
INFO1120 – TCP – Slides – Arquivo 5

Técnicas de Construção de Programas

Prof. Marcelo Soares Pimenta
mpimenta@inf.ufrgs.br

Porto Alegre, agosto a dezembro de 2011

Testes do Desenvolvedor

- Testes e Qualidade de Software
- Erros (falhas) e software
- Testes:
 - Objetivos
 - Observações
 - Visão Geral
 - Dimensões de teste e fundamentos

Teste e Qualidade de Software

- Princípio Geral da Qualidade:
 - “Melhorar a qualidade REDUZ os custos de desenvolvimento”
- Quanto mais cedo um erro é detectado, mais fácil e menos custoso é corrigi-lo
 - Erros nos requisitos SÃO mais caros do que erros de projeto e de implementação
- Objetivo do Teste:
 - Detectar a presença (existência) de erros
- Objetivo da Depuração:
 - Localizar e Corrigir o erro
- Testes, em si, NÃO melhoram a qualidade.
 - Resultados dos testes são INDICADORES da qualidade
 - Se quiser melhorar seu software, deve-se encontrar uma forma melhor de atingi-la

Erros (falhas) e Software

- Primeira premissa: ERRO é fato **cotidiano** e não excepcional
- 3 formas de tratar erros (falhas) de software:
 1. **Evitar falhas** (*fault-avoidance*): especificação, projeto, implementação e manutenção usando métodos sistemáticos e confiáveis , tipicamente baseadas em métodos formais e reuso de componentes de software altamente confiáveis; -> praticável somente para sistemas altamente críticos
 2. **Eliminar falhas** (*fault-elimination*): análise, detecção e correção de erros cometidos durante a desenvolvimento . **Aqui incluem-se as atividades verificação, validação e teste.**
 3. **Tolerar falhas** (*fault-tolerance*): compensação em tempo real de problemas residuais como mudanças fora da especificação no ambiente operacional, erros de usuário, etc. Geralmente lida com recursos de hardware e/ou software adicionais ou redundantes;
- Devido ao fato de *fault-avoidance* ser economicamente impraticável para a maioria das empresas, e de *fault-tolerance* exigir muitos recursos a tempo de execução, a técnica de eliminação de falhas (*fault-elimination*) geralmente é a adotada pelos desenvolvedores de software.

Objetivo dos Testes

- Objetivo:
 - Detectar a presença (existência) de erros
- Suposição incorreta 1: ‘Mostrar **ausência** de erros’
 - Se presumir assim, provavelmente não encontrará
- Suposição incorreta 2: ‘**Assegurar** que o programa funciona corretamente’
- Verificação (‘We Build it right ?’):
 - O que foi especificado e projetado foi construído corretamente?
 - Checado em relação à especificação e aos requisitos
- Validação (‘We Build the right thing ?’):
 - Construimos o que era certo ?
 - Checado pelo cliente/usuário

Sessão Depoimentos

- Discussão:
 - Alguém já participou de alguma estratégia de testes em alguma instituição/empresa? Qual tipo de teste foi adotado no(s) seu(s) ambiente(s) de trabalho?

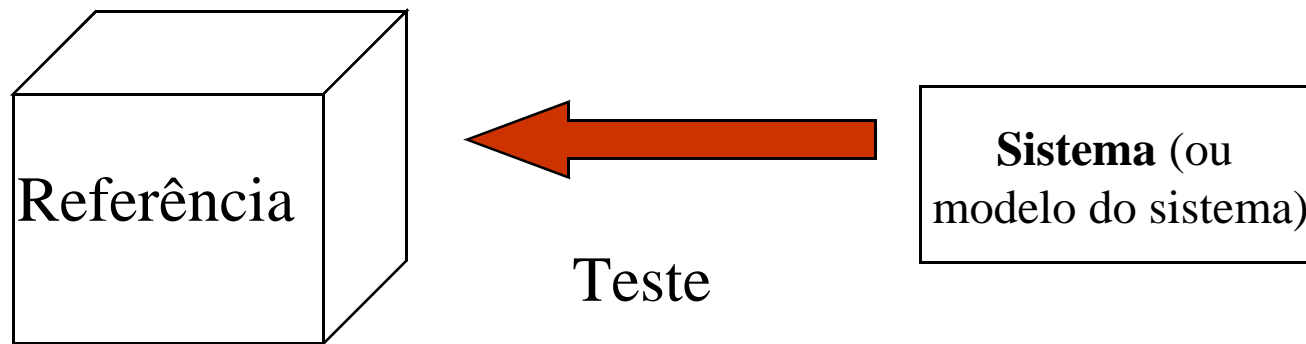
Testes no Desenvolvimento de software

- Testes iterativos e contínuos possibilitam uma medida objetiva do status do projeto
- Inconsistências nos requisitos, projeto e implementação são detectadas mais cedo
- A responsabilidade pela qualidade de trabalho da equipe é melhor distribuída ao longo do ciclo de vida
- Os envolvidos (*stakeholders*) no projeto podem ter evidências concretas do andamento do projeto
- Testes – PAs Verification & Validation para nível 3 CMMI

Testes no Desenvolvimento de SW

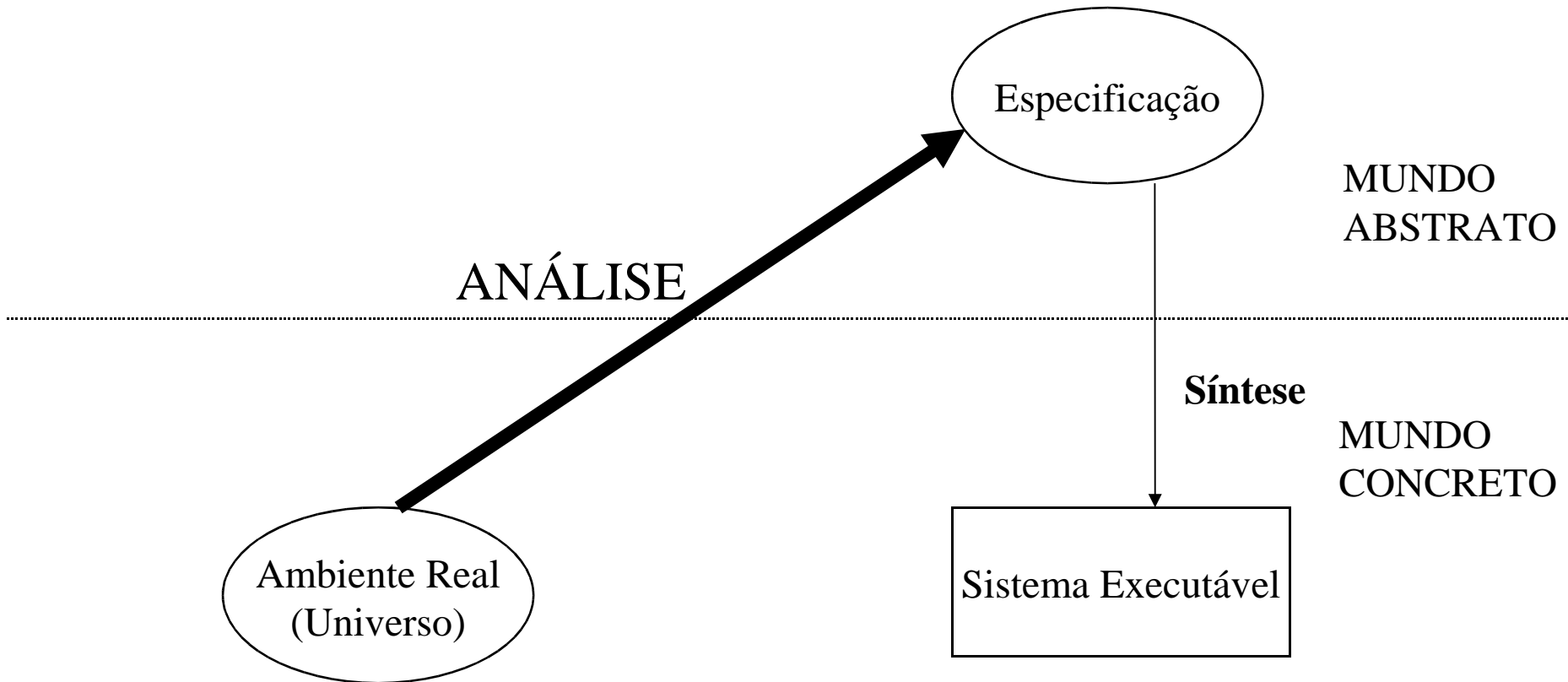
- Processo efetivo de teste é solução para alguns problemas do desenvolvimento:
 - Medida do status do projeto é objetiva porque os testes reportados são avaliados
 - Esta medida objetiva expõe inconsistências nos requisitos, projeto e implementação
 - Testes e verificações estão focadas nas áreas de maior risco, aumentando a qualidade destas áreas.
 - Defeitos são descobertos antes, reduzindo o custo de seu conserto
 - Ferramentas automatizadas de teste provêm testes para funcionalidade e performance.
 - Teste é a única técnica de validação para requisitos não-funcionais

Teste x Análise

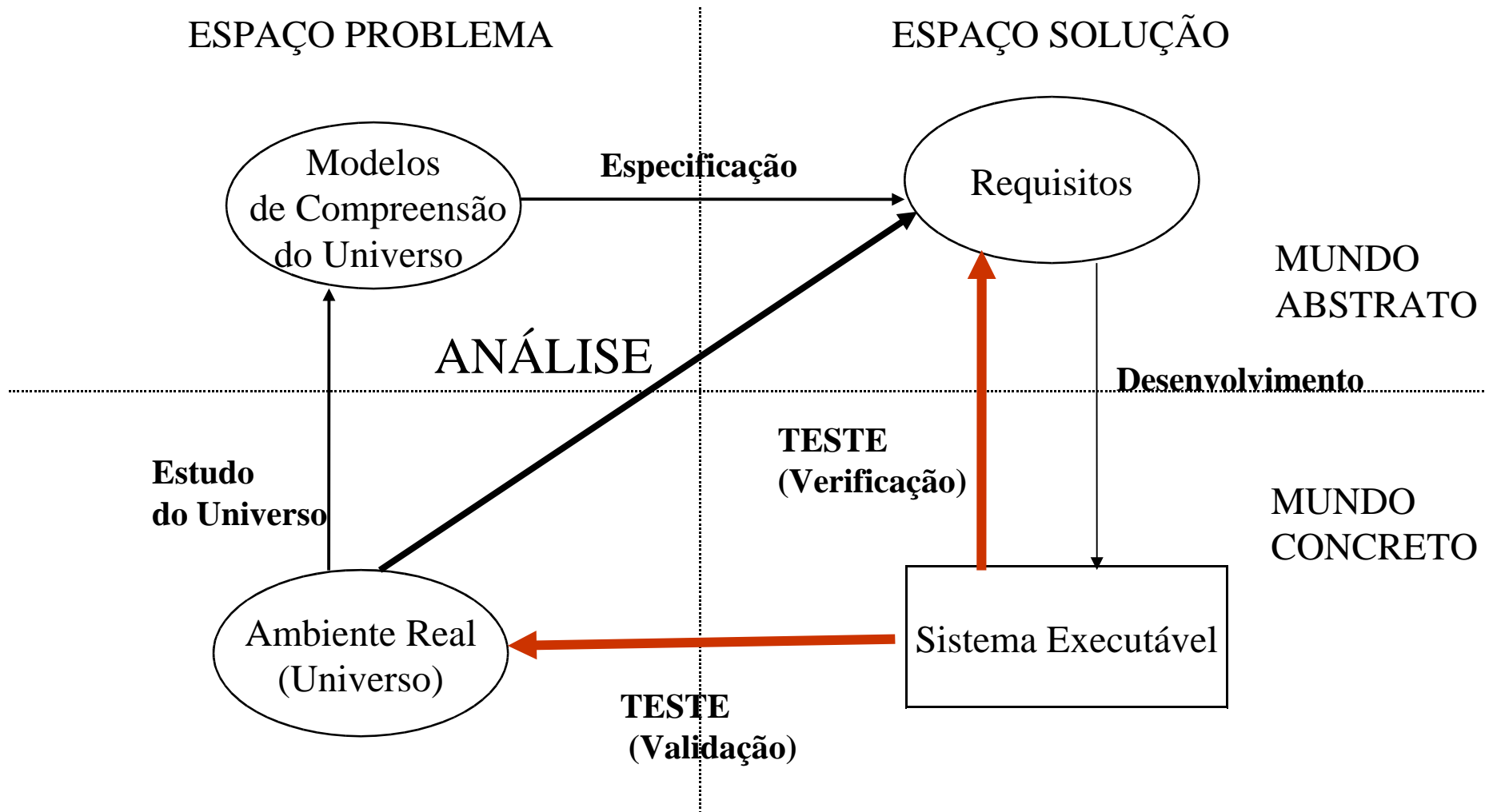


- Modelo explícito
- Idéias, expectativas (modelo implícito)
- Procedimentos manuais realizados
- Sistema atual existente
- Especificação de Requisitos

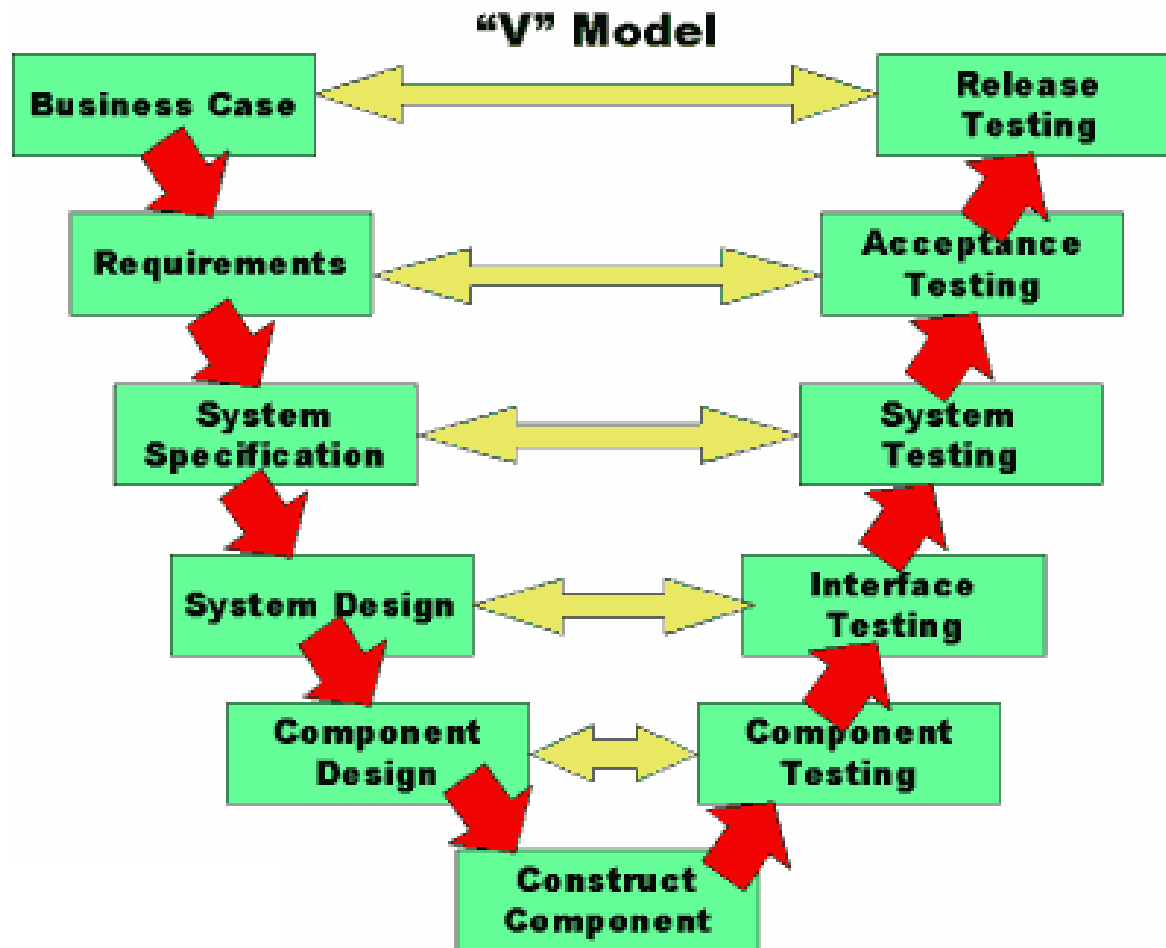
Teste x Análise



Teste x Análise



Priorizando Teste: o modelo 'V'



Observações sobre Testes

- Testes em conflito com outras atividades do desenvolvimento de software?
 - Objetivo:
 - Detectar a presença (existência) de erros
 - Teste bem sucedido é aquele em que se detectam falhas;
 - Testador como “advogado do diabo” da equipe:
 - É melhor a equipe detectar falhas do que o usuário!!
 - Todas outras atividades visam EVITAR erros e impedir a falha do software
- Testes NUNCA garantem a ausência de erros;
 - Testes exaustivos são inviáveis na prática: testar cada valor de entrada concebível para o programa, bem como cada combinação concebível dos valores de entrada.
 - Ex. Programa que recebe Nome (20 char) , Endereço (20 char) e Num (10 dígitos) de Telefone e os armazenasse em arquivo.
 - Nome 26^{20} (20 char, 26 letras no alfabeto)
 - Endereço 26^{20} (20 char, 26 letras no alfabeto)
 - Fone 10^{10} (10 algarismos, 10 escolhas)

Total de possibilidades = $26^{20} * 26^{20} * 10^{10} = 10^{66}$

Visão Geral de Testes (1/2)

- Princípios de teste:
 - Teste é uma fase planejada DURANTE o desenvolvimento do sistema e NÃO após;
 - Teste é **responsabilidade de todos na equipe**:
 - Teste deve ser planejado pelo analista e projetista ;
 - **Teste de unidade (MÓDULO) deve ser realizado pelo desenvolvedor**;
 - Todos outros testes podem ser realizados por outra pessoa (evitar indução) -> **Equipe de teste**
 - Os melhores testes são aqueles que apontam o maior número de erros diferentes com o menor número de experimentos;
 - Fim dos testes é ditado geralmente por argumentos econômicos;
 - É melhor ter em mente que os **erros são inevitáveis** e você é que tem dificuldade de encontrá-los

Visão Geral de Testes (2/2)

- O que fazer com os resultados dos testes?
 - i. Avaliar a confiabilidade do produto em desenvolvimento
 - % de erros
 - ii. Avaliar o progresso do desenvolvimento
 - % do sistema já está funcionando
 - iii. GUIAR as correções dos bugs
 - “Bug trackers”
 - iv. Criar uma memória de testes (“os erros mais comuns”)
 - Maior cuidado nos próximos projetos
 - Direcionar treinamento e formação à equipe
 - Direcionar atividades de revisão técnica

3 dicas de Técnicas de design de testes

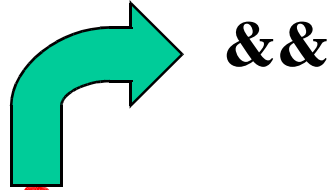
- Projeto de teste:
 - O que faz um bom teste ? Quanto teste se deve fazer?
- Dica 1:
 - Preste atenção a seu catálogo de erros
 - Erros se repetem - frequentemente

Dica 1 – faça um catálogo de erros

- Pessoas não são muito imaginativas quando cometem erros :
 - Tendem a cometer SEMPRE OS MESMOS erros de antes
- Projeto de testes é quase sempre o ato de verificar os erros historicamente plausíveis e deve verificar se o código faz tudo que era intencionado – catálogo é um bom início
- Dica 1: Crie um catálogo – sua memória de erros – e preste atenção a ele na hora de projetar testes
- Ex de catalogo <http://www.testing.com/writings/catalog.pdf>.
- Ex. de conteúdo de catálogo:
 - DESVIOS de fluxo – IFs, SWITCHs, etc
 - Estruturas recursivas
 - Ponteiros, etc
 - Todos tipos de entradas devem ser manipuladas diferentemente
 - Booleanos, inteiros , strings, ponteiros ou referências , etc
- CATÁLOGO tem idéias de teste , não obrigatoriamente TUDO vira teste em outra ocasião

Dica 2

- Erros prováveis: Expressões Booleanas

 **&&**

```
if (publicClear || technicianClear) {  
    bomb.detonate();  
}
```

Acharam o erro??

Dica 2 (cont)

- **Teste1: Condição “public clear” (P) E “technician clear”(T)**
 $P = V, T = V, P \parallel T = V, P \ \&\& \ T = V$
- **Teste2: Condição “técnico junto à bomba” e “tem público”**
 $P = F, T = F, P \parallel T = F, P \ \&\& \ T = F$
- **Teste3: Condição “public clear” E “técnico junto à bomba”**
 $P = V, T = F, \textcolor{red}{P} \parallel \textcolor{red}{T} = \textcolor{red}{V}, \textcolor{red}{P} \ \&\& \ \textcolor{red}{T} = \textcolor{red}{F}$

Dica 2 (cont)

- **Expressões booleanas mal formuladas**

- **A||B&&C ?**

- Poderia ser:

- **(A||B)&&C**
- **A&&B&&C**
- **A||B&& !C**
- **A|| !B&&C**
- **A||B ||C**
- **A|| !(B&&C)**
- **!A||B&&C**
- Ou mesmo **A||D&&C**

A dica consiste em tentar achar causas de erros prováveis na formulação destas expressões...Pode não achar erros mas aumenta a atenção sobre o uso deste tipo de expressão.



Test ideas for (a || (b && c)):

a	b	c
-----	-----	-----
True	true	FALSE
FALSE	true	true
FALSE	FALSE	true
FALSE	true	FALSE

Ver <http://www.testing.com/tools.html>

Dica 2 (cont)

- **ATENÇÃO !!!:**
- **Expressões somente com &&**
 $A1 \ \&\& \ A2 \ \&\& \ ... \ \&\& \ A_n$

- **têm $N+1$ casos:**

–all conditions true

– $A1$ FALSE, all others true

– $A2$ FALSE, all others true

–...

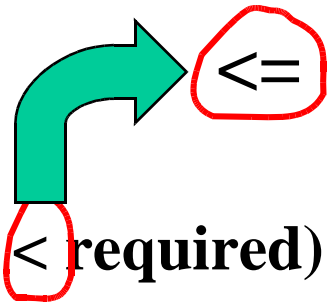
– A_n FALSE, all others true

$A \ \&\& \ B$

A	B
V	V
F	T
T	F

Dica 3

- Erros prováveis:
Expressões Relacionais
(>, <, etc)



- ...if (finished < required)
{
- siren.sound();
- }...

Achando o erro:

finished	required	$F < R$	$F \leq R$
0	100	V	V
5	5	F	T

Dica 3(cont)

- Sugestão de testes:

Se expr $A < B$ ou $A \geq B$:

- $A = B - e$
- $A = B$

No caso da expressão

$finished < required$

Se expr $A > B$ ou $A \leq B$

- $A = B$
- $A = B + e$

Deve have ao menos 2 testes:

- Se A e B inteiros, $e=1$
- Se A e B reais, $e < 1$

- 7) FINISHED um pouco menor que REQUIRED: V
- 8) FINISHED exatamente igual a REQUIRED: F

Resumo das dicas

- Dicas servem para auxiliar criatividade e não substituí-la
- Catálogo como gerador de idéias
- Converse com colegas e troque catálogos e idéias
- Tenha foco no usuário – erros aparecem a eles

Teste - Exercício

- Como você testaria o programa P com a seguinte definição?
 - Seja P um programa que leia três inteiros que corresponderiam aos lados de um triângulo (A,B,C), verifica se é efetivamente um triângulo, e em caso positivo classifica-o em triângulo escaleno, isósceles e equilátero
 - Formar grupos de 2 a 3 alunos, iniciar trabalho em aula, terminar em casa, entregar na próxima aula (via email ou em papel)

Requisitos

- Requisitos são o PONTO DE PARTIDA para qualquer processo de software !!!!
 - Tradicional, prototipação, incrementais, orientado a modelos (MDA), etc
- Tipos de Requisitos
 - Requisitos Funcionais : associados a funcionalidade
 - Requisitos Não Funcionais: associados à propriedades (critérios de qualidade) e restrições

Requisitos x Teste

- Uma clara compreensão dos requisitos é fundamental para qualquer atividade de teste, verificação ou validação
- Como checar algo se não se sabe precisamente o que deveria ser feito?

Dimensões de Testes

- Dimensão de qualidade
 - Os atributos de qualidade que estão sendo o foco do teste
 - P.ex. : Tipos de teste - Teste de funcionalidade, Teste de segurança, Teste de Carga (ou Teste de Stress), Teste de Desempenho (ou Teste de performance), Teste de Usabilidade
- Categoria do teste
 - Técnica de teste usada
 - P.ex. Teste estático vs Teste dinâmico
- Estágio do teste
 - O ponto dentro do ciclo de desenvolvimento em que o teste está sendo executado
 - P.ex. Níveis de teste: Teste de Unidade, Teste de Integração, Teste de Sistema, Teste de Aceitação (Homologação)
- Método de teste
 - O objetivo específico do teste individual
 - P.ex. “caixa preta” vs “caixa branca”

Testes Estáticos vs Testes Dinâmicos

- Categorias de Teste (em relação à execução)
 - Estático: checagem do código SEM execução
 - Inspeção (leitura cruzada em equipe com argumentação)
 - Walkthrough ('teste de mesa')
 - Análise estática de propriedades do código com ferramentas :
 - verificação FORMAL de tipos, estruturas de controle, métricas de complexidade
 - Dinâmico: checagem do código a partir de sua execução sobre conjuntos de dados (casos de teste)
 - Como escolher casos de teste?
 - Como decidir se um resultado é correto ou não?
 - Quando decidir parar os testes?
 - Geralmente o MAIS ADOTADO é o Dinâmico

Métodos de Teste

- **Teste Funcional (caixa preta)**
 - Visa verificar funcionalidade baseado apenas nos dados de entrada e saída.
- **Teste Estrutural (caixa branca)**
 - Visa explorar certos caminhos do programa (ou sistema).

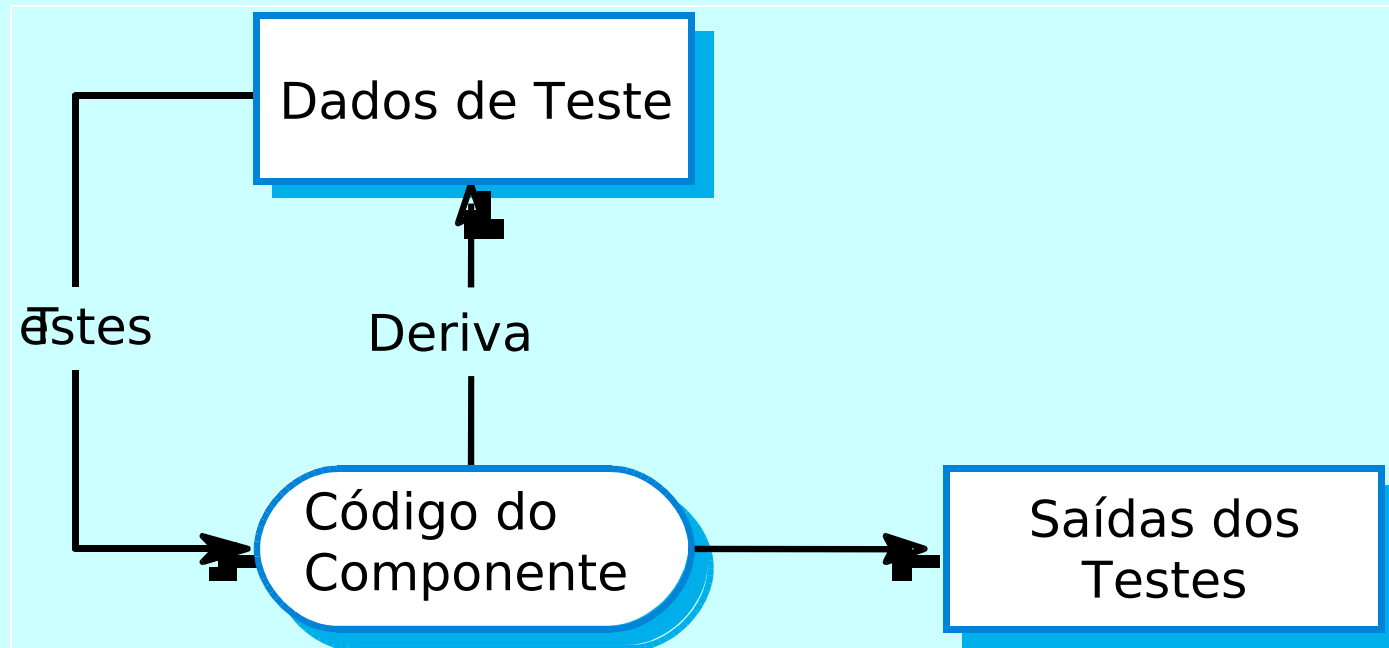
Métodos de Teste

- **Teste Funcional (caixa preta)**
 - Visa verificar funcionalidade baseado apenas nos dados de entrada e saída.
 - Verificar resultados finais , **não importando a estrutura, estados e comportamento internos.**
 - O termo ‘caixa preta’ traduz a idéia de uma caixa que **não** permite que seu conteúdo seja visível do lado de fora.

Métodos de Teste

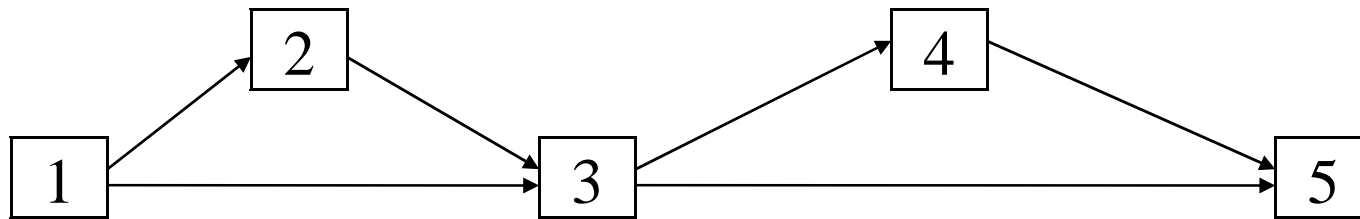
- **Teste Estrutural (caixa branca)**
 - Visa exploração de certos caminhos do programa (ou sistema).
 - (Um caminho é uma das sequências possíveis de serem executadas no fluxo de controle do programa.)
 - Um teste estrutural é realizado prevendo que cada caminho de um programa seja executado ao menos uma vez, e por isto necessita ter acesso e conhecimento da estrutura ou do código fonte do programa.
 - Na verdade, os casos de teste são derivados da estrutura dos programas. O conhecimento do código programa é usado para propor casos de teste adicionais – para “exercitar” um caminho.
 - O termo ‘caixa branca’ traduz a idéia de transparência de uma caixa que permite que seu conteúdo seja visível do lado de fora.

Testes Estruturais



Teste Estrutural

- Exemplo de Casos de Teste para Teste Caixa-Branca
 - Grafo de Controle entre blocos de instruções elementares



Cobertura de instrução: (1,2,3,4,5)

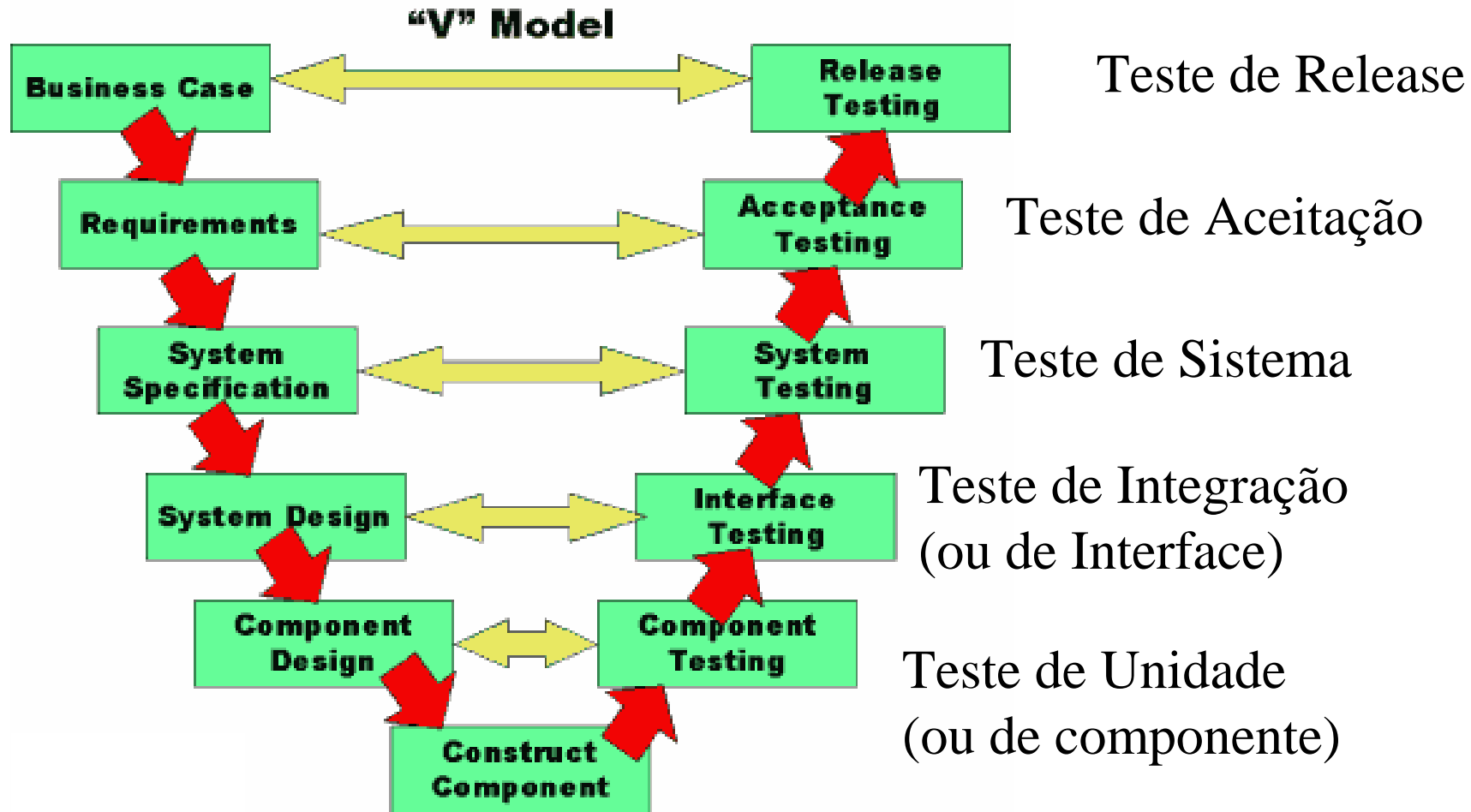
Cobertura de Ramos: (1,2,3,4,5) (1,3,5)

Cobertura de Caminhos Lógicos: (1,2,3,4,5) (1,3,5) (1,2,3,5)
(1,3,4,5)

Níveis de Teste

- As atividades de testes ocorrem ao longo do processo de desenvolvimento.
 - Teste de Aceitação
 - Teste de Integração
 - Teste de Sistema
 - Teste de Unidade (ou Teste Unitário)
 - Teste de Regressão

Níveis de Teste: o modelo 'V'



Teste de Unidade (ou Teste Unitário)

- **Teste de Unidade (ou Teste Unitário ou Teste de Componente)**
- Seu objetivo é encontrar erros em unidades individuais do sistema, sendo que essas unidades são testadas isoladamente.
- É feito pelo **desenvolvedor da unidade (programador)**
- Verificação de uma unidade de software
 - Pode ser via **teste funcional**, desenvolvido a partir da especificação das funções previstas para a unidade, ou via **teste estrutural**, desenvolvido a partir da descrição da estrutura do código da unidade.
- Uma unidade pode ser um módulo, uma subrotina, uma *procedure*, uma classe ou até mesmo um programa simples. No contexto da UDS, unidade é uma porção de código com identificação única.

Teste de Unidade (ou Componentes)

- Teste Funcional (Caixa-Preta)
 - Casos de Teste selecionados a partir das especificações
 - Regras para Casos de Teste:
 - No mínimo, um caso de teste para exercitar cada característica importante do módulo
 - Descobrir casos especiais não cobertos pelos casos de teste da regra anterior (p.ex. valores limites entre classes de soluções, valores extremos de solução)
 - Examinar casos de teste para dados de entrada ausentes, inválidos, errados ou pouco usuais; Exemplos incluem combinações com zero, dados negativos ou não pertencentes ao domínio de entrada

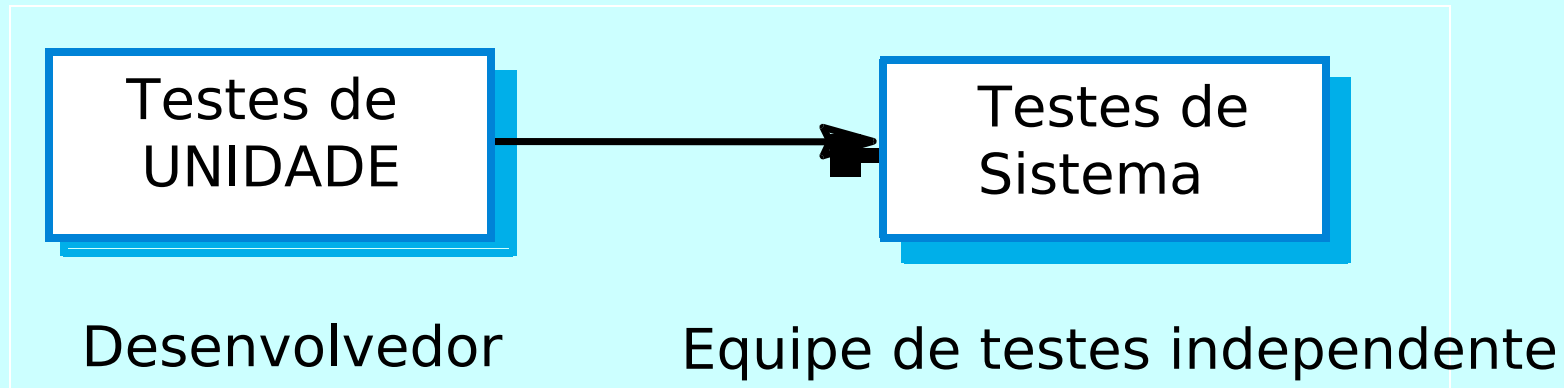
Testes de Unidade (ou Componentes)

- Testes de componentes ou testes unitários visam testar componentes isolados do sistema, um a um
- É um processo de descoberta de defeitos
- Componentes podem ser:
 - Métodos de uma classe
 - Classes com diversos atributos e métodos
 - Conjuntos de classes que proporcionam uma fachada de acesso às suas funcionalidades

Teste de Integração

- Teste da interconexão entre os módulos que visa encontrar defeitos relacionados às interfaces entre unidades.
- Deve ser feito com as unidades já devidamente testadas; Portanto, sucede ao Teste de Unidade;
- Geralmente começa verificando poucas unidades interagindo entre si e termina verificando a cooperação de todas unidades do sistema, incrementalmente.
- Facilitado pela modularização do sistema e pela programação defensiva (módulos que testam seus parâmetros de E/S - pré e pós condições)

Fases de Testes



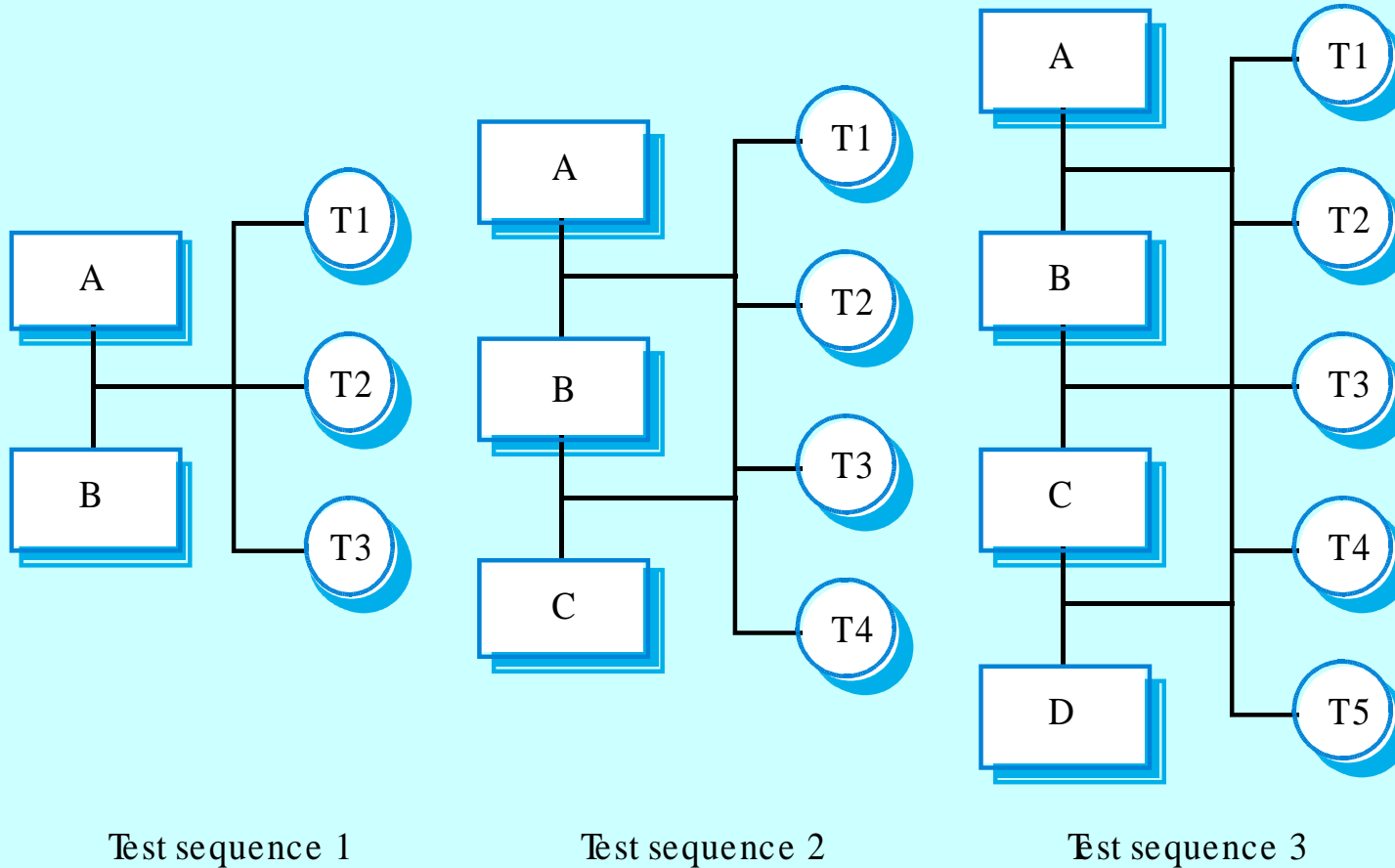
Teste de UNIDADE = Teste de Unidade (isoladamente)

Teste de Sistema = Teste após integrar unidades

Testes de integração

- Envolve construir um sistema através de seus componentes e testá-lo em busca de problemas que podem ocorrer na interação entre os componentes
 - Integração Ascendente (*bottom up*)
 - Uso de *Drivers* (chamadores artificiais)
 - Desenvolve-se o esqueleto de um sistema que é populado por componentes
 - Integração Descendente (*top down*)
 - Uso de *Stubs* (subordinados artificiais)
 - Componentes de infra-estrutura são integrados e depois são adicionados os componentes funcionais
 - Integração híbrida (combinação das anteriores)
 - Para simplificar a localização de erros, os sistemas devem ser integrados de maneira incremental

Testes Incrementais



Níveis de Teste

- As atividades de testes ocorrem ao longo do processo de desenvolvimento.
 - Teste de Aceitação
 - Teste de Sistema
 - Teste de Integração
 - Teste de Unidade (ou Teste Unitário)
- Teste de Regressão

Teste de Sistema

- Verificação global do sistema após a integração com outros sistemas que deverão operar juntos
- Processo de testar um sistema **feito pelo analista** para verificar se satisfaz seus requisitos especificados, ou seja, todos os requisitos funcionais e não funcionais.
- Teste do sistema deve ser realizado com todas as partes do sistema já integradas, operando conjuntamente. Portanto, sucede ao Teste de Integração.

Testes de Sistema

- Envolve integrar componentes de forma a criar um sistema ou sub-sistema
- Pode envolver testar um incremento a ser entregue a um cliente
- Duas fases:
 - Testes de integração: A equipe de testes deve ter acesso ao código fonte. O sistema é testado na medida em que os componentes são integrados
 - Testes de release: A equipe de testes testa o sistema completo a ser entregue como uma ‘caixa-preta’

Teste de Aceitação

- Teste que envolve checagem do sistema em relação a seus requisitos iniciais feito pelo usuário, ou às necessidades do gestor que solicitou o sistema.
- É bastante similar ao teste do sistema; a diferença é que é realizado pelo usuário do sistema.
- Teste conduzido para determinar se um sistema satisfaz ou não seus critérios de aceitação e para permitir ao usuário ou gestor determinar se aceita ou não o sistema solicitado.
- **Requer participação do usuário** pois ele é o único que pode validar e aceitar!!

E o que são Testes de Regressão?

- Estratégia de testes que tenta garantir que:
 - Os defeitos identificados em execuções anteriores dos testes foram corrigidos
 - As mudanças feitas no código não introduziram novos defeitos (ou re-ativaram defeitos antigos)
- Testes de regressão podem envolver a re-execução de quaisquer tipos de testes
- São feitos periodicamente, tipicamente relacionados a interações e testes anteriores

Tipos de Teste

- Atributos de qualidade em foco:
 - o Teste de funcionalidade
 - o Teste de interface
 - o Teste de segurança
 - o Teste de Carga (ou Teste de Stress)
 - o Teste de Desempenho (ou Teste de performance)
 - o Teste de Usabilidade
 - o Teste de Tolerância a Falha
 - o Teste de Instalação

Teste de Funcionalidade

- Verifica se as funcionalidades definidas na especificação de requisitos funcionais são executadas adequadamente, de acordo com o comportamento esperado.
- O MAIS COMUM

Teste de Unidade

Exercício

- Escolha Casos de Teste para teste funcional do seguinte programa
 - Seja um programa que leia três inteiros que corresponderiam aos lados de um triângulo, verifica se é efetivamente um triângulo, e em caso positivo classifica-o em triângulo escaleno, isósceles e equilátero

Exemplo de caso de teste (CT)

(Test Set #, CT #, Nome do CT, Condições, Status, Regras relacionadas, Obs)

TS01	TC04	Remove origin without shipments	Condition 1: Remove an origin for a TBN voyage Condition 2: Remove an origin for a FOB voyage Condition 3: Remove the last origin and add another origin Condition 4: Remove a TBN voyage that is already assigned to a berth				
	TC05	Remove an origin with all shipments belonging to user's trading office	Condition 1: User selects to move the removed the shipment to unassigned Condition 2: User selects to remove shipment permanently				
	TC06	Remove an origin with shipments for a trading office not assigned to the user	Condition 1: Some shipments belongs to user trading office and some does not belongs to it and selects to unassign shipments Condition 2: Some shipments belongs to user trading office and some does not belongs to it and selects to remove shipments Condition 3: All the shipments does not belong to users trading office				

Teste de Unidade

Respostas do Exercício

- Casos de teste das características principais:

1. Equilátero 10,10,10

2. Isósceles 10,10,17 10,17,10 17,10,10

3. Escaleno 8,10,12 8,12,10 10,12,8

4. Não é Triângulo 10,10,22 10,22,10 22,10,10

Casos de Teste usando Template

(Test Set #, CT #, Nome do CT, Condições, [Status, Regras relacionadas, Obs])

(1, 1, Caract.Principal 1 – Equilátero, cond1 – ok normal, [- , - , -]

cond2 – lados zerados, [-, -, -])

(1, 2, Caract. Principal 2 – Isósceles, cond1 – ok normal a,a,b []

cond2 – ok normal a,b,a []

cond3 – ok normal a,b,b [])

(1,3, Caract Principal 3 – Escaleno, cond1 – ok normal [])

(1,4, Caract Principal 4 – não triângulo, cond 1 – ok normal [])

Teste de Unidade

Respostas do Exercício

- Casos de teste do domínio de dados
 - Casos 5 a 8: Executar os testes 1 a 4 acima incluindo casos que contenham o menor (MININT) e o maior (MAXINT) valores inteiros aceitáveis pelo programa
- Dados Anormais

9. Zero 0,0,0
 0, 0, 25 25,0,0 0, 25,0

10. Negativos -10,-10,-10
 -10,-10,15 -10,15,-10 15, -10,-10
 -8,10,17 -8,17,10 17,-8,10

Teste de Unidade

Respostas do Exercício

- Dados Anormais

11. Omitindo dados __, __, __ 10, __, __ __, 15, __
 __, __, 25 10, 5, __ __, 10, 5

10. Incorretos A, B, C =, +, -
 8, 10, A A, 8, 10 8, A, 10
 7E3, 10.5, A 10.5, 7E3, A
 A, 10.5, 7E3

- Limites Máximo e mínimo:

- incluir nos casos de teste inteiros com valores próximos acima e abaixo dos valores máximo e mínimo aceitos pelo programa

Rumo a Automação de Testes

- Ferramentas de teste de software
- JUnit e similares – idéias e conceitos !!!

Princípios

- Os testes devem ser:
 - Automatizados (tanto quanto possível)
 - Repetíveis
 - Auto-verificáveis

Automação de testes

- Testes são atividades caras. Workbenches de testes provêm ferramentas para reduzir o tempo necessário e os custos de testes
- Sistemas como Junit suportam a execução automática de testes
- A maioria dos workbenches de teste são sistemas abertos, já que as necessidades de testes variam conforme a organização
- Normalmente podem ocorrer problemas ao integrar com ambientes fechados para análise e projeto

Introdução

- Junit é um framework de testes de regressão desenvolvido por Erich Gamma e Kent Beck
- É usado por desenvolvedores que utilizam testes unitários em Java
- É um software open source, disponível como um projeto sourceforge:
 - <http://www.junit.org>

Porque usar Junit?

- Testes automatizados provam que funcionalidades estão corretamente implementadas
- Testes mantêm seu valor com o passar do tempo
- Permite que outras pessoas verifiquem se o software ainda está funcionando após mudanças
- Melhora a confiança e a qualidade da aplicação
- Efetivo, open source, integrado

O que é JUnit?

- JUnit é uma ferramenta que suporta a criação e execução de testes de unidade
- JUnit estrutura os testes e provê mecanismos para executá-los automaticamente
- Provê ferramentas para:
 - Asserções
 - Rodar testes
 - Agregar testes (suites)
 - Mostrar resultados

Asserções disponíveis no JUnit

Table 1.2 The JUnit class `Assert` provides several methods for making assertions.

Method	What it does
<code>assertTrue(boolean condition)</code>	Fails if <code>condition</code> is false; passes otherwise.
<code>assertEquals(Object expected, Object actual)</code>	Fails if <code>expected</code> and <code>actual</code> are not equal, according to the <code>equals()</code> method; passes otherwise.
<code>assertEquals(int expected, int actual)</code>	Fails if <code>expected</code> and <code>actual</code> are not equal according to the <code>==</code> operator; passes otherwise. There is an overloaded version of this method for each primitive type: <code>int</code> , <code>float</code> , <code>double</code> , <code>char</code> , <code>byte</code> , <code>long</code> , <code>short</code> , and <code>boolean</code> . (See Note about <code>assertEquals()</code> .)
<code>assertSame(Object expected, Object actual)</code>	Fails if <code>expected</code> and <code>actual</code> refer to different objects in memory; passes if they refer to the same object in memory. Objects that are not the same might still be equal according to the <code>equals()</code> method.
<code>assertNull(Object object)</code>	Passes if <code>object</code> is <code>null</code> ; fails otherwise.

Idéias do Teste Automatizado

- A filosofia é:
 - Deixar que os desenvolvedores escrevam os testes
 - Tornar fácil o desenvolvimento de testes
 - Testar cedo e testar sempre
- Permite experimentar diferentes idéias de projeto (sem quebrar o q estava funcionando):
 - Inicie com “o mais simples que possa funcionar”
 - Refine o projeto através do uso de padrões e aplicação de refatorações
- Tentar quebrar o ciclo:
 - Mais pressão, Menos testes
- Menos tempo com atividades de depuração

Como Usar o JUnit

- Escreva uma classe que estende *TestCase*
 - Para ter acesso aos métodos privados, coloque a classe de teste no mesmo pacote da classe que está sendo testada
- Cada teste é um método
- O conjunto de testes a serem executados é definido por `public static Test suite()`

Síntese - Pontos Chave

- Testes podem mostrar a presença de falhas em um sistema
 - Não podem provar que não existem falhas
- Desenvolvedores de componentes/classes são responsáveis pelos testes de componentes/classes
 - Testes de sistema normalmente são realizados por uma equipe separada de testes
- Utilize os requisitos, sua experiência e o catálogo de erros (memória de teste) para projetar casos de testes

Leitura Recomendada

- “Teste para Desenvolvedores” ”, Cap. 22 (original cap 22)

do livro McConnell, Steve *Code Complete – Um Guia Prático para Construção de Software*, 2ª edição, 2005.

PDF do original em inglês disponível no moodle da disciplina

- Texto (curto) adicional (opcional) para leitura
“Testadores vs Desenvolvedores”

PDF do original em inglês disponível no moodle da disciplina