




Laboratório de Engenharia de Software

Teste Automatizado 1

Arndt von Staa
Departamento de Informática
PUC-Rio
Maio 2015

Especificação



Laboratório de Engenharia de Software

- Objetivo desse módulo
 - motivar o uso de testes automatizados.
 - apresentar algumas abordagens para a automação dos testes de unidade.
- Justificativa
 - Se realizada de forma manual
 - é uma atividade cara e sujeita a muitos enganos
 - dificulta atestar a qualidade do artefato sob teste
 - Uma forma de reduzir os custos e prover meios para fornecer um atestado de qualidade é a realização de testes automatizados

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

Laboratório de Engenharia de Software

Algumas classes de ferramentas de apoio aos testes

- apoio à gestão dos testes
- verificadores de especificações
- verificadores de código
 - verificadores estáticos
 - verificadores dinâmicos
 - identificadores de anomalias
 - medidores dinâmicos
- teste de desempenho
 - teste de capacidade
 - teste de limite
 - teste de exaustão
- teste de aplicações web
 - links quebrados
 - simuladores de variedades de browsers
 - simuladores de variedades de equipamentos
- teste de unidade
- teste de banco de dados
- teste de integração
 - integração contínua
- comparadores de arquivos
 - diff textual, diff estrutural
- capturadores de janelas
- capturadores de eventos
- instrumentação
 - pré-processadores
- geradores de casos de teste
 - totais ou parciais
- geradores de *drivers* de teste
- máquinas virtuais
- . . .

LES

Mai 2015
Arndt von Staa © LES/DI/PUC-Rio
3

Laboratório de Engenharia de Software

Recordação: teste de módulo

- **Testes são necessários** mesmo quando há preocupação com o desenvolvimento correto por construção
 - inspeções e provas da corretude são falíveis (Yelowitz and Gerhart, 1976)
 - o uso de componentes incorretos resultará em um sistema contendo defeitos
 - muitas bibliotecas de classes não são confiáveis (Thomas, 2002)
 - existem coisas que são melhor observadas através de experimentos envolvendo humanos do que a partir de especificações
 - interface com o usuário (Boehm, 1984)
 - satisfação do usuário
 - . . .

• Boehm, B.W.; Gray, T.E.; Seewaldt, T.; "Prototyping versus specifying: A multiproject experiment"; *IEEE Transactions on Software Engineering* SE-10(5); Los Alamitos, CA: IEEE Computer Society; 1984; pages 290-303
 • Thomas, D.; "The Deplorable State of Class Libraries"; *Journal of Object Technology* 1(1); ETH Zürich; 2002; pages 21-27
 • Gerhart, S.L.; Yelowitz, L.; "Observations of fallibility in applications of modern programming methodologies"; *IEEE Transactions on Software Engineering* 2(9); Los Alamitos, CA: IEEE Computer Society; 1976; pages 195-207

LES

Mai 2015
Arndt von Staa © LES/DI/PUC-Rio
4

Recordação: teste de módulo



- Recordação: teste de módulo
 - **cenários de teste**
 - descreve como deve estar **configurado** o contexto utilizado ao testar
 - descreve a natureza dos defeitos sendo procurados
 - **massa de teste** → conjunto de **casos de teste**, executados em uma única submissão de teste
 - **suíte de teste** → conjunto de massas de teste
 - **roteiro de teste** → descrição detalhada de como **pessoas** devem proceder para realizar os casos de teste
 - **script de teste** → "código" do teste automatizado
 - para cada caso de teste da suíte
 - definir comandos e dados a serem utilizados
 - estabelecer como será verificado o resultado do teste
 - qual será o oráculo a ser utilizado
 - quais são os critérios de aceitação

Recordação: teste manual



- Ao testar manualmente, o testador deve, **para cada caso de teste**:
 - assegurar que o contexto para o teste está estabelecido
 - selecionar um a um os comandos que formam o caso de teste
 - fornecer os dados requeridos para cada um desses comandos
 - **repetir a digitação** em caso de erro de dado
 - verificar se os resultados obtidos **são coerentes** com o que é esperado
 - no teste manual o oráculo é usualmente verificação visual
 - caso os resultados não satisfaçam o oráculo
 - **registrar os problemas observados**
 - **registrar as condições** em que ocorreram os problemas
- **Sempre que o artefato for alterado:**
 - **repetir o procedimento para todos os casos de teste**

Recordação: teste manual



- O controle visual é sujeito a muitos enganos por parte do testador
 - o testador **entra com dados ou comandos errados** e conclui:
 - encontrei uma falha → **falso positivo**
 - o testador **acha** que o resultado obtido está OK sem efetivamente conferir com o devido cuidado se isto é fato
 - não encontrei falha, embora exista evidência → **falso negativo**
 - teste baseado em **achologia** é convite ao desastre
 - o testador **acha** que o resultado obtido é suficientemente aproximado ao esperado, quando na realidade não é
 - **cansaço** contribui para aumentar a taxa de enganos por parte do testador
 - digitar errado e não observar os erros → não registrar as falhas
 - o testador **não anota** nem a falha nem as condições em que ocorreu
 - leva a retrabalho inútil, ou software com defeito remanescente
 - ...

Teste manual



**Teste manual
difícilmente pode ser utilizado
como atestado de qualidade!**

Por que ?

- Excessiva confiança na **infallibilidade** e na **disciplina** do testador
 - torna necessário um significativo esforço de controle da realização dos testes
- A busca por contenção de custos do teste manual muitas vezes leva a testes realizados de forma parcial

Recordação: teste manual



- Vantagens do teste manual
 - relativamente simples e barato de programar
 - muitas vezes nem requer programação
 - viabiliza o teste exploratório
 - teste sem um roteiro fixo
 - visa entender o serviço prestado pelo artefato
 - pode basear-se em especificações intuitivas ou tácitas...
 - virtualmente irrestrito
 - facilita a verificação de imagens, figuras e leiaute de janelas
 - o oráculo pode basear-se em
 - resultados aproximados ou insinuados
 - assume-se que o testador saiba identificar as discrepâncias com relação ao resultado esperado
 - verificações difíceis ou impossíveis de mecanizar
 - interface humano computador, usabilidade
 - aparência, beleza?

Teste automatizado




- Desenvolvimento incremental e manutenção frequente requerem a **realização repetida** de:
 - teste a cada **incremento**
 - cada incremento é testado de forma cuidadosa
 - procurar atingir elevado nível de **qualidade por desenvolvimento**
 - cada módulo é testado individualmente
 - **expectativa**: a integração de módulos corretos leva a construtos (quase) corretos
 - teste de **integração**
 - cada construto é testado com o **rigor viável**
 - ênfase nas interfaces entre módulos e componentes
 - teste de **regressão** de módulo e de construto
 - a cada manutenção deve-se ser capaz de mostrar que o que não foi alterado continua operando tal como vinha operando antes das alterações

manutenção: evolução, adaptação, correção, preventiva

Laboratório de Engenharia de Software

Teste automatizado

- Para assegurar qualidade ao manter precisa-se
 - assegurar que os testes tenham sido **completa e corretamente** reaplicados **a cada alteração**
 - laudo / log caso a caso do teste
 - dispor de evidência de que o teste foi realizado com a profundidade satisfatória
 - **atestado de qualidade *segundo* a suíte de teste *utilizada***
 - poder **inspecionar os casos de teste** que compõem a suíte
 - possibilidade de **auditar o atestado de qualidade**
 - tornar o teste **independente do testador**
 - eliminar o fator de falibilidade humana ao realizar o teste
 - assegurar que o custo do reteste seja baixo
 - ...




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

Laboratório de Engenharia de Software

Teste automatizado

- O teste automatizado pode gerar um **log do teste** (laudo de teste) na console ou em um arquivo
 - documenta o fato de ter executado toda a suíte de teste
 - documenta todas as falhas observadas
 - provê informação relativa ao contexto do caso de teste
 - funções `assertXXX()` ou `compararXXX()` geram log
 - documenta os casos testes que foram realizados
 - permite a análise de cada caso de teste executado
 - facilita a análise da abrangência e do rigor dos testes
 - ou seja, viabiliza auditar a qualidade do teste



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

Motivação para automação



- Mesmo que imperfeito, o

Teste automatizado pode ser utilizado como um atestado da qualidade.

Por que?

- determina de forma inequívoca **como a qualidade foi controlada**
 - a auditoria do teste permite identificar imperfeições na forma de realizar o controle da qualidade
 - permite identificar o grau da qualidade assegurada
 - mesmo que seja baixo
 - a melhoria do script de teste **assegura** o aumento do rigor do controle da qualidade

Mai 2015

Arndt von Staa © LES/DI/PUC-Rio

13

Comparação manual vs. automatizado



confirmada na prática, ou seja, é quase sempre um fato

Hipótese: o teste automatizado é mais barato do que o teste manual

Condição	Manual	Automatizado
custo da formulação do roteiro de teste		mais alto, porém o teste tende a ser mais rigoroso
custo da co-evolução dos artefatos relacionados com a automação dos testes		possivelmente mais alto: mais artefatos a evoluir; mais difícil realizar
custo do reteste	bem mais alto → trabalho humano	
custo da garantia da qualidade	mais alto → mais esforço de controle	
custo das falhas endógenas pós-entrega	mais alto → menos rigor ao testar, logo mais defeitos remanescentes	
custo do teste para fins de diagnose	bem mais alto → trabalho humano	baixo: o teste reproduz a falha enquanto não for removida a causa (defeito)

Mai 2015

Arndt von Staa © LES/DI/PUC-Rio

14

Breve história



- Teste automatizado tem sido usado desde o início dos tempos ☺
 - sempre soluções ad hoc, exemplos
 - sistemas de teste sob medida para determinado artefato
 - geração de dados aleatórios com o emprego de funções inversas como oráculos
 - geração de dados aleatórios, ou uso em ambiente controlado, empregando assertivas estruturais como oráculos
 - possivelmente usando alguma forma de multi-threading
 - ...
 - eu usei a primeira vez em 1963 ao desenvolver uma biblioteca para o processamento numérico de matrizes, exemplo

$$M \times M^{-1} \cong I$$

Problemas com o teste automatizado



- Se a **suíte for superficial**
 - o teste automatizado pode tornar-se pouco eficaz
 - vantagem: execução rápida do teste
 - desvantagens:
 - limitada a encontrar poucos dos defeitos existentes
 - baixa eficácia

Problemas com o teste automatizado



- A construção de **suítes segundo um critério cuidadoso** pode requerer muito esforço
 - o **custo de criação** da suíte pode exceder o custo de **uma ocorrência de teste manual**, considerando os custos do testador humano
 - o **grau de rigor** da suíte pode ser ajustado ao risco de uso do artefato, possivelmente reduzindo o esforço de criação
 - porém o **custo do reteste** é muito baixo
 - se o desenvolvimento for incremental, retestes serão frequentes
 - se a expectativa de vida do software for longa, retestes serão frequentes
 - se o artefato passar por algumas manutenções, retestes abrangentes deveriam ser realizados sempre
- Portanto, com teste automatizado o **custo total** acaba sendo **bem** mais baixo


CrITÉRIOS de avaliação dos scripts de teste



- Os critérios de valoração foram considerados?
 - em particular as condições de contorno
- Caso existam, as relações inversas foram exploradas nos testes?
 - isso pode ser generalizado para redundâncias judiciosamente inseridas no código
- Caso existam, formas de calcular alternativas foram utilizadas como oráculos?
 - trata-se aqui de redundância da forma de calcular
- Foram forçadas as ocorrências das condições de erro?
 - foram exercitados todos os **throw**, condições de retorno e mensagens de erro?
- As características de desempenho estão dentro dos limites esperados?

Laboratório de Engenharia de Software

Critérios de avaliação dos scripts de teste




- Os valores estão em conformidade com o formato especificado (ou sub-entendido)?
- Os valores estão ordenados (ou não) conforme especificado?
- O domínio dos valores (mínimo e máximo) são respeitados?
 - o domínio de valores plausíveis é controlado?
- O código depende de condições que estão fora do controle do testador?
 - objetos externos utilizados precisam estar em determinado estado?
 - o usuário precisa estar autorizado?

Adaptado de: [Hunt & Thomas, 2004]

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

Laboratório de Engenharia de Software

Critérios de avaliação dos scripts de teste



- A cardinalidade (número de objetos de um conjunto) ou as contagens estão corretas?
- A temporização é respeitada?
 - os eventos ocorrem na ordem correta?
 - e satisfazem as restrições de tempo especificadas?

Adaptado de: [Hunt & Thomas, 2004]

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

Critérios de avaliação dos scripts de teste



- Todos os atributos e referências dos objetos alterados devem ser verificados
 - pode tornar necessária a introdução de redundâncias
 - ver teste baseado em assertivas

```

== Test emptying list
=emptylist    list0
=getnumelem   list0 0
=movetofirst  list0 0
=movetolast   list0 0
=moveelem     list0 -1 0
=moveelem     list0 1 0
=getelem      list0 "."
=deleteelem   list0 listwasempty

== Insert an element into an empty list
=insertafter  list0 "abcdefg"
=getnumelem   list0 1
=moveelem     list0 -1 0
=moveelem     list0 1 0
=getelem      0 "abcdefg"
    
```

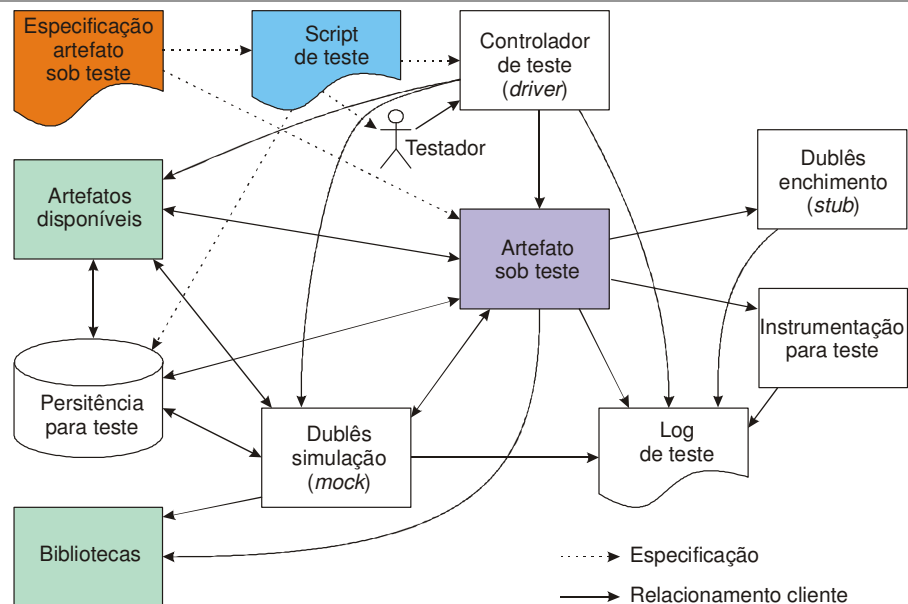
"." corresponde a null element

Mai 2015

Arndt von Staa © LES/DI/PUC-Rio

21

Esquema da organização para teste



Mai 2015

Arndt von Staa © LES/DI/PUC-Rio

22

JUnit: Exemplo, uma classe bem simples



```
public class Counter
{
    int count = 0;

    public int increment( )
    {
        return count += 1;
    }

    public int decrement( )
    {
        return count -= 1;
    }

    public int getCount()
    {
        return count;
    }
}
```

- Isso precisa mesmo ser testado?
- Classe para a qual não existe uma suíte de teste completa, não tem evidência de estar correta.

Exemplo extraído de Matuszek 05-java; University of Pennsylvania

JUnit: Um teste para a classe bem simples



```
public class CounterTest
{
    Counter counter1; // o objeto a ser testado
    @Before           // é sempre executado antes de cada caso
    public void setUp( )
    {
        counter1 = new Counter( ); // inicializa
    }
    @Test             // Sinaliza que o método a seguir é um caso de teste
    public void testIncrement( )
    {
        assertTrue(counter1.getCount( ) == 0 );
        assertTrue(counter1.increment( ) == 1 );
        assertTrue(counter1.increment( ) == 2 );
    }
    @Test
    public void testDecrement( )
    {
        assertTrue(counter1.decrement( ) == -1 );
    }
}
```

Note que o `setUp` é executado antes de testar esse método

JUnit: O método equals



- A asserção `assertEquals(esperado, calculado)` compara utilizando o operador `==` ou o método `x.equals(y)`
- Atributos com tipos primitivos podem ser comparados com `==`
- Algumas classes Java implementam o método `x.equals(y)` para comparar objetos `x` e `y` de uma mesma classe, ex.
 - String
- As demais classes devem implementar

```
public boolean equals(Object obj) { ... }
```

 - Exemplo, na classe `Simbolo`

```
public boolean equals( Object simbolo )
{
    Simbolo s = ( Simbolo ) simbolo ;
    return this.valor == s.valor ;
}
```
 - o parâmetro é sempre do tipo `Object` e precisa ter o seu tipo imposto (*down cast*) para o tipo de comparação, para então poder ser comparado

Mai 2015

Arndt von Staa © LES/DI/PUC-Rio

25

JUnit: assertivas de teste



- `assertEquals([mensagem] , esperado , obtido)`
- `assertEquals([mensagem] , esperado , obtido , tolerância)`
 - funciona desde que ou *esperado* e *obtido* forem do mesmo tipo primitivo, ou o método `equals` está definido no tipo *esperado*
- `assertNull([mensagem] , refObjeto)`
- `assertNotNull([mensagem] , refObjeto)`
- `assertSame([mensagem] , refObjetoEsp , refObjetoObt)`
- `assertNotSame([mensagem] , refObjEsp , refObjObt)`
- `assertTrue([mensagem] , expressão booleana)`
- `assertFalse([mensagem] , expressão booleana)`
- `assertFail([mensagem])`
- `[mensagem]` – a mensagem é opcional

Mai 2015

Arndt von Staa © LES/DI/PUC-Rio

26

JUnit: Inicialização e término



@Before

```
public void setUp( ) { ... }
```

- efetua a inicialização antes de cada @Test

@After

```
public void tearDown( ) { ... }
```

- limpa o que tiver que ser limpo após cada @Test

@BeforeClass

```
public static void setUpClass( ) { ... }
```

- efetua a inicialização antes de iniciar a execução da massa

@After

```
public static void tearDown( ) { ... }
```

- limpa o que tiver que ser limpo após a execução da massa

JUnit: Argumentos para @Test



- Controle de tempo, por exemplo para interceptar loops infinitos

```
@Test ( timeout = 10 )           10 milisegundos
public void nuncaTermina( )
{
    assertTrue( algum.metodoNaoPara( ) == 100 ) ;
}
```


- Receber uma exceção esperada
 - O teste falha se a exceção não for recebida

```
@Test ( expected = IllegalArgumentException.class )
public void factorial( )
{
    program.factorial(-5);
}
```

Laboratório de Engenharia de Software

xUnit: vantagens

- É possível gerar uma sequência de código
 - tão extensa quanto for necessário
 - obedecendo rigorosamente a um ou mais critérios de seleção de casos de teste
- É possível utilizar os controles da própria linguagem de programação para
 - utilizar estruturas de controle tipo *if then else*, *while*, *for*, etc.
 - criar funções que realizem sequências repetitivas
 - criar funções de comparação especializadas
- Reduz a barreira decorrente da necessidade de aprender uma nova linguagem
- Existe uma grande variedade de xUnits: C++, C#, Http, Lua, ...




Mai 2015
Arndt von Staa © LES/DI/PUC-Rio
29

Laboratório de Engenharia de Software

xUnit: desvantagens


- JUnit usa reflexão – impede que se possa realizar testes no contexto de módulos cliente já aprovados
 - para alguns isso é uma vantagem → força o teste isolado de cada módulo
- O arcabouço de teste precisa conter o “main”
- A função de teste tende a
 - ser extensa ou particionada em um número grande de pequenas funções
 - ser difícil para o **usuário** ler e entender
 - dificulta a análise da abrangência e do rigor do teste
 - ser mal documentada
 - dificulta a co-evolução
 - induzir o uso de critérios de seleção de casos de teste do gênero adivinhação de defeitos (“*defect guessing*”)



Mai 2015
Arndt von Staa © LES/DI/PUC-Rio
30

Laboratório de Engenharia de Software

xUnit: desvantagens




- Pode-se **atenuar** as desvantagens através de
 - disciplina ao redigir o módulo de teste específico
 - fortemente recomendado
 - exigir o mesmo rigor e detalhe da documentação para os módulos de teste que os usados para os módulos de produção
 - seleção cuidadosa e documentação do critério de seleção de casos de teste utilizado
 - gerar o módulo de teste
 - existem ferramentas **que dizem** que fazem isso
 - quebrar a suíte de teste em várias massas, cada uma com um propósito bem definido

Mai 2015Arndt von Staa © LES/DI/PUC-Rio31

Laboratório de Engenharia de Software

DBUnit: teste para bases de dados




- Permite verificar a consistência do modelo de dados
- Facilita a carga do banco de dados antes dos testes
- Oferece vários métodos de comparação para tabelas e arquivos temporários de extração
- Segue as ideias inerentes ao JUnit

Mai 2015Arndt von Staa © LES/DI/PUC-Rio32

Laboratório de Engenharia de Software

DBUnit: Ciclo de Vida 1/2




- Limpar o banco de dados
- Carregar no banco de dados os dados projetados para o teste
- Executar os testes
 - comparar os resultados obtidos utilizando os métodos do DBUnit

Mai 2015Arndt von Staa © LES/DI/PUC-Rio33

Laboratório de Engenharia de Software

DBUnit: Ciclo de Vida 2/2



- As operações limpar e carregar ocorrem durante a execução do método JUnit `setUp()`
 - embora isso possa consumir tempo de processamento, a ideia é que o `setUp` somente carregue os dados para um teste específico, ao invés dos dados requeridos para todos os testes
 - `setUpClass()` permite criar um banco de dados para todos os casos de teste do módulo específico de teste
 - porém: se um método falhar o banco de dados poderá estar comprometido ao realizar os casos subsequentes
- Assegura que o banco contém exatamente o que foi projetado para o teste

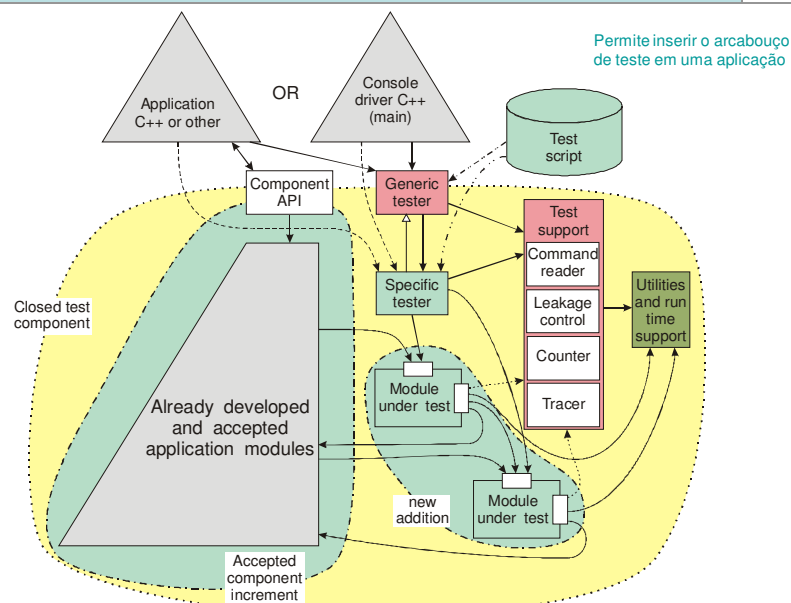
Mai 2015Arndt von Staa © LES/DI/PUC-Rio34

DBUnit: observações finais



- DBUnit
 - é uma extensão do JUnit
 - primeiro aprenda a utilizar JUnit
 - é uma ferramenta poderosa e simples de usar, permitindo o teste de aplicações com bancos de dados SQL
 - faz parte de uma estratégia eficaz de testes
 - conduz a código estável
 - aumenta significativamente a confiança da equipe de desenvolvimento

TalTest: Arcabouço de teste Talisman-5



TalTest: controle da suíte



```
<Program>
  ProgramName = tst-bcd
  <TestScript>
    TestScriptName = tst-bcd-01
  </TestScript>
</Program>
```

Script interpretado por um programa Lua

```
<Program>
  ProgramName = tst-dsp
  <TestScript>
    TestScriptName = tst-dsp-01
    ExpectedErrors = 1
    ExpectedReturn = 6
  </TestScript>
  <TestScript>
    TestScriptName = tst-dsp-02
  </TestScript>
  <TestScript>
    TestScriptName = tst-dsp-03
    ExpectedReturn = 6
  </TestScript>
</Program>
```

TalTest: casos de teste




```
== Add two numbers of length 6
=ConvertASCIIToBCD      0 6 "-8072377"
=ConvertASCIIToBCD      1 6 "1293877"
=Add                    0 1
=GetBCDNumber           0 "E00006778500" hex E = bin 1110 → - len = 6
                        1 2 3 4 5 6

== Add two numbers of length 6
=ConvertASCIIToBCD      0 6 "-8071234"
=ConvertASCIIToBCD      1 6 "1291234"
=Add                    0 1
=GetBCDNumber           0 "E00006780000"

== Add numbers of different size
=ConvertIntToBCD         0 4 -3785328
=ConvertIntToBCD         1 5 108303031
=Add                    0 1
=ExceptionProgram        0 ErrorLossOfData 'c'
```

Laboratório de Engenharia de Software

TalTest: log do teste



```

34 == Add two numbers of length 6
40 == Add two numbers of length 6
46 == Add numbers of different size with loss of data


>>> 1   Line 49  Tester caught a program exception >>
      ERROR: Loss of data while assigning a BCD number.
      Exception thrown in Line: 536  File: ..\sources\bcdarit.cpp
<<< 0   Line 50  Expected exception has been ignored.

52 == Add numbers of different size
...
  
```

Mai 2015
Arndt von Staa © LES/DI/PUC-Rio
39

Laboratório de Engenharia de Software

TalTest: exemplo interpretador



```

// Test: BCD  &Add BCD number B to BCD number A
// Command: Add  <i inxBcd A> <i inxBcd B>

else if ( strcmp( Command , BCD_Add_023_CMD ) == 0 )
{
    /*****
        Function:  char * BCD_Add( char * BCDNumberA ,
                                const char * const BCDNumberB )
    *****/
    int  inxReg_A = -1 ;
    int  inxReg_B = -1 ;
    int  numRead = TST_pReader->ReadCommandLine( "ii" ,
                                                &inxReg_A , &inxReg_B ) ;

    if ( ( numRead != 2 )
        || !VerifyInxElem( inxReg_A , YES )
        || !VerifyInxElem( inxReg_B , YES ) )
    {
        return TST_RetCodeParmError ;
    } /* if */

    vtObj[ inxReg_A ] = BCD_Add( vtObj[ inxReg_A ] , vtObj[ inxReg_B ] ) ;

    return TST_RetCodeOK ;

} // end selection: Test: BCD  &Add BCD number B to BCD number A
  
```

} Leitura dos dados

} Validação dos dados lidos

Operação a testar


Controle de retorno da execução

Mai 2015
Arndt von Staa © LES/DI/PUC-Rio
40

Laboratório de Engenharia de Software

TalTest: vantagens

- Visa C++ e o subset C contido em C++
- O arcabouço de teste não precisa ser a origem do sistema
- Através da linguagem de diretivas de teste pode-se realizar testes detalhados
 - casos de teste selecionados a partir de critérios bem definidos
 - o script de teste pode apoiar a geração de dados aleatórios
- Facilita inspecionar a abrangência e o rigor do teste
 - a linguagem de script de teste leva a suítes densas
 - os casos de teste estão necessariamente confinados à interface do(s) módulo(s)
 - os casos de teste podem realizar comandos relativos a operações de outros módulos
 - pode-se testar módulos não orientados a objetos




Mai 2015
Arndt von Staa © LES/DI/PUC-Rio
41

Laboratório de Engenharia de Software

TalTest: vantagens

- Os logs gerados
 - documentam os laudos dos testes realizados
 - necessário para que outros possam manter o script de teste
 - permitem estabelecer com precisão (ex. linha do script de teste) quando e onde o *debugger* deve ser ativado
 - permitem inspecionar se o teste foi realizado completamente segundo o critério de teste requerido




Mai 2015
Arndt von Staa © LES/DI/PUC-Rio
42

Laboratório de Engenharia de Software

TalTest: desvantagens

- É necessário escrever um interpretador
- O arquivo de diretivas é na realidade um programa
 - mais uma linguagem a aprender
 - mesmo que seja muito simples
 - pode conter defeitos
 - se não se tomar cuidado, a linguagem *script* de teste pode tornar-se complicada
- Ao encontrar um problema é necessário determinar se é causado por defeito contido
 - no módulo sob teste
 - no *script* de teste
 - no módulo de teste específico – i.e. no interpretador
 - no arcabouço de teste




Mai 2015
Arndt von Staa © LES/DI/PUC-Rio
43

Laboratório de Engenharia de Software

Bibliografia

- Beck, K., *Extreme Programming Explained: Embrace Change*; Addison-Wesley; 1999
- Fewster, M.; Graham, D.; *Software Test Automation*; Addison-Wesley; 1999
- Hunt, A.; Thomas, D.; *Pragmatic Unit Test: in Java with JUnit*; Raleigh, North Carolina: The Pragmatic Bookshelf; 2004
- Staa, A.v.; *AutoTest - Arcabouço para a Automação dos Testes de Módulos Redigidos em C*; site www.inf.puc-rio.br/~inf1301
- 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
<http://www-di.inf.puc-rio.br/~arndt/Talisman-5>
- Wake, W.C.; *Extreme Programming Explored*; Addison-Wesley; 2001
- DbUnit: <http://www.dbunit.org/>
- JUnit: <http://www.junit.org/>



Mai 2015
Arndt von Staa © LES/DI/PUC-Rio
44

Laboratório de Engenharia de Software

LES

FIM

Mai 2015

Arndt von Staa © LES/DI/PUC-Rio

45