




Laboratório de Engenharia de Software

Teste Estrutural 4

Arndt von Staa
Departamento de Informática
PUC-Rio
Maio 2015



Laboratório de Engenharia de Software


Especificação

- Objetivo desse módulo
 - apresentar e discutir o critério de teste baseado em mutantes
 - o teste baseado em assertivas como oráculos e adição de redundância visando aumentar a capacidade de detecção
- Justificativa
 - é possível utilizar defeitos inseridos (mutantes) como instrumento de avaliação da confiabilidade de uma massa de testes.
 - quando se utiliza especificações formais, torna-se possível utilizar as assertivas de entrada e saída como instrumentos de verificação (oráculos) da corretude dos resultados
 - através da judiciosa adição de redundâncias e das correspondentes assertivas pode-se aumentar a capacidade de detecção de erros, habilitando o software a conviver com erros de variadas procedências sem necessitar formular testes

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

Laboratório de Engenharia de Software

Teste com mutantes



- Como saber qual é a eficácia de uma suíte de teste?
- Eficácia: num defeitos observados / número de defeitos existentes
- Qual é o problema inerente a essa definição?


Mai 2015

Arndt von Staa © LES/DI/PUC-Rio

3

Laboratório de Engenharia de Software

Teste com mutantes



- Processo de teste com mutantes, em linhas gerais:
 - desenvolve-se uma suíte de testes para o módulo a testar
 - criam-se diversas versões modificadas do módulo a testar
 - cada **versão contém um (ou poucos) defeitos** inseridos deliberadamente no módulo a testar original
 - cada defeito elementar inserido é um **mutante** do módulo a testar
 - executam-se os testes **para cada um** dos mutantes
 - examinam-se os resultados do teste
 - deveriam ser detectadas falhas decorrentes dos defeitos introduzidos pelos mutantes
 - mutantes encontrados pelos testes são **mutantes mortos**
 - mutantes que permanecem vivos precisam ser examinados, pois podem ser **equivalentes**, ou corresponder a **caminhos impossíveis ou inviáveis**
 - pode ser necessário adicionar mais casos à suíte para matar os mutantes ainda vivos possíveis e viáveis
 - possivelmente serão detectadas falhas originais

Adaptado de (Delamaro et al, 2007)

Mai 2015

Arndt von Staa © LES/DI/PUC-Rio

4

Teste com mutantes



- Seja f a **especificação da função** a ser implementada. Seja p a **implementação** de f . Seja D o **domínio dos dados** aceitos por f . Seja M um **conjunto de mutantes** de p . Uma suíte de teste T é **adequada** para p relativamente a M sse

$$\forall m \in M : \text{se } (\exists d \in D \mid m(d) \neq f(d)) \\ \text{então } (\exists t \in T \mid m(t) \neq f(t))$$

Podem existir vários $d \in D$ e zero ou mais $t \in T$ similares segundo o mutante detectado por d .

- para cada mutante não equivalente existe pelo menos um caso de teste que acusa a correspondente falha
- exceções:
 - mutantes equivalentes** – o resultado do mutante é igual ao original, ou seja são mutantes equivalentes: $\forall d \in D \mid m(d) == f(d)$ ex.
 - mutantes podem encontrar-se em **código não alcançável**
 - a modificação injetada pelo mutante não corresponde a um defeito, ex.:

original:

`int i = 0 ; código sem i ; i = fx () ;`

alterado:

`int i = -1 ; código sem i ; i = fx () ;`

Teste com mutantes



- Em virtude de cada mutante m de p ser **conhecido** é possível determinar um ou mais $t \in D$ tal que $m(t) \neq f(t)$ para cada mutante específico
 - o rigor do teste depende agora da escolha de M
 - M precisa ser **representativo** dos possíveis defeitos
 - para simplificar a discussão M é restrito aos **mutantes relevantes**, i.e. não considera os mutantes equivalentes
 - a **qualidade do teste** (1 é "perfeito"):

$$\text{Mutantes_Mortos} / \text{Total_Mutantes não equivalentes}$$
 - a **qualidade do artefato** (1 é "perfeito"):

$$\text{Mutantes_Mortos} / (\text{Defeitos_Originais} + \text{Total_Mutantes})$$
 - se M **não for representativo**, i.e. contém poucos mutantes relevantes, essas métricas produzem indicações confiáveis?
 - provavelmente não

Teste com mutantes



- **Operadores de mutação (deturpadores)** aplicados ao módulo (função) sob teste efetuam a reescrita introduzindo um defeito, assegurando que o código resultante continue sintaticamente correto
- Exemplos:
 - eliminação de comandos
 - troca de operador relacional
 - exemplo: substitua < por <=, ==, !=, >= e >
 - troca de operadores aritméticos
 - troca de variáveis com mesmo tipo
 - troca de valores iniciais
 - troca de valores de constantes
 - ...

Mai 2015

Arndt von Staa © LES/DI/PUC-Rio

7

Uma possível forma de criar mutantes



Código original

```
if ( NomeDir[ strlen( NomeDir ) - 1 ] == '\\' )
{
    NomeDir[ strlen( NomeDir ) - 1 ] = 0 ;
}
```

Código com mutantes

```
bool temp ;
switch ( idMutante )
{
    case 1: temp = ( NomeDir[ strlen( NomeDir ) - 1 ] == '/' ) ; break ;
            /* mutação: '\\' :: '/' */
    default: temp = ( NomeDir[ strlen( NomeDir ) - 1 ] == '\\' ) ; break ;
}
if ( temp )
{
    switch ( idMutante )
    {
        case 2: NomeDir[ strlen( NomeDir ) ] = 0 ; break ;
                /* mutação: "- 1" :: "" */
        default: NomeDir[ strlen( NomeDir ) - 1 ] = 0 ; break ;
    }
}
```


Mai 2015

Arndt von Staa © LES/DI/PUC-Rio

8

Laboratório de Engenharia de Software

Teste com mutantes




- Hipóteses
 - **programador competente**: programas incorretos distinguem-se de programas corretos em virtude de poucos **defeitos simples** (defeitos locais), ex. $x < y$ ao invés de $x \leq y$
 - consequência: um número finito de mutantes (defeitos simples) é capaz de avaliar a eficácia da suíte de teste
 - mas quais são os operadores de mutação?
 - casos de teste capazes de detectar defeitos simples também **são capazes de detectar defeitos complexos** (ou seja, são compostos por vários simples espalhados)
 - consequência: assume-se que **defeitos originais** possivelmente complexos possam ser detectados por testes que visem detectar defeitos simples
 - a **distribuição de defeitos é uniforme**
 - isso é uma hipótese altamente duvidosa

Mai 2015
Arndt von Staa © LES/DI/PUC-Rio
9

Laboratório de Engenharia de Software

Teste com mutantes: problemas




- A quantidade de mutantes a ser gerada para criar um conjunto de mutantes representativo tende a ser grande
 - custo de geração dos mutantes
 - custo de compilação
 - pode-se instrumentar o código – por intermédio de ferramentas – inserindo switches em que uma seleção é o código original e as demais são mutantes, através de parâmetros de execução escolhe-se qual opção a executar
 - caso se use interpretadores o custo de compilação diminui sensivelmente
 - mas tem-se uma implementação interpretável e outra compilável, possivelmente sutilmente diferentes entre si
 - custo de realização dos testes
 - todos os mutantes precisam ser exercitados
- Precisa-se de boas ferramentas para injetar mutantes e conduzir o processo de instrumentação e teste

Mai 2015
Arndt von Staa © LES/DI/PUC-Rio
10

Laboratório de Engenharia de Software

Teste com mutantes: problemas




- O esforço gasto para determinar o porquê de mutantes permanecerem vivos pode ser grande
 - quais dos mutantes vivos são mutantes equivalentes?
 - que novos casos de teste testariam o programa de modo que o mutante seja observado?
- Existe um ramo de pesquisa – *search based software engineering* – que procura encontrar esses casos de teste utilizando abordagens de inteligência artificial
- Existem plugins para Java: MuJava, MuClique e outras

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

Laboratório de Engenharia de Software

Teste com mutantes: outros usos



- O teste de mutantes pode ser utilizado para avaliar
 - métodos de geração automática de casos de teste
 - criam-se módulos ou componentes, em princípio corretos, que servirão de **termos de avaliação do gerador**
 - padrões de comparação
 - criam-se mutantes para esses módulos
 - realizam-se os testes com uso do gerador de casos de teste
 - a qualidade dos testes observada é um indicador da **confiabilidade do gerador**
 - esta abordagem pode ser estendida
 - a padrões de *design* de testes
 - a testes baseados em máquinas de estado
 - ...

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

Teste com mutantes: outros usos




- O teste de mutantes pode ser utilizado para avaliar
 - instrumentos de **monitoração da integridade de estruturas de dados** → **verificadores estruturais**
 - criam-se estruturas de dados em princípio corretas,
 - criam-se mutantes destas estruturas
 - a capacidade dos verificadores estruturais de identificar os defeitos introduzidos é um indicador da **eficácia da instrumentação**
 - pode-se adotar uma forma de avaliar similar para assertivas de entrada e de saída
 - forçam-se violações dos contratos dos métodos e verifica-se se a instrumentação neles contida evidencia essas violações



Teste com assertivas

Teste com assertivas




Laboratório de Engenharia de Software

- Problema da **perfeição**
 - raras vezes programas são perfeitos, falhas podem ter causas
 - **exógenas** – provocadas por outros artefatos
 - exemplos: falha de hardware, rede, infra-estrutura, bibliotecas
 - usuários podem fazer uso incorreto do artefato
 - » ex. **dado** que não satisfaz o contrato, ou que é inacurado
 - » ex. **comando** que não satisfaz o **protocolo** de uso
 - existência de vulnerabilidades que, se exercitadas de forma desastrosa, geram erros
 - » no uso normal, vulnerabilidades nunca geram erros
 - » exemplos: divisão por zero, falta de memória
 - consequência: mesmo programas “sem defeitos” podem falhar
 - **endógenas** – provocadas por defeitos ainda desconhecidos
 - é muito baixa a probabilidade de um programa não conter defeitos
 - segundo Boehm e Basili menos de 50% dos módulos de um programa são isentos de defeitos

Mai 2015
Arndt von Staa © LES/DI/PUC-Rio
15

Teste com assertivas




Laboratório de Engenharia de Software

- Problema da **imprevisibilidade**
 - não é possível saber de antemão as possíveis causas de falhas
 - se soubéssemos, poderíamos impedir que existam, ou pelo menos avisar quanto ao risco de mau funcionamento
 - falhas durante o uso raramente fornecem dados que permitam diagnosticar a causa – identificar o defeito – desta falha
 - a diagnose sem dados do contexto da falha em geral é bastante trabalhosa
 - uma forma de fornecer dados de apoio à diagnose baseia-se na geração de logs possuindo informação relativa ao estado da execução e de variáveis selecionadas

Mai 2015
Arndt von Staa © LES/DI/PUC-Rio
16

Laboratório de Engenharia de Software

Teste com assertivas




- Problema da **imprevisibilidade**
 - consequência: é necessário ser capaz de identificar erros – i.e. determinar a ocorrência de uma falha – o mais **próximo** possível do defeito ou do fator exógeno causador do erro
 - o que se entende por **próximo**:
 - a **proximidade** é medida pelo **número de instruções executadas** desde o momento da geração do erro até a sua detecção
 - consequentemente
 - » a proximidade deve ser pequena
 - » a **latência** da geração do erro até a sua detecção deve ser curto
 - » além disso o **estado do sistema** no momento do erro deve sofrer poucas alterações até o momento de observação da falha
 - em geral a **distância léxica** (medida do local no código em que é gerado o erro ao local em que é observado) não possui qualquer correlação com a proximidade

Mai 2015
Arndt von Staa © LES/DI/PUC-Rio
17

Laboratório de Engenharia de Software

Teste com assertivas




- Problema do **custo da monitoração**
 - a avaliação de assertivas custa, mesmo que pouco
 - se a frequência da avaliação é alta – **granularidade** pequena – o custo extra total pode ser alto
 - entretanto, é facilitada a diagnose das falhas
 - se a frequência de avaliação é baixa – granularidade grande – o custo extra total tende a ser baixo
 - entretanto, a diagnose das falhas torna-se mais difícil
- Sugestão
 - usar níveis de granularidade padronizados para controlar a compilação condicional


```
#if defined( _DEBUG ) && _GRANULARITY < 5
```

Mai 2015
Arndt von Staa © LES/DI/PUC-Rio
18

Laboratório de Engenharia de Software

Teste com assertivas




- Caso se deseje software com **baixa incidência de falhas danosas durante o uso**, precisamos
 - ser capazes de detectar erros antes que produzam danos expressivos
 - tratar as falhas de modo que os danos permaneçam dentro de limites aceitáveis
- Problema
 - como fazer isso se não conhecemos as causas das potenciais falhas?
- Uma possível solução
 - tornar os programas auto verificáveis (*self-checking*)
 - requer a introdução de redundâncias no código

Mai 2015
Arndt von Staa © LES/DI/PUC-Rio
19

Laboratório de Engenharia de Software

Necessidade de redundância




- Qualquer técnica de controle da qualidade **requer alguma forma de redundância**, exemplos
 - análise estática
 - resultado esperado calculado externamente ao artefato sob teste comparado com o resultado obtido a partir da execução do artefato
 - *capture / replay*: o comportamento gerado a partir da execução (*replay*) de um script deve ser idêntico ao comportamento gerado pelo artefato quando estava sendo usado em modo de registro do script (*capture*)
 - uso de operação inversa
 - programação em pares; verificação; inspeção
 - ...

Mai 2015
Arndt von Staa © LES/DI/PUC-Rio
20

Laboratório de Engenharia de Software

Necessidade de redundância




- Qual é a redundância envolvida no teste baseado em assertivas?
 - as assertivas executáveis correspondem a uma verificação formal
 - possivelmente incompleta
 - para redigir uma assertiva executável é necessária uma especificação *suficientemente formal*
 - a existência de uma especificação formal, mesmo que incompleta, induz uma **redundância de raciocínio** ao desenvolver o artefato
 - contribui para reduzir o número de defeitos inseridos durante o desenvolvimento
 - mais próximo do ideal correto por construção
 - menos retrabalho inútil

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

Laboratório de Engenharia de Software

Adição de redundância




- Vou assumir que estamos usando uma linguagem de programação que permita operações com ponteiros
 - por exemplo: C, C++, assembler
 - ou ao usar armazenamento codificado usando índices
 - para outras linguagens precisa-se fazer adaptações, em geral são simplificações

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

Laboratório de Engenharia de Software

Adição de redundância

- Vou assumir que estamos usando uma linguagem de programação que permita operações com ponteiros
 - por exemplo: C, C++, assembler
 - ou ao usar armazenamento codificado usando índices
 - para outras linguagens precisa-se fazer adaptações, em geral são simplificações




Mai 2015
Arndt von Staa © LES/DI/PUC-Rio
23

Laboratório de Engenharia de Software

Adição de redundância, exemplos

- Função de inserção em lista duplamente encadeada
`InserirElementoApos(pElemAntes , pNovoElem)`
 - assertiva de entrada (contrato)
 - `pNovoElem != NULL ;`
 - `pElemAntes != NULL ;`
 - `pElemAntes->pCabeca != NULL ;`
 - `pElemAntes->pProx != NULL ?`
`pElemAntes->pProx->pAnt == pElemAntes : TRUE ;`
 - assertiva estrutural
 - `pElemAntes != NULL ;` necessários para assegurar o contrato do verificador
 - `pElemAntes->pCabeca != NULL ;`
 - `pElemAntes->pCabeca->VerificarLista() ;`



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

Adição de redundância, processo



Seja *Ref* uma referência para um espaço qualquer

- **Passo 1:** Deve ser possível determinar o **tipo do espaço** referenciado por *Ref*, ou seja, deve ser possível realizar o **controle dinâmico de tipos**
 - todos os espaços iniciam com n caracteres que correspondem ao **identificador do tipo do espaço**: `idTipoXXX` → τ
 - isso permite criar o objeto compatível com o conteúdo da memória

Fábrica de objetos baseada no tipo do dado



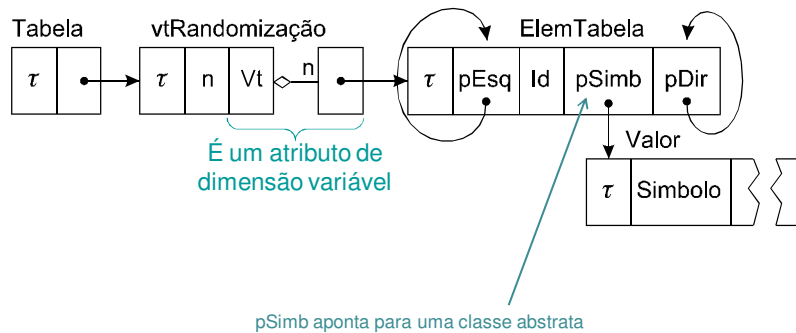
```
// Category: Page - uma família de herança forma um categoria
// Number of page types
static const int numTypes = 3 ;
// Type table array
static TPH_tpTypeDescriptor vtTypes[ numTypes ] = {
    { PAGE_SimplePage      , TPH_IdTypeIllegal , "\xBD\x00" , "Simple page" } ,
    { PAGE_SegmentRootPage , PAGE_SimplePage  , "\xBD\x01" , "Segment root page" } ,
    { PAGE_FreeListPage    , PAGE_SimplePage  , "\xBD\x02" , "Free list page" }
} ; /* page type table */

// Page object factory
PG_Page * PAGE_Construct( int idType ) {
    switch( idType ) {
        case PAGE_SimplePage      : return new PG_Page( ) ;
        case PAGE_SegmentRootPage : return new PG_SegmentRootPage( ) ;
        case PAGE_FreeListPage    : return new PG_FreePage( ) ;
    } // switch
    EXC_FAIL( ) ;
} // End of function: CPAGE &Construct PAGE objects
```

Adição de redundância, processo



- **Passo 2:** Deve ser possível determinar a extensão do elemento apontado por *Ref*



Mai 2015

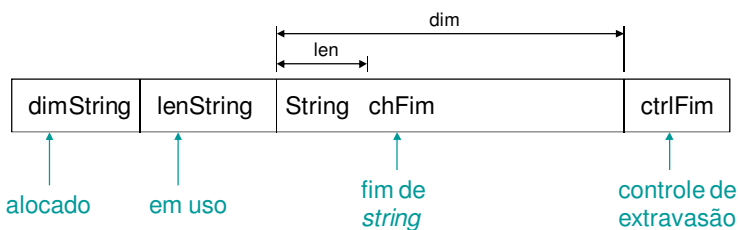
Arndt von Staa © LES/DI/PUC-Rio

27

Adição de redundância, processo



- **Passo 3:** se o espaço pode alocar **dados de tamanho variável**, deve ser possível determinar o **tamanho alocado**
- **Passo 4:** se o **tamanho em uso** é variável, deve ser possível determinar o tamanho em uso e verificar se ocorreu **extravaseamento do espaço**
 - Para *strings* isso equivale a dizer que deveriam ser armazenados de uma forma similar a:



Mai 2015

Arndt von Staa © LES/DI/PUC-Rio

28

Adição de redundância, processo

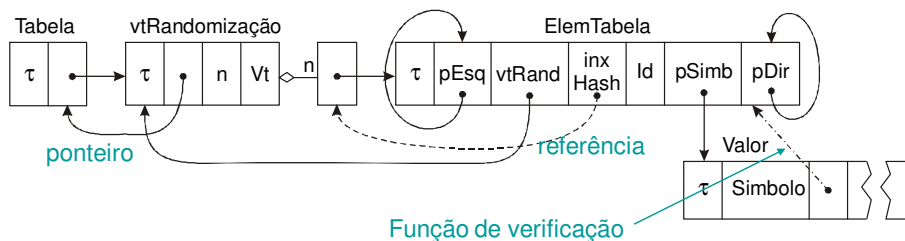


- **Passo 5:** deve ser possível **verificar completamente** o conteúdo do espaço apontado por *Ref*
 - deve existir um **verificador** para cada um dos tipos de espaço conhecidos
 - se os tipos forma uma estrutura de herança, pode-se utilizar *down cast* (C++: `dynamic_cast`)
 - se o tipo destino de uma referência for conhecido, deve ser verificado se o tipo implícito da referência é consistente com o tipo dinâmico do espaço referenciado
 - se estiver correto pode-se ativar diretamente o verificador correspondente ao tipo destino
 - se o tipo destino de uma referência não for conhecido (e.g. `void *`; ou `object`) deve ser ativado o verificador destino através de um **distribuidor de chamadas**

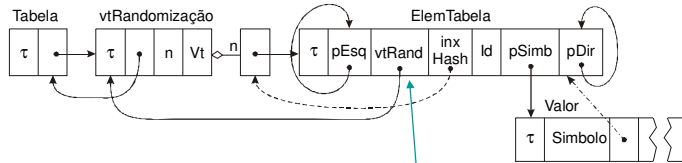
Adição de redundância, processo



- **Passo 6:** deve ser possível verificar completamente todas as referências a **espaços adjacentes** ao espaço referenciado por *Ref*
 - para tal é necessário dispor de uma **referência inversa**
 - pode ser um elemento de um conjunto
 - ex. antecessores e sucessores de um vértice de um grafo
 - pode requerer uma **função de verificação** especializada
 - ex. quando o tipo destino for derivado de classe abstrata



Adição de redundância, processo



- Passo 7:** redija e verifique as assertivas estruturais

$pTabela \rightarrow idTipo == idTipoTabSymb$

$pTabela \rightarrow vtRand \rightarrow idTipo == idTipoTabRand$

$pTabela \rightarrow vtRand \rightarrow pTabela == pTabela$

$pTabela \rightarrow vtRand \rightarrow n \geq 2$

$\forall inxElem \mid 0 \leq inxElem < pTabela \rightarrow vtRand \rightarrow n :$

$pTabela \rightarrow vtRand[inxElem] \rightarrow idTipo == idTipoElem$

$VerificarLista(pTabela \rightarrow vtRand[inxElem] , idTipoElem)$

$\forall elem \in Lista(pTabela \rightarrow vtRand[inxElem]) :$

$pElem \rightarrow vtRand == pTabela \rightarrow vtRand$

$pElem \rightarrow inxHash == inxElem$

$pElem \rightarrow inxHash == ObterHash(ObterSimbolo(pElem \rightarrow pSymb))$

$pElem \rightarrow pSymb \rightarrow idTipo == idTipoVal$

$VerificarSymb(pElem \rightarrow pSymb , pElem)$

Para que seja possível verificar os espaços acessíveis pelos elementos da lista é necessário o ponteiro para a tabela

Para assegurar que o símbolo não tenha sido alterado deve calcular o *hash* do símbolo referenciado e comparar com *inxHash*, ou com *inxElem*

Como o tipo símbolo não é conhecido, necessita-se de uma função de verificação para a referência inversa

Mai 2015

Arndt von Staa © LES/DI/PUC-Rio

31

Exemplo: macro de apoio



```
#define  ASSERT_VER( Condition , idMsg )      \
if ( !( Condition ) )                       \
{                                             \
    EXC_LOG( envelope.pMsg , idMsg ) ;      \
    errorCount ++ ;                         \
}
```

Mai 2015

Arndt von Staa © LES/DI/PUC-Rio

32

Exemplo: verificador estrutural de lista



```
bool SLS_SimpleList ::  
    Verify( const TAL_tpVerifyMode modeParm )  
{  
    int hasCurrent    = 0 ;  
    int errorCount    = 0 ;  
    int countElement = 0 ;  
    SLS_ListElement * pElem = NULL ;  
  
    struct PointerEnvelope {  
        MSG_CMessage * pMsg ;  
        PointerEnvelope( ){ pMsg = NULL ; } ;  
        ~PointerEnvelope( ){ delete pMsg ; pMsg = NULL ; } ;  
    } envelope ; /* struct */  
    envelope.pMsg = new MSG_CMessage( SLS_ErrorRoot ) ;
```

id da mensagem de log padrão.
Contém um parâmetro para
informar a falha observada

Exemplo: verificador estrutural de lista



```
if ( ( pFirstElement == NULL )  
    && ( pLastElement == NULL )  
    && ( pCurrentElement == NULL ) ) return true ;  
  
if ( ( pFirstElement == NULL )  
    || ( pLastElement == NULL )  
    || ( pCurrentElement == NULL ) )  
{  
    EXC_LOG( envelope.pMsg , ErrorEmptyList ) ;  
    return false ;  
} /* if */  
  
pElem = pFirstElement ;  
ASSERT_VER( pElem->pPredecessor == NULL , ErrorFirst )
```

identificam a falha
observada

Exemplo: verificador estrutural de lista



```
while ( pElem != NULL ) {
    countElement ++ ;
    ASSERT_VER( pElem->pValue != NULL , ErrorMissingValue )
    if ( pElem->pValue != NULL ) {
        ASSERT_VER( pElem->pValue->VerifyElement( modeParm ) ,
            ErrorValue )
        if ( pElem == pCurrentElement ) hasCurrent ++ ;
        if ( pElem->pPredecessor != NULL )
            ASSERT_VER( pElem->pPredecessor->pSuccessor == pElem ,
                ErrorPredecessor )
        if ( pElem->pSuccessor != NULL )
            ASSERT_VER( pElem->pSuccessor->pPredecessor == pElem ,
                ErrorSuccessor )
        pElem = pElem->pSuccessor ;
    } // end repetition: Verify body of the list
    ASSERT_VER( pLastElement->pSuccessor == NULL , ErrorLast )
    ASSERT_VER( hasCurrent == 1 , ErrorCurrent )
    ASSERT_VER( numElements == countElement , ErrorCount )
    return errorCount == 0 ;
} // End of function: SLS !Verify list
```

Referências bibliográficas



- Alexander, R.T.; Bieman, J.M.; Ghosh, S.; Ji, B.; "Mutation of Java Objects"; *ISSRE'02 - 13 th International Symposium on Software Reliability Engineering*; IEEE; 2002; pags 1-11
- Delamaro, M.E.; Maldonado, J.C.; Jino, M.; *Introdução ao Teste de Software*; Rio de Janeiro, RJ: Elsevier / Campus; 2007
- Staa, A.v.; *Programação Modular*; Rio de Janeiro: Elsevier / Campus; 2000
- Staa, A.v.; *The Talisman C++ Unit Testing Framework*; Monografias em Ciência da Computação, No. 01/12; Rio de Janeiro: Departamento de Informática, PUC-Rio; 2012

Laboratório de Engenharia de Software

LES

FIM

Mai 2015

Arndt von Staa © LES/DI/PUC-Rio

37