




Laboratório de Engenharia de Software

# Aspectos Formais de Apoio aos Testes 1

Arndt von Staa  
Departamento de Informática  
PUC-Rio  
Maio 2015

## Especificação



Laboratório de Engenharia de Software

- Objetivo desse módulo
  - mostrar a importância do uso de **técnicas formais leves**; apresentar a técnica de especificação de métodos ou funções: *design by contract*, e modos de criar programas auto-verificantes.
- Justificativa
  - o teste baseado em assertivas requer um bom domínio do uso de assertivas e de argumentação da correteza
  - é necessário complementar estes conceitos com os aspectos diretamente relacionados com orientação a objetos, uma vez que não são vistos em PMod ou PPG.
- Texto
  - Pezzè, M.; Young, M.; *Teste e Análise de Software*; Porto Alegre, RS: Bookman; 2008, capítulo 7
  - Staa, A.v.; *Programação Modular*; Rio de Janeiro: Campus/Elsevier; 2000, capítulo 13

Maio 2015Arndt von Staa © LES/DI/PUC-Rio2

## Especificação de funções (métodos)



- Categorias de interface

- interface conceitual

$F(p_1, p_2, \dots, p_n) ::= r_1, r_2, \dots, r_k$

↓ produz

- assinatura (definição física, ou declaração)

$F(\text{tipo}_1 \text{ parm}_1, \dots, \text{tipo}_n \text{ parm}_n) ::= \text{tiporesultado}$

muitas linguagens OO  
não consideram isso  
como parte da assinatura

↓  
existem linguagens, e.g. Lua, em que  
funções podem retornar mais de um valor

- Um **arquivo** (tabela) é uma **variável persistente**, um registro (linha de uma tabela) também
- Exemplos de estados: pilha vazia; elemento gráfico selecionado; arquivo não existe; arquivo aberto.

## Interface conceitual de funções (métodos)



- $F(p_1, p_2, \dots, p_n) ::= r_1, r_2, \dots, r_k$

- $p_i$  são os **parâmetros conceituais**, podem ser

- parâmetros físicos passados por valor ou referência (lista de parâmetros), variáveis membro de objetos, variáveis membro de classes, variáveis globais, variáveis persistentes, estados do sistema, estados da estrutura de dados, estados da classe, estados do objeto


- $r_i$  são os **resultados conceituais**

- valores retornados, parâmetros físicos passados por referência (lista de parâmetros), variáveis membro de objetos, variáveis membro de classes, variáveis globais, variáveis persistentes, estados do sistema, estados da estrutura de dados, estados da classe, estados do objeto
    - devem também estar registradas as **exceções geradas por P**.
      - Alguns dizem que devem ser registradas também as exceções que podem ser geradas por servidores de P. Infelizmente, isso costuma criar uma imensa trabalhadeira de manutenção.

Laboratório de Engenharia de Software

## Especificação usando assertivas de entrada e saída

versão inicial



```

AE: idUsuario      lexicamente correto
    senha          lexicamente correta
    direitosUso    lista válida
    ExisteUsuario( idUsuario , senha ) == FALSE
    numeroPares    >= 0

CadastrarNovoUsuario( idUsuario, senha, direitosUso )

AS: numeroPares == entrada :: numeroPares + 1
    ObterAutorizacaoUsuario( idUsuario , senha ) ==
        { AUTORIZADO , direitosUso }


```

precisa estar especificado em algum lugar, ideal é que exista um método que avalie estas propriedades

Mai 2015
Arndt von Staa © LES/DI/PUC-Rio
5

Laboratório de Engenharia de Software

## Condição envolvendo a execução de código



- Seja  $P$  um fragmento de código (usualmente, mas não necessariamente, um método), e sejam  $AE$ ,  $AS$  e  $AINV$  respectivamente as **assertivas de entrada**, **de saída** e **invariante** deste código, para  $P$  **poder** estar correto, deve valer a expressão:

$? AINV, AE :: P \Rightarrow AS, AINV$

  - $AINV, AE$  – envolve todos os parâmetros da interface conceitual
  - $AINV, AS$  – envolve todos os resultados da interface conceitual
  - $AINV$  – deve valer antes e após executar  $P$
- Por que o pé atrás: “ $P$  pode estar correto”?
  - um fragmento de código pode estar em erro, mesmo quando supostamente provado correto → erros ao provar

Yelowitz, L.; Gerhart, S.L.; “Observations of fallibility in applications of modern programming methodologies”; *IEEE Transactions on Software Engineering* 2(9); Los Alamitos, CA: IEEE Computer Society; 1976; pags 195-207

Mai 2015
Arndt von Staa © LES/DI/PUC-Rio
6

## Passo de argumentação



- Argumentar a corretude de um fragmento de código é mostrar que
    - dada a **validade** das assertivas de entrada e invariante (ou da classe, ou estrutural) **antes** de iniciar a execução do fragmento
      - lembre-se: nada pode ser concluído a partir de uma **premissa falsa**
    - a execução do fragmento **implica a validade** das assertivas de saída e invariante ao terminar a execução do fragmento
    - o fragmento de código **sempre terminará** de executar
      - devem ser levados em conta **todas as formas de se terminar** a execução do fragmento
        - atingir o final do fragmento
        - **break**
        - **continue**
        - **return**
        - **throw**
- Lembre-se: **invariante** é uma expressão lógica, não são os valores que esta usa. A invariante deve ser **true** antes que os valores sejam alterados e após serem alterados. Se não forem alterados temos uma constante (do ponto de vista do código), mas não uma invariante, i.e. temos uma AE.

## Exemplo de uma especificação formal



AINV: ← Quando contém nada, pode ser omitida

AE:

NomeArq – o **nome completo** do arquivo, um string ASCII com no máximo **DIM\_NOME\_ARQ** caracteres

Modo – como deve ser aberto o arquivo, ver **tpModo**

*AbrirArquivo( string NomeArq , tpModo Modo ) =::*  
*FILE\* pArq , tpCondRet CondRet*

AS:

CondRet == OK =>

entrada::NomeArq – é sintaticamente válido,  
o arquivo existe, e pode ser acessado por este programa  
pArq é o ponteiro para o descritor do arquivo aberto

CondRet != OK =>

entrada::NomeArq – ou não vale, ou não existe, ou não é permitido acessá-lo  
pArq == NULL  
CondRet informa o detalhe do erro, ver **tpCondRet**

Não muda o valor, mas muda o que sabemos a respeito dele

## Avaliação de assertivas de entrada



### CrITÉRIOS de avaliação da qualidade de assertivas de entrada

controle a satisfação dessas regras por intermédio de inspeções ou revisões

Na expressão ?  $AINV, AE :: P \Rightarrow AS, AINV$

- $AINV$  e  $AE$  devem mencionar
  - **tudo** (dados, estados e recursos) que será usado pelo fragmento  $P$  antes de redefinir, i.e. **atualizar** ou **destruir**
  - **nada do que será criado** por  $P$
  - **nada do que não** será utilizado por  $P$ 
    - no caso de funções (métodos), considere a **interface conceitual**
    - no caso de fragmentos de código considere os elementos usados no fragmento
- $AE$  deve mencionar nada que seja mencionado na  $AINV$ 
  - $AINV$  é *ao mesmo tempo* assertiva de entrada e de saída
  - as condições que perfazem a  $AINV$  devem ser necessárias na entrada **e** na saída

## Avaliação de assertivas de entrada



- Todas as condições considerando  $AE$  e  $AINV$  devem **efetivamente** poder ser **assumidas verdadeiras** ao iniciar
  - no caso de **fragmentos em sequência**, cabe aos fragmentos de código que, na sequência de execução, antecedem uma assertiva de entrada ou invariante, assegurar que estas sejam verdadeiras
  - no caso de **chamadas**, cabe ao fragmento que antecede a execução da chamada **física** (i.e. estado e argumentos) assegurar que a assertiva de entrada da função seja verdadeira para cada um dos argumentos
    - isso é necessário pois o resultado da **avaliação** das expressões **argumento** precisam ser coerentes com a especificação dos correspondentes parâmetros
    - também vale para as invariantes de objeto e estruturais e para as invariantes envolvendo variáveis globais (**static**)


**chamada física** – é o código gerado pelo compilador e que executa a instrução de máquina **call**

**chamada lógica** – é o fragmento de *código redigido pelo programador* visando chamar uma função ou método, i.e. a chamada mais a lista de expressões argumento.

Laboratório de Engenharia de Software

## Avaliação de assertivas de entrada

- Assertivas de entrada devem ser **avaliadas por instrumentação ativa** contida no código **sempre** que envolvam dados não confiáveis:
  - fornecidos por um usuário humano
  - recebidos através da rede
  - lidos de arquivos com procedência não confiável
  - forem utilizados em fragmentos de código que possam provocar danos elevados
    - exemplo: antes de gravar dados em um banco de dados
- **Por que?** Caso uma assertiva de entrada não se verifique
  - qualquer resultado estará “correto” por mais absurdo que seja
    - qualquer coisa pode ser concluída a partir de uma premissa falsa
  - logo, sempre que as fontes de dados não forem confiáveis, é necessário verificar a validade dos dados por meio de código
    - infelizmente nem sempre se pode avaliar a **acurácia** – i.e. a correspondência exata entre o dado e o mundo real



Mai 2015
Arndt von Staa © LES/DI/PUC-Rio
11


controle a satisfação dessa regra por intermédio de inspeções ou revisões

Laboratório de Engenharia de Software

## Avaliação de assertivas de saída

Na expressão ?  $AINV, AE :: P \Rightarrow AS, AINV$

- $AS$  e  $AINV$  devem mencionar
  - **tudo** (dados, estados e recursos) que for **atualizado**, tudo que for **criado**, e tudo que for **destruído** pelo fragmento  $P$ 
    - no caso de funções (métodos), considere a **interface conceitual**
    - no caso de fragmentos de código considere os elementos usados no fragmento
  - **nada que for somente utilizado** por  $P$
  - **nada que seja local** a  $P$ , ou que somente interesse no âmbito de  $P$  (criado e destruído no âmbito de  $P$ )
    - é definido e deixa de ter interesse ao sair do contexto de  $P$
    - entretanto, **efeitos colaterais implícitos**, por exemplo ausência de vazamento de recursos (e.g. memória) provocado por variável local, precisam ser registrados
- $AS$  deve mencionar nada do que é mencionado em  $AINV$



Mai 2015
Arndt von Staa © LES/DI/PUC-Rio
12

## Avaliação de assertivas de saída



- AS e AINV devem conter (continuação)
  - os itens de AE atualizados e que serão substituídos em AS pelas alterações realizadas por P
  - para referenciar na saída os dados de entrada utilizaremos:  
**entrada :: nome**
    - outras notações: old:nome; nome ; nome

## Avaliação de assertivas de saída



- AS de uma função (método) deve mencionar ainda:
  - a relação de **todos os recursos** que devem ser desalocados ou liberados ao sair da função
    - ponteiros para memória dinâmica
      - em Java ou outra linguagem que realize coleta de lixo automática, deve-se atribuir `null` às referências de objetos não mais necessários
    - recursos requisitados ao sistema operacional,
      - ex. arquivos, janelas
  - a relação de **todas as exceções geradas** pela função
    - mas não as que podem ser geradas por funções chamadas por P
    - há os que exigem “todas as exceções” que possam ser passadas por P. Infelizmente isso é difícil, tende a gerar listas enormes, e dificulta a manutenção

## Avaliação de assertivas de saída



- A assertiva de saída da função deve relacionar os resultados considerando **todas as formas de término** da execução
  - `return` no meio do corpo
  - para C++ , Java , C# : `throw`
- A assertiva de saída, ou uma especificação de requisitos, deve especificar o que ocorrerá se a função for cancelada pelo processador de exceções ao procurar por um *catch* não tratado pela função
  - item importante para evitar vazamento de recursos ou de memória

## Avaliação de assertivas de saída



- Sempre explicitar se a função **não retorna**
  - são funções que utilizam alguma função do gênero `exit ( )`
  - muitas vezes são funções que provocam o cancelamento do programa caso seja detectada uma falha ou algum dado errado
    - função de **arrumação da casa** (*housekeeping*) usualmente cancela o processamento
    - procure usar `throw` ou condições de retorno para sinalizar falhas, evite cancelamentos ao detectar erros (falhas)
      - a exceção será capturada em algum lugar em que se possa ativar com segurança a função de arrumação da casa
      - forneça parâmetros no `throw` que permitam saber-se por que falhou
    - **funções que cancelam são um risco** muito grande quando se desenvolve sistemas que devem operar continuamente, ex.
      - comércio eletrônico
      - controle de processo



## Design by contract



- Pré-condição do método i.e. **assertiva de entrada**
  - condições que devem estar satisfeitas para poder ativar o método
- Pós-condição do método i.e. **assertiva de saída**
  - condições que devem estar satisfeitas após a execução do método
  - caso tenha sido ativado com uma pré-condição válida
- Invariante da classe, i.e. **assertiva estrutural da classe**
  - condições que devem ser verdadeiras envolvendo todos os atributos da classe (estado da classe) quando não existir método da classe (inclusive herdeiros, ou redefinidos) ativo
- **Assertiva estrutural**
  - condições que devem ser satisfeitas por um conjunto de objetos inter-relacionados possivelmente pertencentes a diferentes classes

Meyer, B.; "Applying Design by Contract"; *IEEE Computer* 25(10); 1992; pages 40-51

## Design by contract, exemplo




```
class Dicionario
{
    ...
    Adicionar( Chave c ; Valor v )
    {
        requer
            tem_espaco( ) ;
            not existe( c ) ;
        assegura
            numElem( ) == old numElem( ) + 1 ;
            existe( c ) ;
            obtervalor( c ) == v ;
    }
}
```

Assertivas de entrada e saída como contrato		LES
Laboratório de Engenharia de Software	<b>AINV:</b> <code>EhValidoCadastroUsuarios ( )</code> <b>AE:</b> <code>EhValidoUsuario( idUsuario )</code> <code>EhValidoSenha( senha )</code> <code>EhValidoDireitos( listaDireitosUso )</code> <code>ExisteUsuario( idUsuario , senha ) == false</code> <code>numParesEntrada = GetNumPares ( ) ;</code> <code>numParesEntrada &gt;= 0</code>  <code>CadastrarNovoUsuario( idUsuario , senha ,</code> <code>listaDireitosUso )</code>  <b>AS:</b> <code>GetNumPares ( ) == numParesEntrada + 1</code> <code>ObterAutorizacaoUsuario( idUsuario , senha ) ==</code> <code>listaDireitosUso</code>	<p><i>idUsuario, senha e listaDireitosUso devem ser imutáveis → constantes</i></p> <p><i>listaDireitosUso pode ser um objeto? Se for, a assertiva de saída está correta?</i></p>
	<p>Mai 2015</p> <p>Arndt von Staa © LES/DI/PUC-Rio</p> <p>19</p>	

Design by contract		LES
Laboratório de Engenharia de Software	<ul style="list-style-type: none"> <li>Os contratos são <b>definidos na interface</b> <ul style="list-style-type: none"> <li>devem mencionar somente propriedades que o código <b>cliente seja capaz de entender ou utilizar</b></li> <li>não devem referenciar atributos ou propriedades <i>protegidas</i> (?) ou <i>privadas</i>, i.e. encapsulados               <ul style="list-style-type: none"> <li>por exemplo, variável contendo o número de elementos de uma estrutura não pode aparecer no contrato</li> <li>porém, uma função pública que retorna o número de elementos pode</li> </ul> </li> <li>podem referenciar <b>propriedades abstratas</b> e atributos acessados via um <b>getter</b> ou uma função predicado, exemplo               <ul style="list-style-type: none"> <li>"pilha não está vazia"</li> <li><code>!Pilha.EhVazia ( )</code></li> <li><code>GetNumPares ( )</code></li> </ul> </li> </ul> </li> </ul>	
	<p>Mai 2015</p> <p>Arndt von Staa © LES/DI/PUC-Rio</p> <p>20</p>	

Laboratório de Engenharia de Software

## Design by contract



	Cliente	Servidor
Pré-condição (Assertiva de entrada)	Precisa assegurar a validade	Assume válida
Pós-condição (Assertiva de saída)	Assume válida	Precisa assegurar a validade

Esta forma de tratar assertivas minimiza a verificação inutilmente repetida das mesmas condições, desde que todo mundo respeite rigorosamente as pré e pós condições

Implica a necessidade de boa especificação das interfaces.


Podemos assumir **com segurança** que clientes e servidores **sempre respeitem os contratos**?

Mesmo quando forem externos à equipe de desenvolvimento?

Mai 2015
Arndt von Staa © LES/DI/PUC-Rio
21

Laboratório de Engenharia de Software

## Design by contract **verificado**

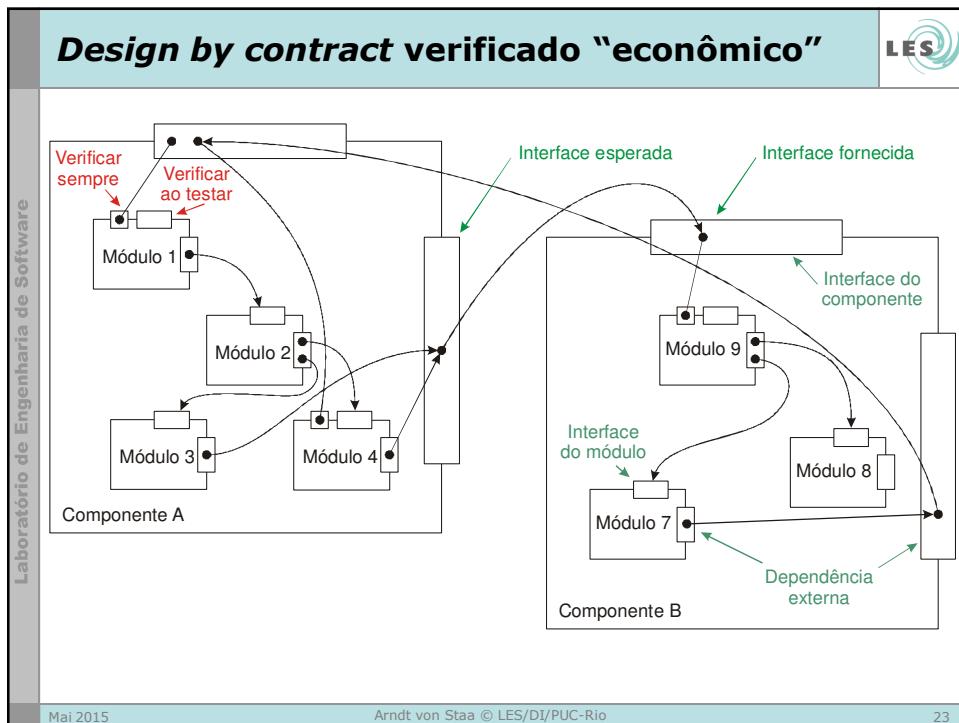


	Cliente	Servidor
Pré-condição (Assertiva de entrada)	Precisa assegurar a validade	<b>Verifica se é válida</b>
Pós-condição (Assertiva de saída)	<b>Verifica se é válida</b>	Precisa assegurar a validade

Problema: **perda de desempenho** – aumento do esforço computacional – em virtude da **repetição redundante** de verificações de mesmas condições em diferentes pontos do programa

Solução: usar **compilação condicional** nos métodos internos (private) da classe ou componente

Mai 2015
Arndt von Staa © LES/DI/PUC-Rio
22



**Design by contract verificado**

- Uso de assertivas executáveis (C++)

```
int LSP_CListPage :: SelectElem( )
{
    #ifdef _DEBUG
        EXC_ASSERT( numElements >= 0 ) ;
        EXC_ASSERT( pPage != NULL ) ;
    #endif
    . . .
}
```

Em Java existe `assert` que gera uma exceção e que, como todas exceções, fornece dados de contexto no momento do `throw`

Meu controle de assertivas 😊

- Gera uma exceção, evita cancelamento imediato
  - se for necessário cancelar, permite fazer antes a "arrumação da casa"

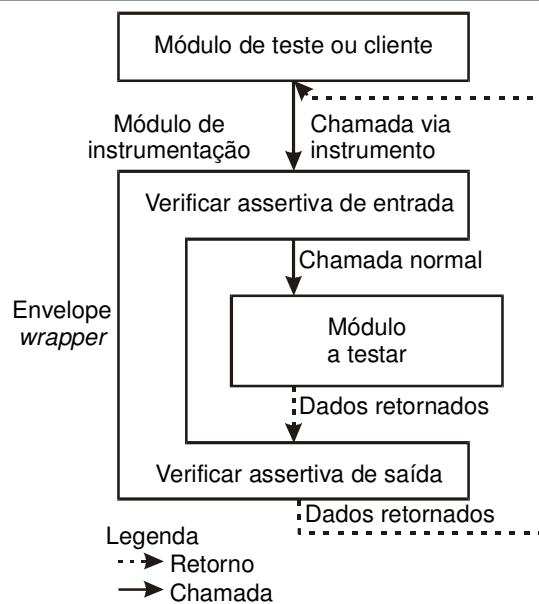
Laboratório de Engenharia de Software

Mai 2015      Arndt von Staa © LES/DI/PUC-Rio      24

## Design by contract verificado



- Uso de **módulos de instrumentação**
  - podem ser ativados ou desativados por meio de uma pequena alteração



Mai 2015

Arndt von Staa © LES/DI/PUC-Rio

25

## Esquema de código, compilação condicional



Cliente contém

```

#ifdef _INSTRUM
#define methodX instrumMethodX
#endif
    
```

um #define para cada um dos métodos públicos a serem controlados

Métodos públicos contidos no módulo de instrumentação

```

tipoR instrumMethodX( tipoA a , tipoB b , ... , tipoN n )
{
    if ( !( assertivaEntrada ) )
        throw new EXC_Assertiva( onde , ... ) ;

    tipoR valRet = methodX( a , b , ... , n ) ;

    if ( !( assertivaSaida ) )
        throw new EXC_Assertiva( onde , ... ) ;
    return valRet ;
}
    
```

como ficaria o código se `methodX` puder gerar uma exceção?

Mai 2015

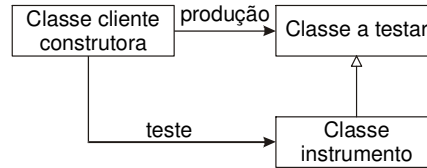
Arndt von Staa © LES/DI/PUC-Rio

26

## Design by contract verificado, objetos



- A *classe cliente* construtora constrói
  - o objeto *classe instrumento*, caso estejam sendo realizados testes da *classe a testar*
  - o objeto *classe a testar*, caso esteja operando em produção normal
- A *classe instrumento* **redefine** todos os métodos a serem controlados
- Cada método redefinido na *classe instrumento* obedece ao esquema ao lado



```

tipoR metodoA( parms )
{
    if( !( AE ) ) exceção ;
    tipoR val = super.metodoA( parms ) ;
    if( !( AS ) ) exceção ;
    return val ;
}
    
```

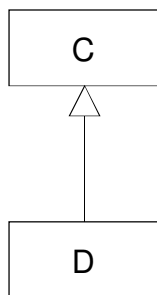
*Problema: o que fazer quando as assertivas dependem de dados privados da classe a testar?*

Mai 2015

Arndt von Staa © LES/DI/PUC-Rio

27

## Design by contract: herança



Super: C

```

void m( p1, ..., pn ) r1, ..., rk
@pre  pC( P )
@post rC( R )
    
```

- $pC(P)$  – pré-condição envolvendo os parâmetros conceituais  $p1, \dots, pn$
- $rC(R)$  – pós-condição envolvendo os retornos conceituais  $r1, \dots, rk$

Sub: D

o propósito de D é uma especialização do propósito de C

```

void m( p1, ..., pn ) r1, ..., rk
@pre  pD( P )
@post rD( R )
    
```

antecedente pode ser mais restrita do que consequente  $\left[ \begin{array}{l} pC(P) \Rightarrow pD(P) \\ rD(R) \Rightarrow rC(R) \end{array} \right]$  consequente não pode ser mais restrita do que antecedente

Mai 2015

Arndt von Staa © LES/DI/PUC-Rio

28

## Exemplos de programas instrumentados



- String no formato: <dimString><length><corpo>\0\?

```
struct String
{
    int dimString ;
    int length ;
    char corpo[ dimString ] ; ← essa sintaxe vale?
    char fim[2] ;
} /* end string */

EXC_ASSERT( length >= 0 ) ;
EXC_ASSERT( length <= dimString ) ;
EXC_ASSERT( memcmp( fim , "\0\?" , 2 ) == 0 ) ;
```

## Exemplo C++ string de dimensão qualquer



```
struct stringControlado
{
    int dimString ;
    int length ;
    char * pCorpo ;

    stringControlado( int dim )
    {
        dimString = dim ;
        length = 0 ;
        pCorpo = new char[ dimString + 2 ] ;
        memset( pCorpo , 0 , dimString ) ;
        memcpy( pCorpo + dimString , "\0?" , 2 ) ;
    }

    ~stringControlado( )
    {
        delete pCorpo ;
    }
} ; /* struct */

stringControlado s1( 5 ) ;
stringControlado s2( 100 ) ;
```

## Exemplo de verificador estrutural (parcial)



```
// Verify max and num known segments
ASSERT_VER( maxKnownSegments >= 0 , 1 ) ;
ASSERT_VER( maxKnownSegments <= sizVtSegments , 2 ) ;
ASSERT_VER( numKnownSegments >= 0 , 3 ) ;
ASSERT_VER( numKnownSegments <= maxKnownSegments , 4 ) ;
// Verify pointers beyond maxKnownSegments
for( i = maxKnownSegments ; i < sizVtSegments ; i++ )
{
    ASSERT_VER( vtSegments[ i ] == NULL , 5 ) ;
} /* for */
// Validate segments and count number of known segments
countSegments = 0 ;
for( i = 0 ; i < maxKnownSegments ; i++ )
{
    if ( vtSegments[ i ] != NULL )
    {
        ASSERT_VER( vtSegments[ i ]->VerifySegment( modeParm ) , 6 ) ;
        countSegments ++ ;
    } /* if */
} /* for */
ASSERT_VER( countSegments == numKnownSegments , 7 ) ;
```

ASSERT\_VER recebe dois parâmetros: a condição a ser controlada, e um identificador numérico usado para explicitar a causa da falha sem que se precise ler o código. Os id's podem ser declarados em uma enumeração, evitando o uso de "números mágicos".

## Bibliografia



- Gries, D.; *The Science of Programming* ; New York: Springer; 1981
- Karaorman, M.; Hölzle, U.; Bruno, J.; *jContractor: A Reflective Java Library to Support Design by Contract*, [www.cs.ucsb.edu/TRs/TRCS98-31.html](http://www.cs.ucsb.edu/TRs/TRCS98-31.html)
- Kramer, R.; *iContract*, [www.reliable-systems.com](http://www.reliable-systems.com)
- Meyer, B.; *Applying Design by Contract*; *IEEE Computer* 25(10); 1992; pages 40-51
- Meyer, B.; *Object-Oriented Software Construction*, 2nd edition, New Jersey: Prentice Hall; 1997
- Mitchell, R.; *Design by contract: Bringing together formal methods and software design*; University of Brighton; 2004
- Pezzè, M.; Young, M.; *Teste e Análise de Software*; Porto Alegre, RS: Bookman; 2008
- Staa, A.v.; *Programação Modular* ; Rio de Janeiro: Campus; 2000
- Staa, A.v. *The Talisman C++ Unit Testing Framework*; Monografias em Ciência da Computação 01/12; Departamento de Informática, PUC-Rio; 2012



Laboratório de Engenharia de Software

LES

FIM

Mai 2015

Arndt von Staa © LES/DI/PUC-Rio

33