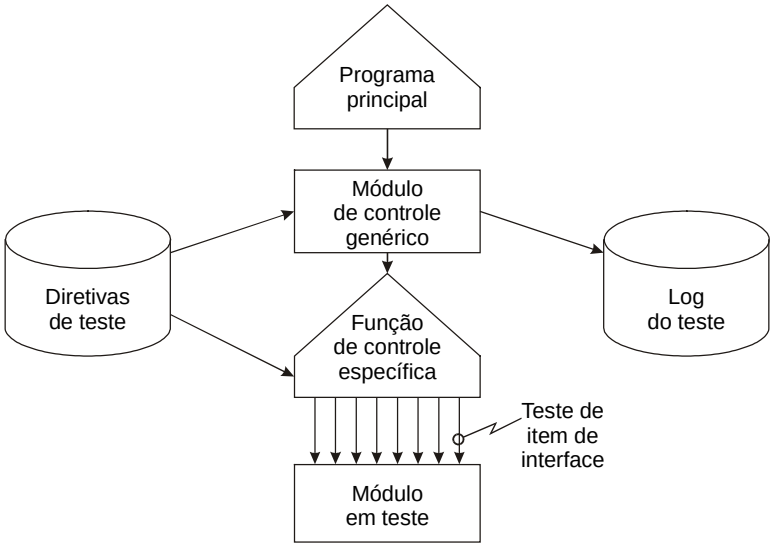

Figura 4. Arquitetura para teste utilizando arquivos de diretivas

A arquitetura desta modalidade de automação de testes é ilustrada na Figura 4. Ela combina alguns aspectos do teste manual com outras do teste automatizado, utilizando um módulo de teste específico. Para cada item de interface do módulo a testar redige-se um pequeno código de controle que estabelece a interface com o item. Este código estará contido na função de controle específica. O código de cada item de interface também lê, de um arquivo de diretivas, os dados e os resultados esperados. Após lido 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 um item de interface.

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

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

            NumLidos = sscanf( TST_Buffer , "%*s %c %i" ,
                               &ValorDado , &CondRetEsperada ) ;
            if ( NumLidos != 2 )
            {
               return TST_CondRetParm ;
            } /* if */

            CondRetObtido = ARV_InserirEsquerda( ValorDado ) ;

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

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

O *programa principal* é uma função (`main`) que faz parte do módulo `PRINC`. O programa principal lê os parâmetros de linha de comando e coordena a execução dos testes. Caso o desenvolvimento se dê de maneira *top-down*, a função deve ser substituída por outra que re-

cebe os parâmetros, indicando o arquivo de diretivas a utilizar e, opcionalmente, o arquivo de `log` dos testes. Esta outra função poderá ser chamada de algum ponto no construto.

O *módulo de controle genérico*, módulo `TESTEGEN`, 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 tabela a seguir apresenta os comandos genéricos disponibilizados pelo arcabouço disponibilizado em `Teste-Automatizado.zip`:

```
#define     COMMENT_CMD         "//"
#define     INICIO_CMD          "=="
#define     RECUPERA_CMD        "=recuperar"
#define     BKPT_CMD            "=bkpt"
```

A tabela a seguir apresenta as declarações das funções de apoio aos testes disponibilizadas no módulo de controle genérico:

```
int TST_ObterNumCasos( ) ;
int TST_ObterNumLinhas( void ) ;
int TST_ObterNumFalhas( void ) ;

TST_tpCondRet TST_CompararChar( char   ValorEsperado ,
                                char   ValorObtido   ,
                                char * pMensagem       ) ;

TST_tpCondRet TST_CompararInt( int    ValorEsperado ,
                               int    ValorObtido   ,
                               char * pMensagem       ) ;

TST_tpCondRet TST_CompararString( char * ValorEsperado ,
                                  char * ValorObtido   ,
                                  char * pMensagem       ) ;

TST_tpCondRet TST_CompararEspaco( void * ValorEsperado ,
                                  void * ValorObtido   ,
                                  int    TamEspaco     ,
                                  char * pMensagem       ) ;

void TST_ImprimirPrefixo( char * pMensagem ) ;
```

A *função de controle específica*, módulo `TESTESPC`, interpreta os comandos desenvolvidos para exercitar o módulo em teste. Na realidade 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.

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, ao invés de estar contida em um código, como acontece no que foi visto na seção anterior. O arquivo de diretivas é, na realidade, um programa de teste, redigido em uma linguagem de programação *ad hoc* especialmente desenvolvida para testar o módulo em teste. A sua arquitetura é geral para todos os testes realizados com este arcabouço e lembra fortemente uma linguagem do tipo *assembler*. A escolha de uma linguagem com esta estrutura tem por finalidade reduzir os custos de desenvolvimento do interpretador.

A seguir ilustramos parte de um arquivo de diretivas para teste. O arquivo completo é o arquivo `Teste.script`:

```
// Script:    Teste.script
// Autor:     Arndt von Staa
// Data:      23/fev/2003
```

```
// Propósito: Testar a construção e exploração de árvores

== Verificar assertivas de entrada de irdir
=irdir      4

== Verificar comando recuperar
=irdir      0
=recuperar

== Verificar ObterValorCorr relativo a uma árvore inexistente
=obter  !   4

== Verificar recuperar de ObterValorCorr relativo a uma árvore inexistente
=obter  !   0
=recuperar

== Criar árvore
=criar      0   0
=irdir      5

== Inserir à direita
=insdir     a   0

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

Linhas iniciando com // são comentários. Linhas iniciando com == marcam o início de um caso de teste. Linhas iniciando com = são comandos de teste que exercitarão o módulo em teste. Os símbolos dos comandos disponíveis são definidos em uma tabela contida no módulo de controle específico TESTESPC.C e que contém a função de controle específico. Isto permite ao usuário selecionar a terminologia mais apropriada para as diretivas de teste, 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"
```

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[24] pode introduzir mal funcionamento.

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

```
TST_tpCondRet TST_EfetuarComado( char * ComandoTeste )
{

   ARV_tpCondRet CondRetObtido   = ARV_CondRetOK ;
   ARV_tpCondRet CondRetEsperada = ARV_CondRetMemoria ;
                                   /* inicializa para qualquer coisa */
   char ValorEsperado = '?'  ;
   char ValorObtido   = '!'  ;
   char ValorDado     = '\0' ;
```

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

```
   int  NumLidos = -1 ;
   TST_tpCondRet Ret ;
   char EspacEsperado[ 100 ] ;
   char EspacObtido[   100 ] ;

/* Testar ARV Criar árvore */

   if ( strcmp( ComandoTeste , CRIAR_ARV_CMD ) == 0 )
   {
      NumLidos = sscanf( TST_Buffer , "%*s %i" , &CondRetEsperada ) ;
      if ( NumLidos != 1 )
      {
         return TST_CondRetParm ;
      } /* if */

      CondRetObtido = ARV_CriarArvore( ) ;
      return TST_CompararInt( CondRetEsperada , CondRetObtido ,
                              "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 = sscanf( TST_Buffer , "%*s %c %i" ,
                         &ValorDado , &CondRetEsperada ) ;
      if ( NumLidos != 2 )
      {
         return TST_CondRetParm ;
      } /* if */

      CondRetObtido = ARV_InserirDireita( ValorDado ) ;
      return TST_CompararInt( CondRetEsperada , CondRetObtido ,
                              "Retorno errado inserir à direita." );

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

Cada bloco `if` corresponde ao interpretador de um determinado tipo de diretiva. O comando corrente está no vetor de caracteres `TST_Buffer`. A análise do comando é realizada pela função `sscanf`. O formato utilizado ao analisar uma diretiva deve saltar sobre o primeiro `string` da linha (formato `"%*s"`) que corresponde à diretiva a interpretar. O restante da linha são os parâmetros contidos na diretiva sendo interpretada.

De maneira geral, uma diretiva de teste contêm os parâmetros a serem fornecidos à função a ser testada e também os resultados esperados. A função `sscanf` retorna o número de diretivas de formato efetivamente lidas (a primeira – `"%*s"` – não é contada). Isto permite realizar um controle de integridade do comando. Em diversas ocasiões deseja-se controlar também a validade dos dados lidos, o que deve ser realizado por um `if` logo após ao controle do número de dados lidos.

Caso os dados estejam corretos, a função a testar deve ser chamada e a seguir os valores retornados devem ser conferidos. O controle dos valores retornados é realizado por funções que fazem parte do módulo de controle genérico `TESTEGEN`.

6.1 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 algum contexto;
- efetuar alguma operação genérica.

Cada comando ocupa uma linha de texto do arquivo. O padrão usado determina que todos os comandos de ação 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>`
  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 `=recupera` a seguir. Para evitar que acidentalmente se tente interpretar um início de caso de teste como uma diretiva `=recupera`, recomenda-se preceder cada início de caso de teste por uma linha em branco. Isto também contribui para uma melhoria da legibilidade.
- `=recupera`
  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 `=recupera` 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, o comando `=recupera` reportará falha, indicando desta forma que o código de teste não operou como esperado.
- `=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`.

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. O

efeito disso será somente uma complexidade maior do interpretador sem ganhos expressivos em sua funcionalidade.

Através do uso de um programa capaz de analisar o módulo de definição, é possível gerar uma versão inicial da função de controle de teste específica, o que reduz bastante o trabalho de desenvolve-la.

6.2 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 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 um 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. Subprogramas podem ser codificados no módulo de teste específico e disparados por um comando.
- Pode ser necessário alterar o módulo de teste genérico ao desenvolver o módulo de teste específico, reduzindo, assim, a possibilidade de reúso do módulo de controle genérico.

7 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 adicione mais funções. 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 (`make`).

Siga o seguinte roteiro ao desenvolver programas utilizando esta abordagem de teste:

- 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*)
  - ♦ cada função faz nada
  - ♦ se a função deve retornar alguma coisa, retorna algum valor neutro
- Redija a função de teste específico
- 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)
- Compile o programa
- 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 não deve acusar falhas.
- 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.
- Redija o código 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 em teste, no código do módulo em teste, no módulo de controle do 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 módulo esteja completamente implementado.

8 Conclusão

Nesta seção descreveremos as principais vantagens e desvantagens advindas do uso de teste automatizado.

O teste automatizado contribui para um aumento significativo da produtividade[25] e da qualidade dos módulos a serem desenvolvidos. Facilita muito a realização de testes de regres-

[25] ***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.

são. 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 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.
    - ♦ 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 e formatadores são alguns dos exemplos.

9 Composição do exemplo

Nesta seção descreveremos a composição do arquivo ZIP contendo o exemplo de uso do arcabouço e do ambiente de desenvolvimento utilizado.

O arquivo `Teste-Automatizado.zip` contém um exemplo de programa utilizando o arcabouço de teste aqui descrito. A documentação contida nos módulos detalha e complementa este artigo. Recomenda-se fortemente a sua leitura antes de iniciar o desenvolvimento

de um programa utilizando o arcabouço de teste. A seguir descrevemos o conteúdo deste arquivo:

- `PRINC.C`, `PRINC.H` – são os módulos de implementação e de definição do programa principal que faz parte do arcabouço de teste automatizado.
- `TESTEGEN.C`, `TESTEGEN.H` – são os módulos de implementação e de definição do módulo de controle de teste genérico do arcabouço.
- `TESTESPC.C`, `TESTESPC.H` – são os módulos de implementação e de definição do módulo de controle de teste específico para o exemplo editor de árvore utilizado no artigo.
- `ARVORE.C`, `ARVORE.H` – são os módulos de implementação e de definição do exemplo usado neste artigo.
- `teste.script` – é um exemplo de arquivo de diretivas utilizado para testar o módulo árvore. Do ponto de vista de qualidade de teste este exemplo é bastante superficial.
- `laudo.log` – ilustra o *log* gerado pelo teste utilizando `teste.script`.
- `TesteAutomatizado.pdf` – é o texto do presente artigo.
- `Exemplo-Teste.MAK` – é o arquivo de diretivas MAKE utilizado para compilar o exemplo.
- `Exemplo-Teste.EXE` – é o programa executável do exemplo.
- `GMAKE.EXE` – é um programa utilitário que gera o arquivo de diretivas MAKE a partir de uma descrição de composição de um programa.
- `gmake.pdf` – é a documentação de `GMAKE`.
- `PLATC.DAT` – é um arquivo de diretivas que induz o programa `GMAKE` a gerar diretivas de MAKE para o compilador MS Visual Studio 6.0. Ajustando as diretivas pode-se gerar arquivos de diretivas MAKE para uma variedade de compiladores.
- `Exemplo-Teste.cmp` – é o arquivo de diretivas de composição do programa exemplo, é utilizado por `GMAKE`.
- `Exemplo-Teste.LST` – é a listagem da composição do programa exemplo, gerado por `GMAKE`.
- `Exemplo-Teste.BLD` – é o arquivo de diretivas para ligar o programa exemplo a partir de seus módulos, é gerado por `GMAKE`.
- `FAZ.BAT` – é um *batch* que pode ser utilizado para compilar um programa utilizando as diretivas de MAKE geradas por `GMAKE`.

Referências bibliográficas

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

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

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

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

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

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

[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

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

[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