Teste Estrutural 1

Arndt von Staa
Departamento de Informática
PUC-Rio
Maio 2015

Especificação

- Objetivo desse módulo
  - Apresentar os conceitos de teste estrutural visando o teste de unidades. Apresentar critérios de cobertura de instruções, arestas e condições.
- Justificativa
  - O teste estrutural é utilizado para testar módulos e, assim, verificar se os componentes elementares possuem o nível de qualidade necessário.
  - O teste estrutural permite alcançar altos níveis de cobertura
- Texto
  - Pezzè, M.; Young, M.; *Teste e Análise de Software*; Porto Alegre, RS: Bookman; 2008, capítulos 12

Princípios de teste estrutural

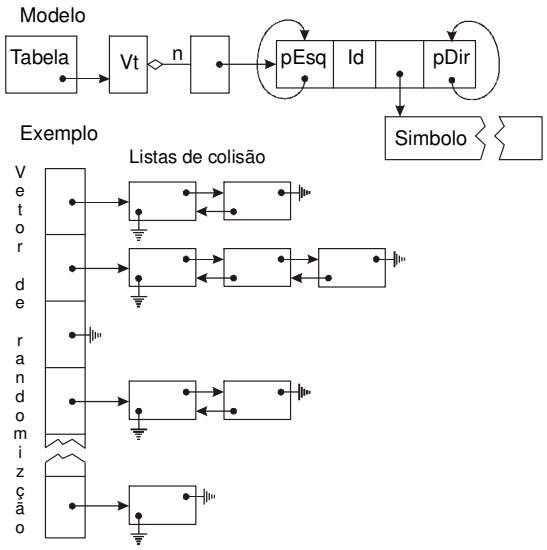
- O teste estrutural utiliza a estrutura do código como base para gerar os casos de teste
  - teste caixa aberta
- Teste estrutural está fortemente relacionado com o teste de unidades
- O teste de unidades é utilizado para testar
  - funções ou métodos
  - classes
  - módulos físicos
- Observação: a adequação e a unicidade das unidades é controlada por inspeção ou verificação
  - adequação: implementa algo de interesse a algum cliente – pessoa ou outra unidade
  - unicidade: implementa um e somente um propósito

Teste de unidade

- Objetivos
  - permitir o reteste completo com frequência
    - requer automação do teste
  - verificar se cada função (método) realiza o que se espera
    - em condições de uso normais
    - em condições de uso anormais
      - em particular: lançamento e tratamento de exceções
    - requer o uso de bons critérios de seleção de casos de teste
  - verificar se as classes (módulos) estão completas e corretas
    - requer o uso de bons critérios de seleção de casos de teste
  - verificar se a documentação técnica do módulo é coerente com o que de fato se encontra implementado
  - verificar a coerência das interfaces com o que de fato está implementado
    - requer testes realizados via a interface do módulo
    - deve-se utilizar a interface conceitual

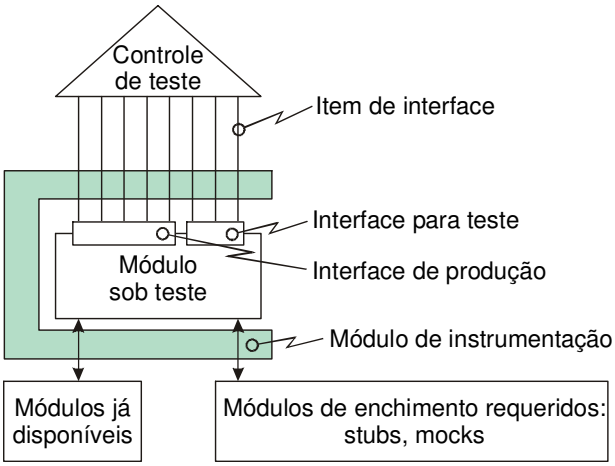
Quais seriam os casos de teste?

- Tabela de símbolos realizada por randomização ("hashing")

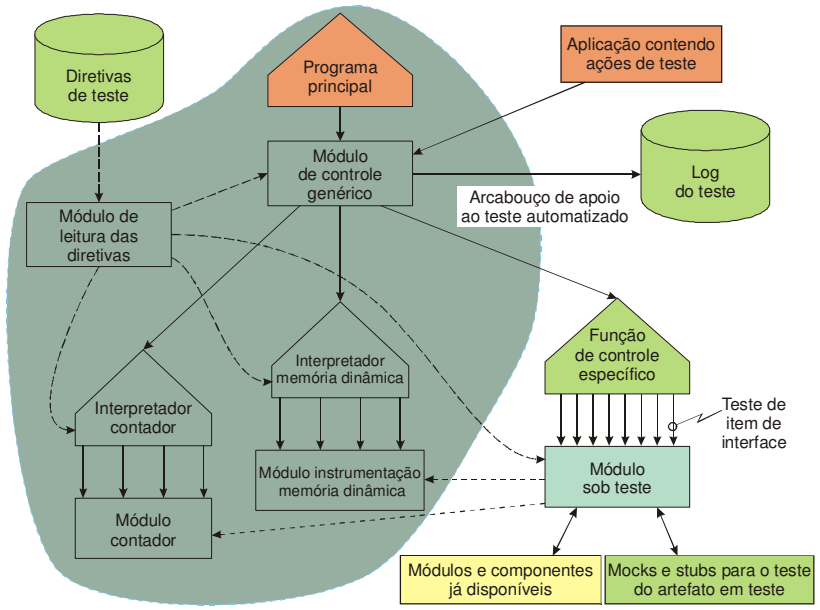


Teste de unidade

- De maneira geral, conhecida a interface da unidade a testar, cria-se um módulo de controle de teste que permite exercitar detalhadamente a unidade a testar



Teste de unidade : meu arcabouço de teste



Critérios de teste estrutural

- Critérios de cobertura da estrutura
  - todas instruções — todos nós
  - todas arestas — todas decisões
  - todas condições
  - todos comandos tipo x
  - todos *fragmentos* de caminho
    - usualmente fragmentos de um determinado comprimento
  - todos caminhos
    - da origem ao término da função isolada
    - da origem ao término da função, levando em conta as funções (método) encapsuladas (**`static`**, **`private`**)
    - da definição (atribuição, declaração) aos usos
  - todos mutantes
  - todas as condições de assertivas estruturais

Grafo da estrutura do código

- Processo genérico para gerar suítes de teste:
  - criar o grafo da estrutura do código a testar
    - pode envolver chamadas a funções ou métodos
  - com base no grafo e no critério de cobertura escolhidos gerar os casos de teste

- Convenções utilizadas

Chamada de função ou método

Sequência de um ou mais comandos, sem conter decisões internas, exceto o último que pode envolver uma decisão

Início ou término de função ou método

As arestas são dirigidas

Uma pseudo-operação (a elipse) pode ter uma ou mais saídas

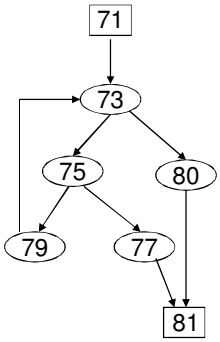
Usualmente são utilizados rótulos de condição

Regras básicas

- Numere cada linha de código
  - cada linha deve conter uma única instrução
- Cada método / função corresponde a um grafo conexo com uma única entrada e uma única saída
  - o grafo inicia e termina com um retângulo
- Cada chamada de método / função corresponde a um hexágono
- Elipses podem corresponder a várias instruções
  - a primeira é início, ou sucessora de chamada, ou ponto de confluência
  - a última é final, ou predecessora de chamada, ou decisão
- Cada **`return`** e cada **`throw`** liga ao retângulo de término do grafo
- Cada seleção (**`if`**, **`switch`**) corresponde a uma elipse com n saídas: **`if`**: 2 ; **`switch`**: tantos quanto forem os **`case`** + 1 para o **`default`**
  - sempre considerar **`default`** mesmo que o código não o contenha
- Cada repetição inicia com uma elipse correspondendo ao controle de início (**`for`**, **`while`**, **`do`**) e termina com uma elipse correspondendo ao fecha chave da repetição
  - o término da repetição liga ao início desta

Exemplo de grafo de estrutura

```
71 int STR_String :: FindStringElement( int idStr )
72 {
73    for( int i = 0 ; i < NUM_STR_MEM ; i ++ )
74    {
75       if ( vtStr[ i ].idString == idStr )
76       {
77          return i ;
78       } // if
79    } // for
80    return -1 ;
81 } // End of: Find the index of the string element
          in the memory resident string table
```

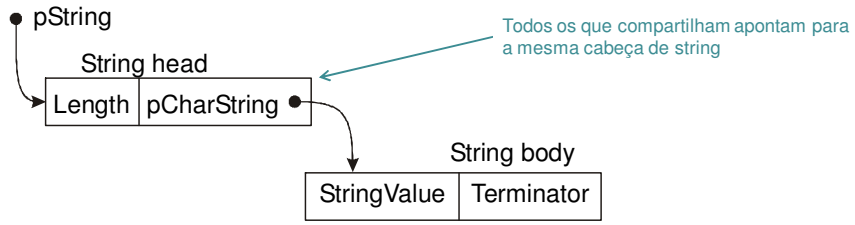


Descrição do exemplo

- *Strings* convencionais (terminados com 0) em C e C++ oferecem grandes riscos de segurança, pois as funções que manipulam *strings* podem muito facilmente extravasar o espaço destino.
- Deseja-se uma forma de processar *strings* que permita
  - compartilhamento de strings
  - qualquer um que compartilha pode alterar o string
  - não ofereça este risco de extravasão

Um possível modelo de solução

- Assertiva estrutural de um string



Structural assertions

```
pString->Length >= 0
pString->Length == 0 => pString->pCharString == NULL
pString->Length > 0   => (pString->pCharString)[ pString->Length ] == 0
```


Tabela de constantes string

- A tabela é um vetor de elementos
- { int idString , int len , char str[DIM_MAX_STR] }
  - para fins de simplicidade utilizamos strings de tamanho fixo
  - uma implementação mais eficiente utilizaria strings de tamanho variável, reduzindo assim a perda de memória devida à fragmentação
- Exemplo

```
  0 ,  0 , ""
  1 , 23 , "idString não existe: %d"
  2 , 19 , "Tabela desconhecida"
  3 , 14 , "string exemplo"
  4 ,  1 , "a"
999 , 13 , "fim da tabela"
```

Tabela de constantes string

- Constantes *string* podem ser armazenadas em tabelas, sendo que, na tabela, cada *string* é representado da forma convencional C/C++
  - não ocorre risco, pois a tabela é constante, i.e. não pode ser destino de armazenamento
  - facilita a criação da tabela, pois usa construções normais da linguagem
- Nesta tabela cada *string* é identificado por um **`idString`**
- Cada elemento da tabela informa o tamanho e o valor do *string*
- Podem-se utilizar diversas tabelas, por exemplo uma residente em memória outra em arquivo em disco.
  - o exemplo utiliza somente a tabela residente em memória
  - as outras possibilidades não estão implementadas
    - o ponto de inserção está implementado
  - o **`idString`** identifica também a tabela física em que se encontra o *string*

Exemplo

- Deseja-se implementar o construtor da classe **`STR_String`** que constrói um objeto *string* copiando-o da tabela

```
STR_String( int idString )
```

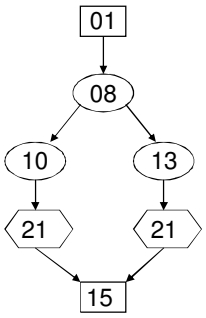
- **`idString`** identifica a tabela e o string contido na tabela

```
idString ::= < idTabela , idStringTabela >
```

- isso pode ser implementado em um único inteiro **`unsigned`** em que alguns bits correspondem ao **`idTabela`** e os outros ao **`idStringTabela`**
- esta solução encapsula a forma de acesso ao string
  - somente quem for construir as tabelas precisa saber da existência de **`idTabela`**

Grafo da estrutura do código: exemplo

```
01 void STR_String ::
02           STR_String( int idString )
03 {
04    #ifdef _DEBUG                                  } instrumentação de
05       EXC_ASSERT( idString >= 0 ) ;               } teste não faz parte
06    #endif                                         } do grafo
07    int idStr = ( idString & STR_ID ) ;
08    if ( ( idString & STR_TABELA ) == STR_MEM )
09    {
10       BuildString( idStr & STR_ID ) ;
11    } else
12    {
13       BuildString( STR_NotImplemented & STR_ID ) ;
14    } // if
15 } // End of: Construct a string given an idString
```

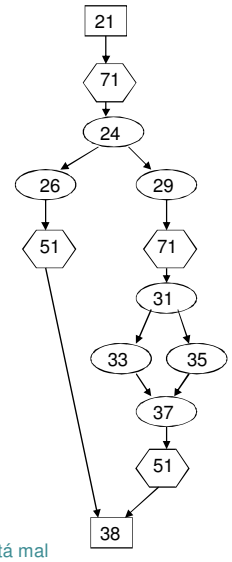


STR_TABELA e **STR_ID** são máscaras utilizadas para decodificar um **idString** e obter, respectivamente, a chave de acesso à tabela a usar, e a chave de acesso à constante string contida nesta tabela. **STR_MEM** é o **idTabela** da tabela residente em memória real.

STR_NotImplemented é uma constante string que informa que a tabela almejada não está implementada.

Grafo da estrutura do código: exemplo

```
21 void STR_String :: BuildString( int idString )
22 {
23    int i = FindStringElement( idString ) ;
24    if ( i >= 0 )
25    {
26       BuildString( vtStr[ i ].len , vtStr[ i ].str ) ;
27       return ;
28    } // if
29    char msg[ DIM_MSG ] ;
30    i = FindStringElement( STR_ErrorUndefinedId ) ;
31    if ( i >= 0 )
32    {
33       sprintf( msg , vtStr[ i ].str , idString ) ;
34    } else {
35       sprintf( msg , ILLEGAL_TABLE , idString ) ;
36    } /* if */
37    BuildString( strlen( msg ) , msg ) ;
38 } // End of: Build string for a given Id
```

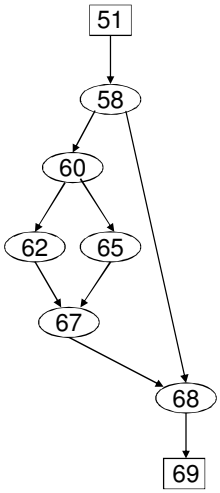


STR_ErrorUndefinedId é uma constante *string* que informa que o string solicitado não existe

ILLEGAL_TABLE é uma constante literal informando que a tabela de *strings* está mal formada ➔ não contém os *strings* obrigatórios definidos pelo projeto

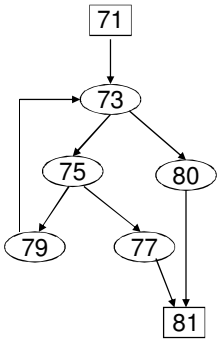
Grafo da estrutura do código: exemplo

```
51 void STR_String :: BuildString( int    len ,
53                                char * pStr )
54 {
55    delete pCharString ;
56    length      = len ;
57    pCharString = new char[ len + 1 ] ;
58    if ( len > 0 )
59    {
60       if ( pStr != NULL )
61       {
62          memcpy( pCharString , pStr , len ) ;
63       } else
64       {
65          memset( pCharString , STR_NULL_CHAR , len ) ;
66       } /* if */
67    } /* if */
68    pCharString[ len ] = 0 ;
69 } // End of: Build a string of a given size
```



Grafo da estrutura do código: exemplo

```
71 int STR_String :: FindStringElement( int idStr )
72 {
73    for( int i = 0 ; i < NUM_STR_MEM ; i ++ )
74    {
75       if ( vtStr[ i ].idString == idStr )
76       {
77          return i ;
78       } // if
79    } // for
80    return -1 ;
81 } // End of: Find the index of the string element
          in the memory resident string table
```



A pesquisa é sequencial para simplificar o exemplo

Arrasto

- Como testar repetições?
- O custo do teste cresce com o número de iterações
  - portanto o número de iterações de cada repetição a testar deverá ser:
    - o menor possível, de modo a minimizar o custo
    - mas suficientemente grande para que o teste tenha elevada eficácia, i.e. seja uma boa aproximação de um teste confiável

Recordação:
- um teste é confiável caso ele detecte todos os defeitos porventura existentes no código. É virtualmente impossível criar testes confiáveis, daí a "aproximação".
- eficácia mede o percentual dos defeitos existentes descobertos pelo teste. Como não sabemos quantos defeitos existem temos que utilizar uma aproximação.

Arrasto

- Arrasto
  - é o maior dos menores números de iterações necessárias para que todas as variáveis ou estados inicializados antes da e modificados durante a repetição passem a depender exclusivamente de valores computados em iterações anteriores de uma mesma instância de execução dessa repetição
  - corresponde ao número mínimo de iterações para atingir o primeiro estado "genérico", considerando o conjunto de alternativas de menores caminhos

Arrasto: força de resistência ao avanço de um objeto em um fluido.

Arrasto: exemplos

- Exemplos:
  - `A[0] = 0 ; A[1] = 0 ; A[2] = 0 ; A[3] = 0 ;`
  - `memset( A , 0 , sizeof( A )) ;`
  - `pElem = ProcurarSimbolo( pTabela , pSimbolo ) ;`
  - todos têm Arrasto == 0
    - chamadas de funções são tratadas como uma operação simples quando os conjuntos de parâmetros e de retornos forem disjuntos, considerando a *assinatura conceitual*
      - lembre-se que a assinatura conceitual considera sendo parâmetros as variáveis membro de objeto e de classes, e as variáveis persistentes

Arrasto: exemplos

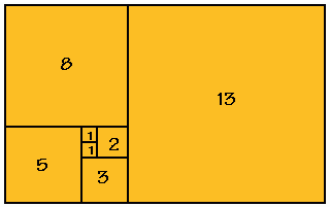
- `for ( i = 0 ; i < 10 ; i++ ) ...`
- `for ( pElem = pOrg ; pElem != NULL ; pElem = pElem->pProx ) ...`
- `fgets, fputs, fread, fwrite`

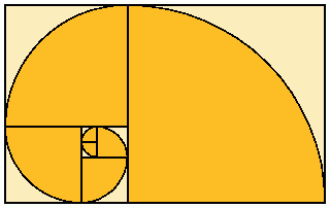
```
tpEstado Corrente ;
Corrente = DefinirPrimeiro( ValorProcurado ) ;  // arrasto == 0
while ( !Terminou( Corrente ))
{
  if ( Comparar( ObterValor( Corrente ), ValorProcurado )
            == EH_IGUAL )
  {
    return Corrente ;
  } /* if */
  Corrente = DefinirProximo( Corrente , ValorProcurado ) ;
} /* while */                                  arrasto == 1
return ESTADO_NIL ;
```

- Todos têm Arrasto == 1

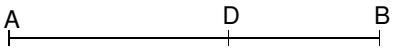
Arrasto: exemplos

- Série de Fibonacci: 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, ...
  - inicializados
  - lei de formação $F_n = F_{n-2} + F_{n-1}$
  - tem Arrasto == 2





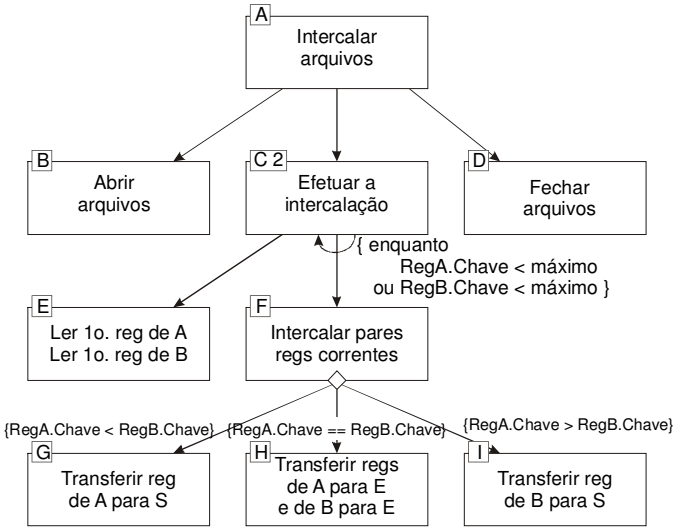
Média áurea
Leonardo da Vinci



$$\frac{AB}{AD} = \frac{AD}{DB} = \phi = \frac{1+\sqrt{5}}{2} = 1{,}61803...$$

Matemática Essencial

Arrasto: exemplos



- Tem Arrasto == 2

Arrasto e argumentação

- Os casos de teste a serem criados para repetições são:
  - caso 0 iteração (caso especial)
  - caso 1 iteração (base da indução)
  - caso *n* >= *arrasto* + *1* iterações (simula o passo de indução)
  - devem sempre ser considerados todos os casos de término:
    - **`break`** ou **`return`** no corpo da iteração
      - **`throw`** requer tratamento especial dependendo do contexto em que está sendo usado
    - ver argumentação da corretude de repetições, aula: *Aspectos formais 2: Proposições para a repetição*

Arrasto, por que 0 iterações?

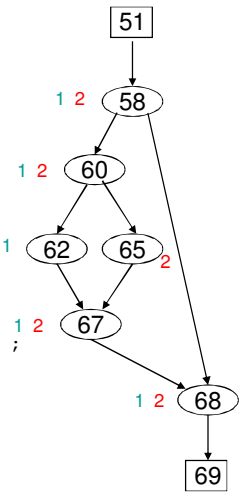
- A preocupação ao testar programas é obter uma informação (quase) confiável a respeito da qualidade do artefato
  - isso pode requerer o uso de dados pouco plausíveis do ponto de vista uso produtivo do programa, exemplos:
    - intercalar dois arquivos vazios produzindo arquivos vazios
    - procurar um símbolo em uma tabela vazia
    - calcular o produto de matrizes de tamanho 1 × 1
  - se o algoritmo for incapaz de tratar estes usos, ele seria capaz de operar corretamente **em todos** os outros casos?

Critério de cobertura de instruções

- Cobertura de instruções
  - todos os nós
  - Cada instrução é executada pelo menos uma vez no conjunto de todos os casos de teste
  - dado o grafo
    - cada caso percorre pelo menos um vértice (nó) ainda não percorrido
    - até que todos os vértices tenham sido percorridos
  - repetições devem ser executadas para
    - n > 1 iterações;

Critério de cobertura de instruções: exemplo

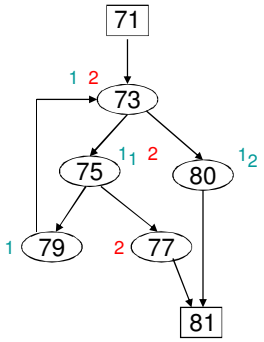
```
51 void STR_String :: BuildString( int    len ,
53                                 char * pStr )
54 {
55    delete pCharString ;
56    length      = len ;
57    pCharString = new char[ len + 1 ] ;
58    if ( len > 0 )
59    {
60       if ( pStr != NULL )
61       {
62          memcpy( pCharString , pStr , len ) ;
63       } else
64       {
65          memset( pCharString , STR_NULL_CHAR , len ) ;
66       } /* if */
67    } /* if */
68    pCharString[ len ] = 0 ;
69 } // End of: Build a string of a given size
```



Casos de teste abstratos:
< 51 58 60 62 67 68 69 > ; < 51 58 60 65 67 68 69 >

Critério de cobertura de instruções: exemplo


```
71 int STR_String :: FindStringElement( int idStr )
72 {
73    for( int i = 0 ; i < NUM_STR_MEM ; i ++ )
74    {
75       if ( vtStr[ i ].idString == idStr )
76       {
77          return i ;
78       } // if
79    } // for
80    return -1 ;
81 } // End of: Find the index of the string element
          in the memory resident string table
```



n > 1 ciclos

Casos de teste abstratos: 1: < 71 73 75 79 73 75 79 73 80 81 > e 2: < 71 73 75 77 81 >

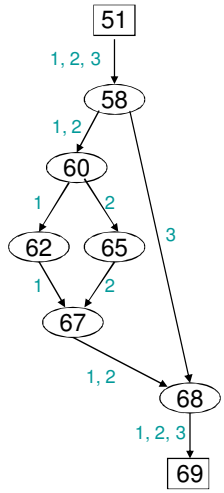
precisa agora converter em caso de teste semântico 1: tabela contém dois elementos diferentes do elemento procurado, e 2: tabela contém um ou mais elementos e o primeiro deles é encontrado.

Critério de cobertura de arestas

- Cobertura de arestas
  - Cada aresta é percorrida pelo menos uma vez no conjunto de todos os casos de teste
  - rotulam-se as arestas e criam-se os casos de teste
    - cada caso percorre pelo menos uma aresta ainda não percorrida
    - até que todas as arestas tenham sido percorridas
  - repetições devem ser executadas para:
    - n = 0 iterações
    - n = 1 iteração
    - n >= 3 iterações
    - por que 3 ou mais iterações?
      - permite tratar os casos: início, meio e fim
      - assume que o arrasto é 1. Para outros valores é necessário ajustar o número de iterações

Critério de cobertura de arestas: exemplo

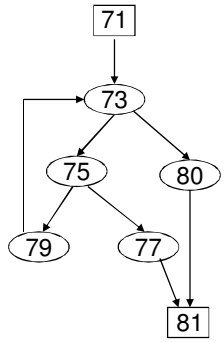

```
51 void STR_String :: BuildString( int    len ,
53                                char * pStr )
54 {
55    delete pCharString ;
56    length      = len ;
57    pCharString = new char[ len + 1 ] ;
58    if ( len > 0 )
59    {
60       if ( pStr != NULL )
61       {
62          memcpy( pCharString , pStr , len ) ;
63       } else
64       {
65          memset( pCharString , STR_NULL_CHAR , len ) ;
66       } /* if */
67    } /* if */
68    pCharString[ len ] = 0 ;
69 } // End of: Build a string of a given size
```



Casos de teste abstratos:
< 51 58 60 62 67 68 69 > ; < 51 58 60 65 67 68 69 > ; < 51 58 68 69 >

Critério de cobertura de arestas: exemplo


```
71 int STR_String :: FindStringElement( int idStr )
72 {
73    for( int i = 0 ; i < NUM_STR_MEM ; i ++ )
74    {
75       if ( vtStr[ i ].idString == idStr )
76       {
77          return i ;
78       } // if
79    } // for
80    return -1 ;
81 } // End of: Find the index of the string element
          in the memory resident string table
```



Casos de teste para zero iterações: < 71 73 80 81 > e < 71 73 75 77 81 >
Casos de teste para uma iteração: < 71 73 75 79 73 80 81 > e < 71 73 75 79 73 75 77 81 >
Casos de teste para três iterações < 71 73 75 79 73 75 79 73 75 79 73 80 81 >
e < 71 73 75 79 73 75 79 73 75 79 73 75 77 81 >

Critério de cobertura de condições

- Cobertura de condições
  - Cada forma de avaliar expressões lógicas compostas é exercitada pelo menos uma vez no conjunto de todos os casos de teste
  - repetição, em adição à cobertura de arestas:
    - cada uma das formas de se decidir pelo término foi exercitada
      - **break**, **return** ou **throw** no corpo da repetição
      - expressão de controle de término composta

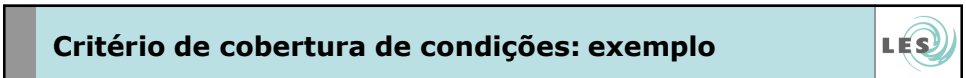
Critério de cobertura de condições: exemplo

- Esquema do algoritmo para pesquisa em qualquer tabela

```
tpEstado Corrente ;
Corrente = DefinirPrimeiro( ValorProcurado ) ;
while ( !Terminou( Corrente ))
{
  if ( Comparar( ObterValor( Corrente ),
          ValorProcurado ) == EH_IGUAL )
  {
    return Corrente ;
  } /* if */
  Corrente = DefinirProximo( Corrente , ValorProcurado ) ;
} /* while */
return ESTADO_NIL ;
```

- arrasto == 1

Critério de cobertura de condições: exemplo



- Caso 0 iterações:
  - tabela vazia
  - tabela com 1 ou mais elementos e acha o primeiro elemento
    - tratar caso 1 elemento
    - tratar caso 2 elementos

    } condição de contorno
- Caso 1 iteração:
  - tabela com 1 elemento e não acha o elemento
  - tabela com 2 ou mais elementos e acha o segundo elemento
    - tratar caso 2 elemento
    - tratar caso 3 elementos
- Caso arrasto + 1 iterações:
  - tabela com 2 elementos e não acha o elemento
  - tabela com 3 ou mais elementos e acha o segundo elemento
    - tratar caso 3 elemento
    - tratar caso 4 elementos

Tabela hash – o teste funciona?

- Os casos de teste do slide anterior não são suficientes para tabelas de randomização (hash)
- Os casos de teste abstratos
  - testam o conteúdo de uma lista de colisão
  - mas não testam o conjunto de listas de colisão
- O problema da tabela hash está na especificação de "definir primeiro"
  - segundo as condições de contorno devem-se testar a primeira lista, a última e uma no meio
- E como ficaria o teste para o caso de resolução de colisão no corpo da tabela?

Referências bibliográficas

- Hunt, A.; Thomas, D.; eds.; *Pragmatic Unit Test: in Java with JUnit*; Sebastopol, CA: O'Reilly; 2003
- Myers, G.J.; *The Art of Software Testing*, 2nd edition; New York: John Wiley & Sons; 2004
- Pezzè, M.; Young, M.; *Teste e Análise de Software*; Porto Alegre, RS: Bookman; 2008
- Staa, A.v.; *Programação Modular*; Rio de Janeiro, RJ: Elsevier / Campus; 2000

FIM