

**cesae**  
digital

Centro para o Desenvolvimento  
de Competências Digitais

**Author**

Vitor Santos

**Version**

v2.0

# Engenharia de Software II

## *Testes de Software*

**Summary |** [Intro. aos testes; teste vs. inspeções; estratégias de teste \(caixa branca e caixa preta\)](#)

## [ Testes de SW ] | Sumário

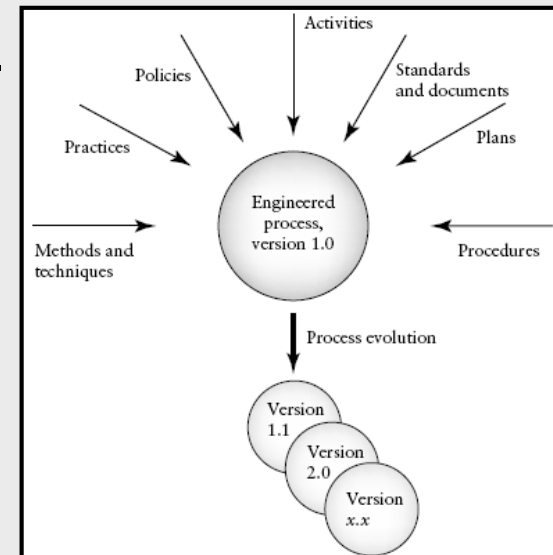
- A importância dos testes
- O especialista de testes
- Processo e qualidade de software
- Testing como um processo
- Testing vs Debugging
- Testing Maturity Model
- Definições básicas (erro, defeito, falha ou fracasso)
- Teste e Caso de teste
- Tipos de teste (níveis ou fases de teste, Atributos de qualidade, Estratégias e técnicas de teste)
- Planeamento e documentação dos testes (Norma IEEE Standard 829-1998)
- Algumas boas práticas

# O especialista de testes

- Alguém cuja formação está baseada nos princípios, práticas, e processos que constituem a disciplina de engenharia de software, e cujo, foco específico está na área de “software testing”
- Deverá ter conhecimento de princípios relacionados com testes, processos, medidas, standards, planos, ferramentas, e métodos, e deverá aprender como aplicá-los às tarefas de teste a serem executadas

# O papel de 'Processo' na qualidade de SW...

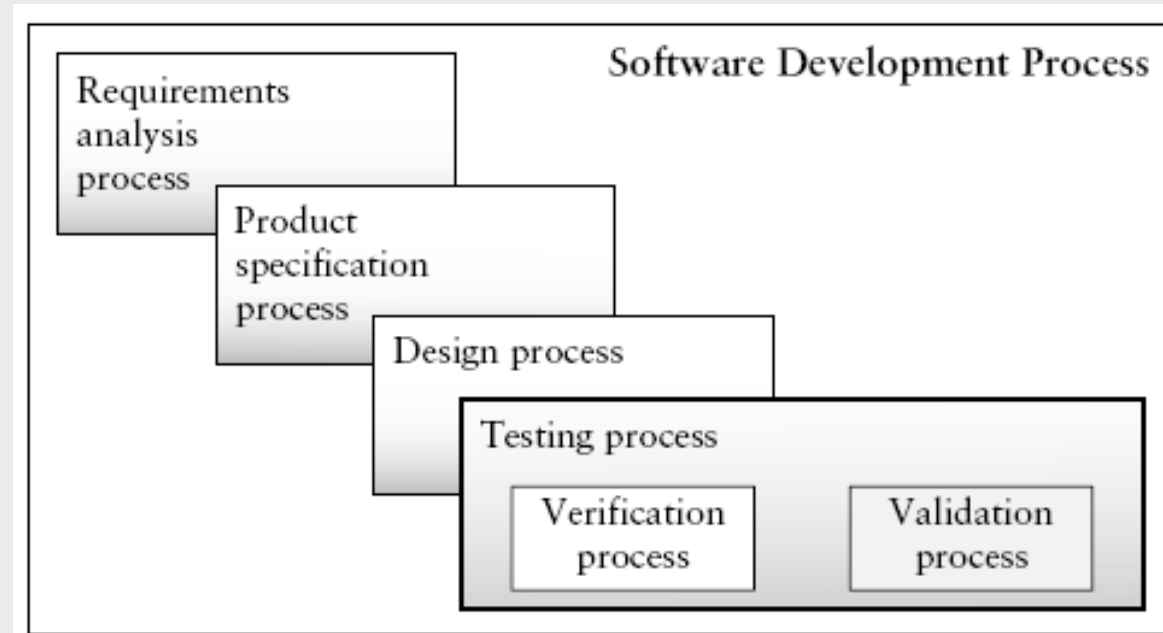
- Processo, no domínio da engenharia de software é:
  - o conjunto de métodos, práticas, standards, documentos, actividades, políticas, e procedimentos que os engenheiros de software usam para desenvolver e manter um sistema de software e os seus artefactos associados, tais como planos de projecto e teste, documentos de projecto, código e manuais.



A maioria dos engenheiros de software concordaria que o “teste” é um componente vital de um processo de qualidade de software, e é um dos maiores desafios e das actividades mais caras realizadas durante o desenvolvimento e manutenção de software.

# Processo de desenvolvimento de SW e Testes

- No processo de desenvolvimento de software existem vários processos incluindo o “Testing”
- Está relacionado com outros dois processos:
  - Verificação
  - Validação



- **Validação - foco no produto**

- É o processo de avaliar um sistema ou componente de software durante, ou no final, do ciclo de desenvolvimento para determinar se satisfaz os requisitos especificados.

- **Verificação - foco no processo**

- É o processo de avaliar um sistema ou componente de software para determinar se os produtos de uma determinada fase de desenvolvimento satisfazem as condições impostas no começo dessa fase (associado a actividades de inspecção, revisão)

- É geralmente descrito como um grupo de procedimentos realizados para avaliar algum aspecto ou parte de software
- Pode ser descrito:
  - como um processo usado para revelar defeitos no software e para estabelecer que o software atingiu um grau especificado de qualidade relativamente aos atributos seleccionados.

- Cobrem as actividades de validação e verificação, e incluem no domínio de testing tudo o que se segue:

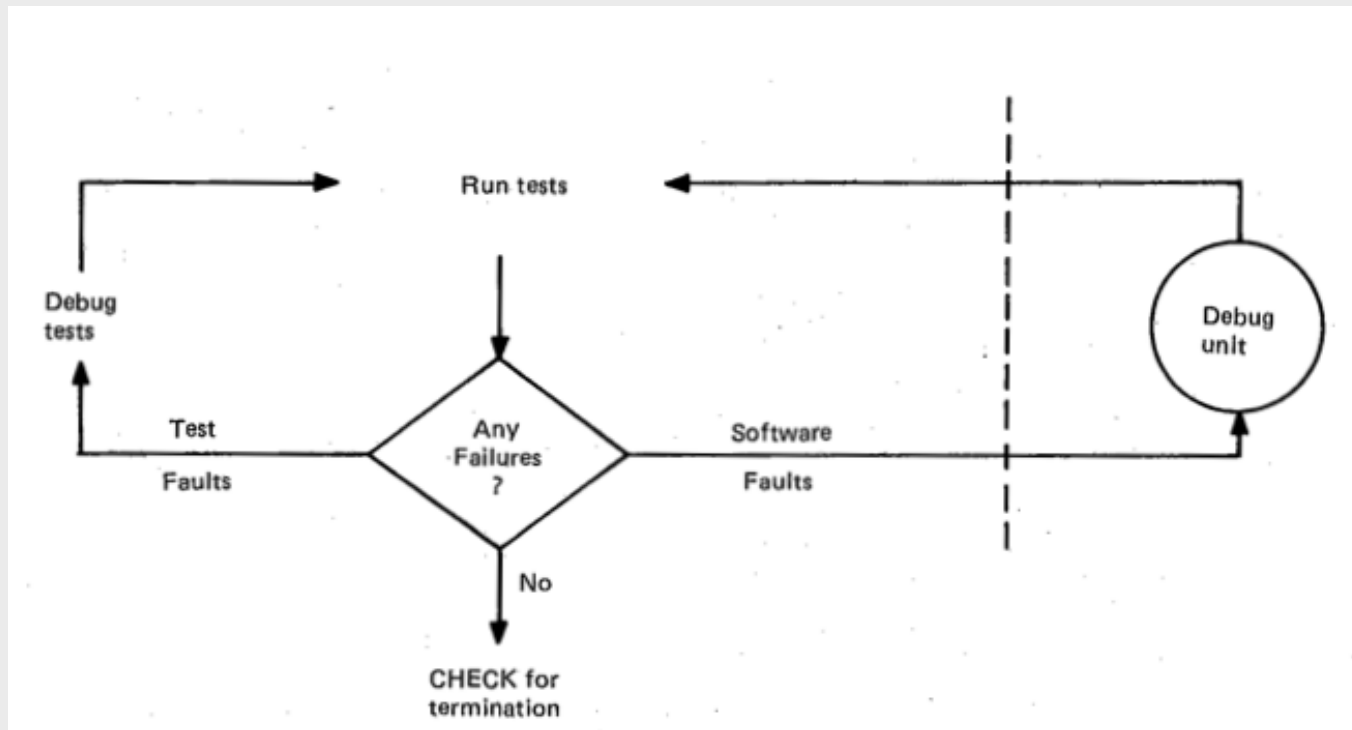
:: Revisões técnicas	:: Testes unitários
:: Planeamento de testes	:: Testes de integração
:: Tacking de testes	:: Testes de sistema
:: Desenho de casos de teste	:: Testes de aceitação

- Estas definições também descrevem “testing” com um processo com um propósito duplo:
  - que revele defeitos
  - e que é usado para avaliar atributos de qualidade de software, tais como segurança, usabilidade e exactidão



# [ Testes de SW ] | Teste vs Debug

Testing vs Debugging



ANSI/IEEE sd. 1008-1987

# Inspeções de código e testes

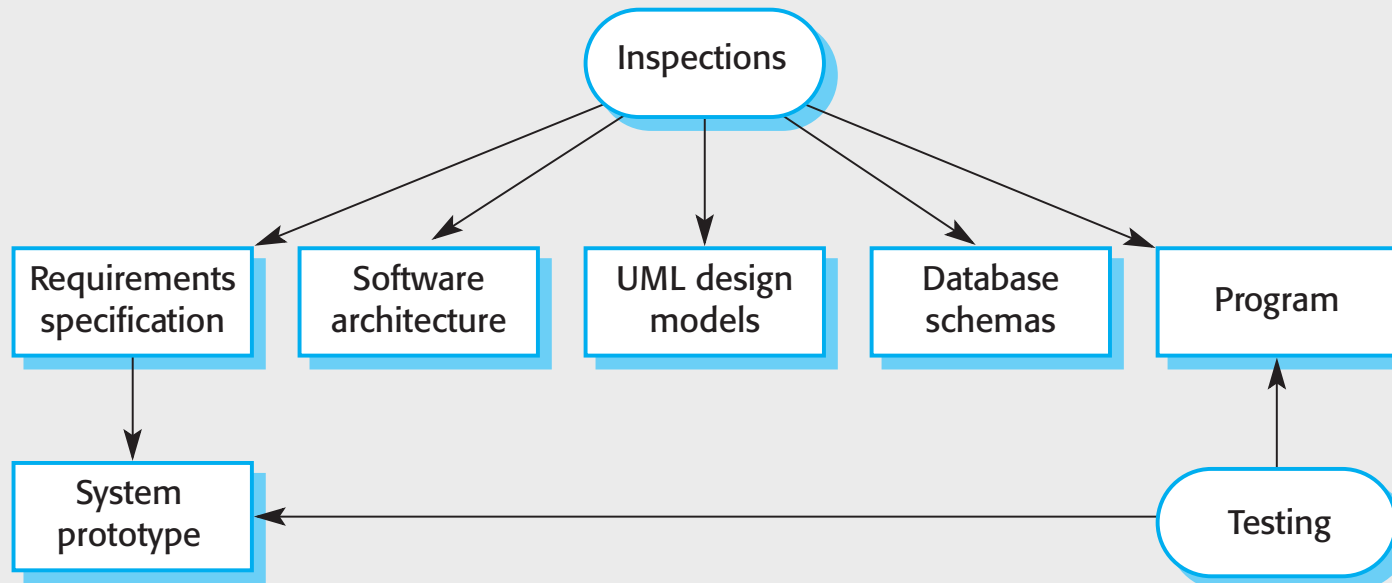
- **SW Inspections**

- verificação estática. Focada numa determinada representação estática do sistema com o intuito de “descobrir” problemas

- **SW Testing**

- verificação dinâmica. avalia o comportamento observável de um produto de software.

# Inspeções de código e testes



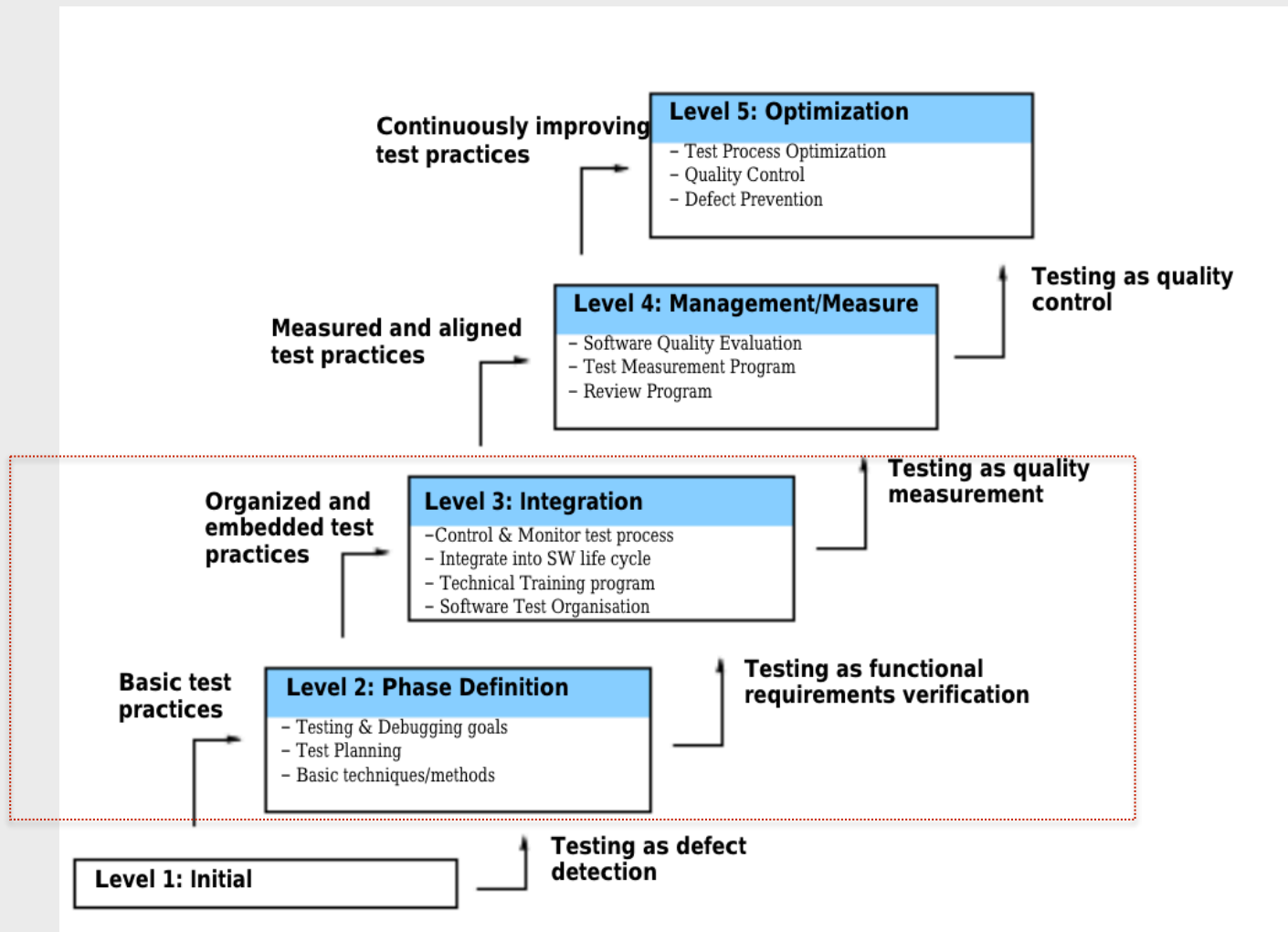
# Inspeções de código e testes

- examinação (estática) da fonte\* com o intuito de descobrir problemas
- não requer a execução do sistema
- esta observação/examinação pode ser aplicada a **requisitos, design, dados de configuração, dados de teste, etc...**
- na realidade é uma boa técnica para descoberta de erros

# Inspeções de código e testes

- Durante a execução há erros que “encobrem” outros erros;
- Não há a necessidade e/ou preocupação com interações
- Versões incompletas podem ser inspecionadas
- Aquando da inspecção podemos considerar outros atributos da qualidade como por exemplo: conformidade com standards, padrões, portabilidade etc...
- Contudo, as inspecções não conseguem avaliar a conformidade com os requisitos reais do cliente nem questões relacionadas com usabilidade e performance
- **SW Inspections** e **Software Testing** são complementares

# Modelo de maturidade de testes de SW



# Conceitos e Definições básicos

## • Erros

- Um erro é um engano, uma ideia errada/equívoco, ou interpretação errada/má compreensão por parte de um “desenvolvedor” de software
- Na categoria de “desenvolvedor” incluímos engenheiros de software, programadores, analistas e os testers.
  - Exemplo: um “desenvolvedor” entender mal a notação de desenho, um programador digitar o nome de uma variável incorrectamente.

## • Defeitos (Faults/Defects)

- Um defeito é introduzido no software como resultado de um erro. É uma anomalia no software que pode fazer com que este se comporte incorrectamente, e não conforme a sua especificação.

- **Defeitos (continuação)**
  - São por vezes chamados “**bugs**”
  - O uso do termo “bug” trivializa o impacto do defeito em termos da qualidade de software....
  - O uso do termo defeito está também associado com artefactos de software, tais como documentos de requisitos e projecto.
  - Os defeitos que ocorrem nos artefactos são causados por erros e são normalmente detectados no processo de revisão.



# Conceitos e Definições básicos

- **Falha/Fracasso** (Failures)

- Fracasso é a **incapacidade de um sistema ou componente de software executar as funções que lhe são requeridas**, dentro dos requisitos de desempenho especificados
  - Por **exemplo**, durante a execução de um componente ou sistema de software, um tester, “desenvolvedor”, ou utilizador **observa que este não produz os resultados esperados**.
  - Comportamento incorrecto pode incluir produzir valores incorrectos para variáveis de saída, etc...
  - Um defeito no código não produz sempre um fracasso....
- **Na realidade, software defeituoso pode operar durante grandes períodos de tempo sem exibir qualquer comportamento incorrecto.**

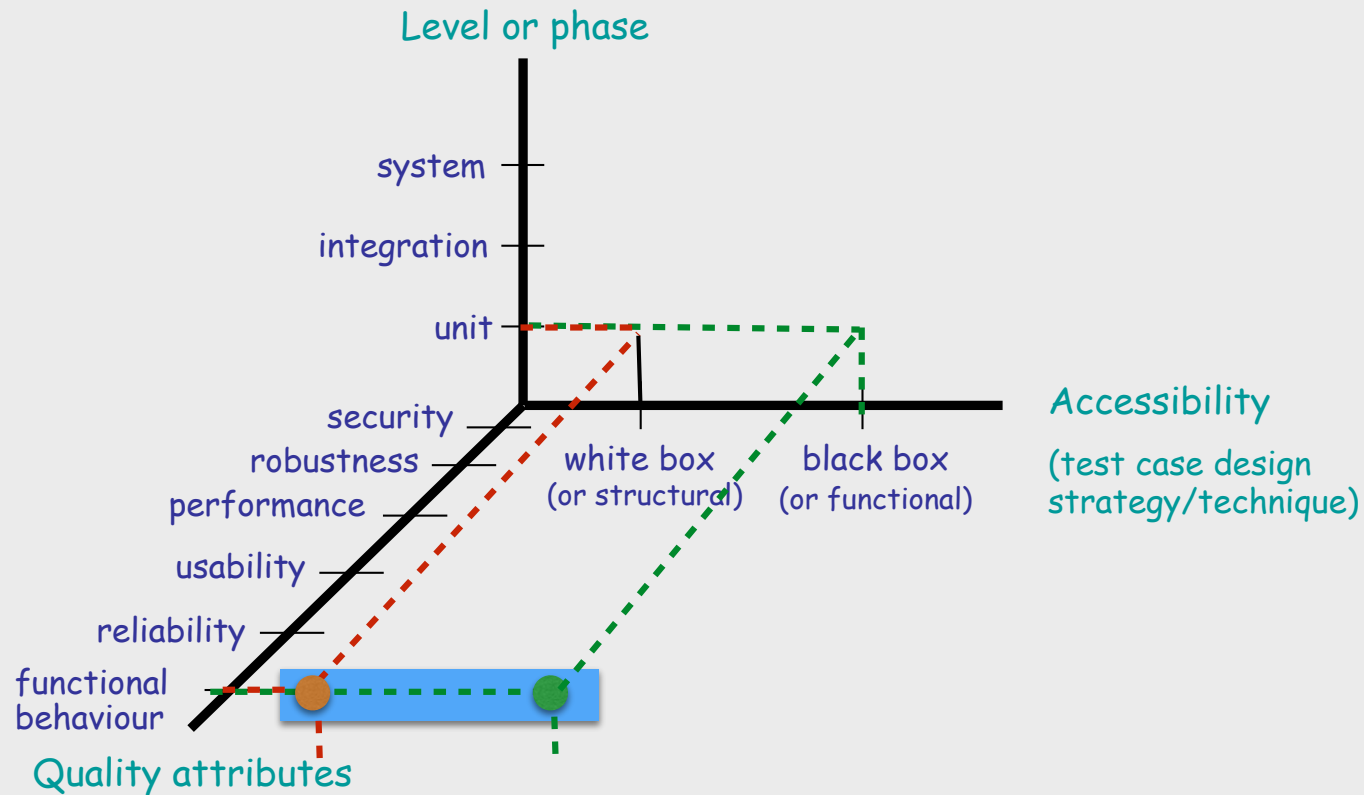
# Conceitos e Definições básicos

- Qual abordagem usual para detectar defeitos numa parte de software? .....
- Para decidir se o software passa ou não no teste, o tester precisa de conhecer os outputs para o software, dado um conjunto de inputs e condições de execução.
- O Tester junta esta informação num item chamado “caso de teste”
- Um caso de teste é num sentido prático um item relacionado com teste, que contem a seguinte informação:
  - Um conjunto de entradas de teste: são dados recebidos de uma fonte externa pelo código em teste. A fonte externa pode ser hardware, software, ou humana.
  - Condições de execução: são condições requeridas para executar o teste, por exemplo, um certo estado de uma base de dados, ou uma configuração de um dispositivo de hardware.
  - Saídas esperadas: São os resultados especificados para serem produzidos pelo código em teste.

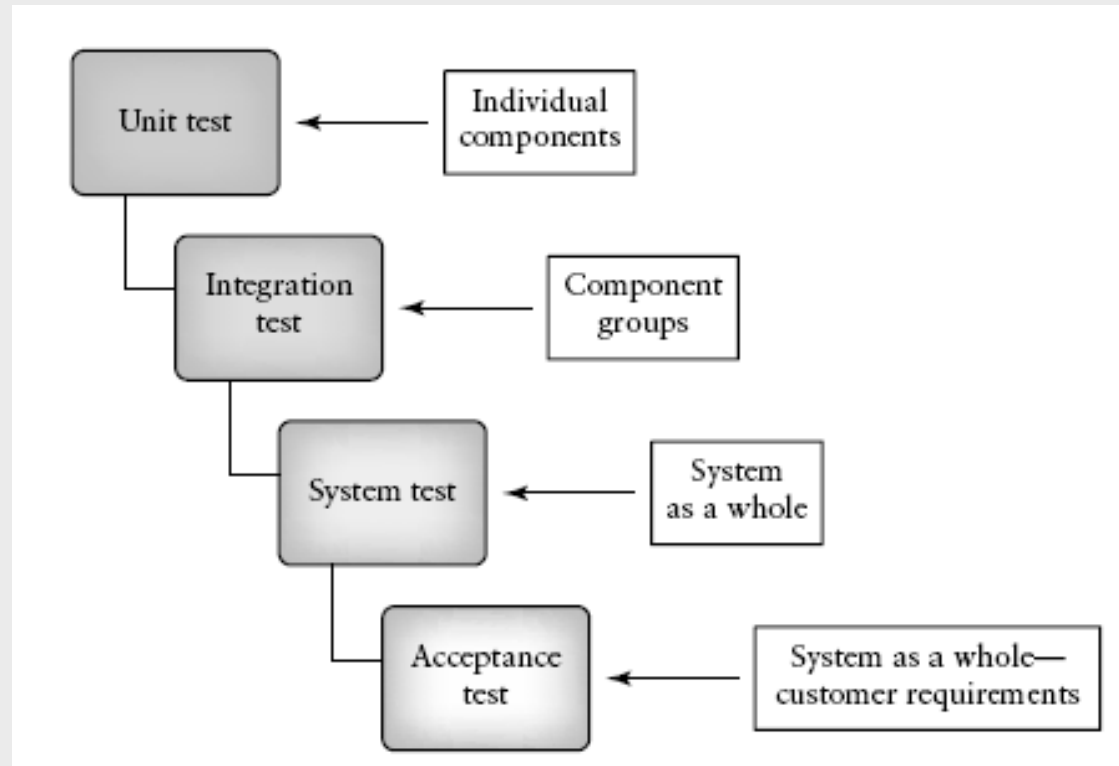
## • O que é um teste?

- É um grupo de casos de teste relacionados, ou um grupo de casos de teste e procedimentos relacionados
- Test Oracle : É um documento, ou parte de software que permite aos testers determinar se um teste passou ou falhou (um programa, ou um documento que produz ou especifica o resultado esperado de um teste, pode servir como um oracle)
- Test Bed: é um ambiente que contém todo o hardware e software necessário para testar um componente de software ou um sistema de software
  - Este inclui o ambiente de teste completo, ou seja, tudo o que é necessário para apoiar a execução dos testes (por exemplo: simuladores, ferramentas de software, etc....)

# Conceitos e Definições básicos



# Conceitos e Definições básicos



Unit Testing, Integration Testing, System Testing são os 3 diferentes níveis de Testes de SW considerados pelo SWEBOK

- **Testes unitários:**
  - Os diversos componentes são codificados e testados de forma isolada, garantindo assim a respectiva correcção interna. Incidem sobre parcelas do sistema, e são realizados por cada programador de forma independente.
  - Testes baseados na experiência, especificações e código;
  - O principal objectivo é detectar defeitos funcionais e estruturais em parcelas de código.

- **Testes de integração:**
  - testes parcelares, que vários programadores realizam conjuntamente com vista a garantir que vários componentes interactuam entre si de forma adequada.
  - Testes de grupos integrados de componentes, integrados para criar um subsistema;
  - O principal objectivo é detectar defeitos que ocorrem nas interfaces das unidades e no seu comportamento comum.

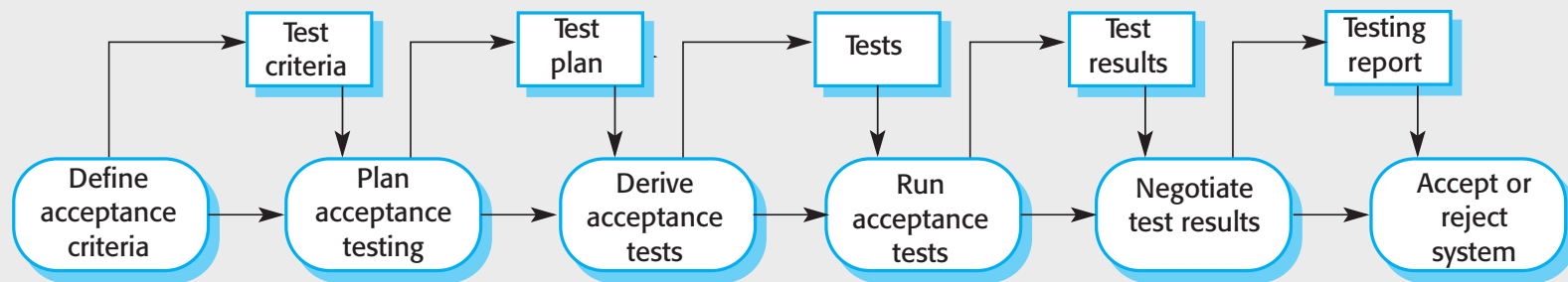
- **Testes de sistema:**

- testes globais em que todos os componentes do sistema são integrados; possibilitam a verificação da conformidade do sistema com todos os requisitos definidos.
  - Normalmente são da responsabilidade de uma equipa de teste independente;
  - São normalmente baseados num documento de requisitos (requisitos/especificações funcionais e requisitos de qualidade);
  - O principal objectivo é avaliar atributos tais como usabilidade, fiabilidade e desempenho (assumindo que os testes unitários e de integração foram realizados);



- Testes de aceitação:
  - testes formais que os utilizadores realizam sobre o sistema. Quando o sistema passa este (difícil!!!!) teste, o cliente deverá aceitar o sistema como “pronto” e consequentemente este pode ser colocado em produção, ou operação, efectiva.
  - O principal objectivo é avaliar se o produto vai de encontro aos requisitos do cliente e expectativas.

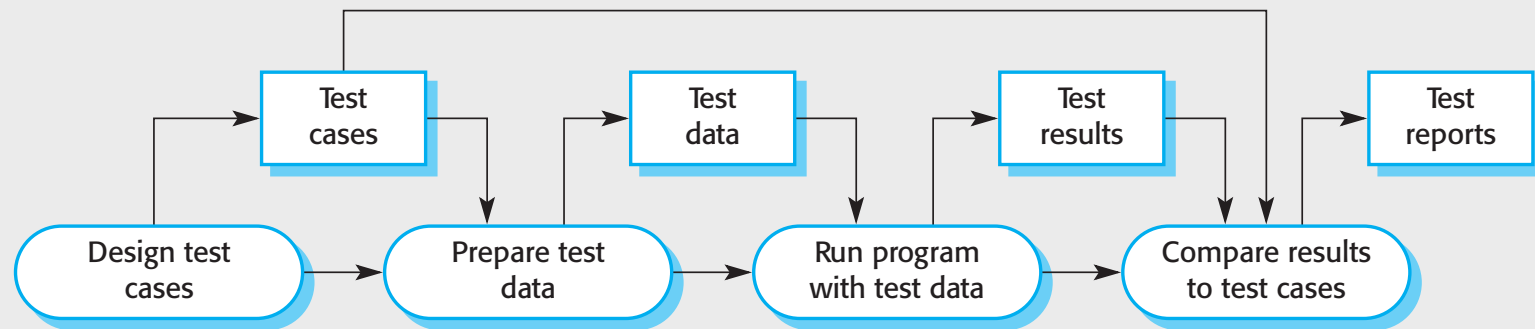
# Ciclo de vida dos testes



# Tipo de testes

- **Testes de desempenho:** permitem analisar o tempo de resposta do sistema e, dum modo geral, verificar o nível de utilização de recursos disponíveis.
- **Testes de usabilidade:** permitem analisar a adequabilidade do desenho das interfaces homem-máquina e constatar se o sistema é fácil de utilizar.
- **Testes funcionais:** permitem determinar a correcção da implementação de funcionalidades, conforme especificadas pelos correspondentes requisitos funcionais.
- **Testes de segurança:** nos testes de segurança estamos à procura de um comportamento anómalo que não sabemos quando acontece....Precaver contra ataques; Garantir robustez do software face a determinados ataques típicos; Detectar vulnerabilidades; Preparar medidas de contingência; Oferecer maior valor acrescentado ao cliente; ...

- **Testes de robustez:** o objectivo é determinar o comportamento de um sistema em situações hostis. Normalmente pensados em conjunto com os testes de segurança.
- **Testes de fiabilidade:** o objectivo é avaliar a capacidade de um sistema de software desempenhar as suas funções sob determinadas condições num determinado período de tempo.



- Norma ANSI/IEEE 829-1998 para Documentação de testes de software define plano de testes como:
  - Um documento que define o âmbito, abordagem, recursos e escalonamento (planeamento) das actividades de teste previstas. Identifica itens de teste, as funcionalidades a serem testadas, as tarefas de teste, quem executará cada tarefa, e quaisquer riscos que requeiram planos de contingência.
- Planos de teste são documentos extensos, normalmente compostos por vários documentos mais pequenos

- O plano de testes como um produto
  - Um bom plano de testes ajuda a organizar e gerir o esforço de teste
  - Muitos planos de teste ultrapassam este âmbito, e tornam-se num produto por si só
  - A estrutura, formato, e nível de detalhe são determinados não só pelo que se entende como mais apropriado para eficácia das tarefas de teste, mas também pelos requisitos do cliente ou entidade reguladora

- Plano de testes: Produto ou Ferramenta?
  - O que os clientes normalmente querem é programas que funcionem correctamente
  - Os clientes tipicamente não estão interessados nos testes efectuados
  - Os clientes estão interessados na forma como o programa funciona
  - Para estes clientes, o plano de testes não é um produto
  - Um plano de testes é uma ferramenta valiosa na medida em que ajuda a gerir o projecto de testes e a encontrar falhas do programa.



## [ Testes de SW ] | Plano de Testes

Description

- O plano de testes como uma ferramenta
  - A norma ANSI/IEEE 829 requer
    - especificações da concepção de testes
    - especificação dos casos de teste
    - registos de testes
    - especificação dos procedimentos de teste
    - relatórios dos testes
    - especificações de entrada/saída
    - especificação de requisitos de procedimentos especiais
    - notas sobre a dependência entre casos

- O plano de testes como uma ferramenta
  - A norma ANSI/IEEE 829 requer (cont.)
    - listas de documentos a serem elaborados após testes
    - escalonamento (planeamento) dos testes
    - planeamento de recursos humanos
    - lista (escrita) de responsabilidades de cada elemento da equipa
    - critérios para a suspensão e reactivação dos testes
    - etc., etc., etc.

## •Secções do plano de testes (IEEE Standard 829)

- Identificador do plano de testes
  - Nome ou número único, que identifica o projecto
- Introdução
  - Descrição do objectivo do plano de testes. Inclui referências a todas as
  - normas e documentos relevantes na definição da política seguida
- Itens de teste
  - Lista de todos os componentes do programa ( função, módulo, classe, método, etc.) que vão ser testados, incluindo os documentos de referência. Se apropriado, listar o que NÃO vai ser testado.
- Características a serem testadas
  - Referenciadas às especificações do desenho (concepção) do teste.
- Características que não vão ser testadas
  - Quais, e porquê.

## • **Secções do plano de testes (IEEE Standard 829) (cont.)**

- Abordagem
  - Quem faz os testes, principais actividades, técnicas e ferramentas.
  - Restrições, nomeadamente prazos e recursos.
- Critério de sucesso/insucesso
- Critério de suspensão e retoma dos testes
  - Quando suspender os testes para que um problema seja corrigido?
  - Quando retomar? O que fazer ao retomar os testes?
- Quais os produtos dos testes
  - Que documentos vão ser produzidos como resultado dos testes
- Tarefas de teste
  - Listar as tarefas necessárias para preparar e realizar os testes.
  - Dependências entre as tarefas.
  - Quem as faz, qual o esforço necessário, quando é feita cada tarefa.
- Ambiente necessário

## • **Secções do plano de testes (IEEE Standard 829) (cont.)**

- Responsabilidades
- Necessidade de recursos humanos e formação
- Escalonamento (planeamento)
  - Listar todas as datas marcantes (milestones).
  - Listar quando os recursos humanos e materiais vão ser necessários.
- Riscos e contingências
  - O que pode correr mal e atrasar os testes?
  - O que fazer nesse caso?
- Aprovações
  - Quem tem que aprovar o plano?

- A documentação de teste facilita a tarefa de teste
  - Para criar um bom plano de teste, é necessário investigar o programa de forma sistemática à medida que se vai desenvolvendo o plano
  - O tratamento do programa torna-se mais claro, mais exaustivo, e mais eficiente
  - A documentação de testes facilita a comunicação sobre as tarefas e o processo de teste
  - A documentação de teste fornece uma estrutura para organizar, escalonar (planear) e gerir o projecto de teste

- Desenvolvimento inicial do material de teste
  - Primeiros passos para desenvolver um plano de testes
    - Testar contra a documentação (especificação, manual, ...)
    - Criar uma documentação que seja organizada para facilitar testes eficientes, por exemplo uma lista de funções
    - Fazer uma análise simples de limites
      - Testar valores limite em todas as situações em que se podem fornecer dados

- Onde focar a seguir
  - Erros mais prováveis
  - Erros mais visíveis (para o utilizador)
  - Áreas do programa mais usadas
  - Área do programa mais referida como distintiva
  - Áreas mais difíceis de corrigir
  - Áreas melhor compreendidas



- Testar tão cedo quanto possível
- Escrever os casos de teste antes do software ser testado
  - aplicar-se a qualquer nível: unidade, integração e sistema
  - ajuda a obter conhecimento sobre os requisitos
- Codificar os casos de teste
  - Devido à frequente necessidade de repetição do teste cada vez que o software é modificado
- O “tester” deve ser desde o mais crítico do sistema até ao mais independente (colegas, outro departamento, outra empresa)
- Ser consciencioso relativamente a custos
- Definir as saídas esperadas no teste com base nas especificações e não no código.

- Burnstein, I., Practical Software Testing: A Process-Oriented Approach. Springer Professional Computing, 2003.
- Norma IEEE Standard 829-1998 for Software Test Documentation (IMPORTANTE A LEITURA PARA A REALIZAÇÃO DOS TRABALHOS)
- Ian Sommerville ([acrescentar](#))



Centro para o Desenvolvimento  
de Competências Digitais

**Author**

Vitor Santos

# Engenharia de Software II

**Version**

V2.0

***Estratégias de Testes de Software***

**Summary | Testes de caixa preta e testes de caixa branca**

- Estratégias ou abordagens de desenho de casos de teste
  - Caixa branca
  - Caixa Preta

# Testes de Caixa Preta


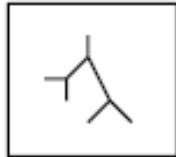
- Abordagem/estratégia também conhecida como “testes funcionais”
- Avaliam o comportamento externo do componente de software, sem se considerar o comportamento interno do mesmo
- Apenas são consideradas as entradas e saídas como uma base para desenho dos casos de teste
- Objectivo: assegurar que os requisitos do software e as especificações foram atendidos, ou seja, valida se o que foi especificado foi implementado correctamente
- É executado considerando como base os requisitos e as funcionalidades do software
- Quanto mais entradas são fornecidas, mais rico será o teste. Numa situação ideal todas as entradas possíveis seriam testadas, mas na ampla maioria dos casos isso é impossível

# Testes de Caixa Branca

- Abordagem também conhecida por “teste estrutural”, pois avalia o comportamento interno dos componentes de software
- Abordagem principalmente usada para a realização de testes unitários
- Neste caso o “tester” tem conhecimento da estrutura lógica interna do software a ser testado
- Objectivo: Determinar se todos os elementos lógicos e de dados nos componentes de software estão a funcionar adequadamente
- O “tester” define os casos de teste para determinar se existem defeitos na estrutura do programa
- O conhecimento necessário para esta abordagem é adquirido nas fases posteriores do ciclo de vida do software, especificamente na fase de desenho.

# Caixa Preta vs Caixa Branca

- A estratégia de “caixa preta” pode ser usada tanto para componentes de software grandes como pequenos
- Os testes de “caixa branca” são mais apropriados para testar componentes pequenos (pelo facto do detalhe requerido para o desenho do teste ser muito elevado)

Test Strategy	Tester's View	Knowledge Sources	Methods
Black box		Requirements document Specifications Domain knowledge Defect analysis data	Equivalence class partitioning Boundary value analysis State transition testing Cause and effect graphing Error guessing
White box		High-level design Detailed design Control flow graphs Cyclomatic complexity	Statement testing Branch testing Path testing Data flow testing Mutation testing Loop testing

- O “Tester” (engenheiro de software) pode derivar um conjunto de condições de entrada que exercitem praticamente todos os requisitos funcionais para um programa.
- O “Tester” fornece os inputs (entradas) , os testes são executados e é verificado se os resultados obtidos são “equivalentes” aos previamente especificados.
- Os testes de caixa preta procuram descobrir erros nas seguintes categorias:
  - funções incorrectas ou ausentes;
  - erros de interface;
  - erros nas estruturas de dados ou no acesso a base de dados externas;
  - erros de inicialização e de término.



- Tipicamente, são projectados para responder às seguintes questões:
  - Como é testada a validade funcional de um sistema?
  - Que classes (e.g. conjunto de valores) de entrada poderão constituir bons casos de teste?
  - O sistema é particularmente sensível a determinados valores de entrada?
  - Como estão isoladas as fronteiras de uma determinada classe de dados?
  - Quais os índices de dados e volumes de dados que o sistema pode tolerar?
  - Que efeito poderão ter certas combinações específicas de dados sobre a operação do sistema?

- Numa abordagem baseada em testes de caixa preta, para testar uma determinada operação, o “tester” deve obter casos de teste suficientes para verificar que:
  - para cada valor aceite (escolhido) como valor de entrada (input), um valor apropriado é retornado (output) pela operação.
  - para cada valor não aceite (escolhido) como valor de entrada (input), apenas um valor apropriado é retornado (output) pela operação
  - para cada estado de entrada válido ocorre um estado de saída apropriado
  - para cada estado de entrada inválido, ocorre um estado de saída apropriado

- Alguns métodos (técnicas) de caixa preta:
  - Equivalence Class partitioning (Partição em classes de equivalência)
  - Boundary Value Analysis (Análise de valor limite)
  - Cause-and-Effect Graphing (Técnicas de Grafos de causa-efeito)
  - Random testing
  - Error Guessing
  - State Transition Testing
  - ...

The acordo com o SWEBOK, as técnicas mencionadas são classificadas como:  
“Specification-based Techniques” dentro das “Test Techniques” no domínio de “Software Testing”

# Equivalence Class Partitioning (ECP)

- Técnica destinada a reduzir o número de testes necessários
- Técnica que divide o domínio de entrada (ou saída) em classes de dados em que os casos de teste podem ser derivados
- Para cada operação, o “tester” deve identificar as classes de equivalência dos argumentos e os estados dos objectos
- Uma classe de equivalência: um conjunto de valores de acordo com os quais o objecto se deve comportar
  - Por exemplo, um Conjunto contém três classes de equivalência: vazio, algum elemento e cheio
- Deve-se considerar classes de valores válidos e valores inválidos
  - **Exemplo para  $\text{sqrt}(x)$ :  $x < 0$  (inválido),  $x \geq 0$  (válido)**

“The input domain is subdivided into a collection of subsets, or equivalence classes, which are deemed equivalent according to a specified relation, and a representative set of tests (sometimes only one) is taken from each class.” in SWEBOK about Equivalence Class Partitioning

# ECP - exemplo

- **Testar uma função que calcula o valor absoluto de um inteiro X**
- Classes de equivalência:

<u>Critério</u>	<u>Classe de Equivalência válida</u>	<u>Classe de equivalência inválida</u>
<u>nº de entradas (inputs)</u>	1	0 , >1
<u>tipo de entradas</u>	inteiro	não inteiro
<u>valor específico x</u>	< 0 , >= 0	

Casos de testes baseados nas classes de equivalência especificadas:

X= -10



x = 100



x=



x= 10 20



x= "XYZ"



# Boundary Value Analysis (BVA)

- Técnica focada nos limites do domínio de entrada (ou saída) e imediatamente acima e abaixo (além de ou em vez de valores intermédios)
- Testar também valores especiais (null, 0, etc.)
- Baseada na observação de que bugs ocorrem frequentemente em valores fronteira:
  - Problemas com índices de arrays, decisões, overflow, etc.
  - Se o sistema se comportar bem nos casos fronteira então provavelmente comportar-se-á bem em valores intermédios.

“Test cases are chosen on and near the boundaries of the input domain of variables, with the underlying rationale that many faults tend to concentrate near the extreme values of inputs.” in SWEBOK about Boundary Value Analysis

- Para um dado domínio de valores:
  - Se as condições de entrada especificarem um intervalo entre a e b, os casos de testes deverão incluir a e b, bem como valores acima e abaixo de a e b.
  - Se a condição de entrada especificar um número de valores, os casos de teste deverão contemplar o valor máximo e mínimo desses valores, bem como valores abaixo e acima do valor mínimo e do valor máximo.
  - Se as estruturas de dados internas do programa tiverem limites (e.g. limite de tamanho), o “tester” deverá certificar-se de que testa os limites.

# Exemplo -Rotina de Pesquisa

```
procedure Search (Key : ELEM ; T: ELEM_ARRAY;  
                  Found : out BOOLEAN; L: out ELEM_INDEX) ;  
  
Pre-condition  
    -- the array has at least one element  
    T'FIRST <= T'LAST  
  
Post-condition  
    -- the element is found and is referenced by L  
    ( Found and T (L) = Key)  
  
or  
    -- the element is not in the array  
    ( not Found and  
      not (exists i, T'FIRST >= i <= T'LAST, T (i) = Key ))
```

(source: Ian Sommerville)



# BVA - Partições segundo as condições

- **P1 - Entradas de acordo com as pré-condições (válida)**
  - array com um valor (valor limite)
  - array com mais de um valor (de diferente tamanho de caso de teste para caso de teste)
- **P2 - Entradas em que a pré-condição não é assegurada (inválida)**
  - array com tamanho zero
- **P3 - Entradas em que o elemento chave é membro do array**
  - alternar a primeira, a última e a posição do meio em diferentes casos de teste.
- **P4 - Entradas em que o elemento chave não é membro do array**

# BVA - casos de teste válidos

## Cenários

Array	Element
Single value	In sequence
Single value	Not in sequence
More than 1 value	First element in sequence
More than 1 value	Last element in sequence
More than 1 value	Middle element in sequence
More than 1 value	Not in sequence

## Casos de teste

Input sequence (T)	Key (Key)	Output (Found, L)
17	17	true, 1
17	0	false, ??
17, 29, 21, 23	17	true, 1
41, 18, 9, 31, 30, 16, 45	45	true, 7
17, 18, 21, 23, 29, 41, 38	23	true, 4
21, 23, 29, 33, 38	25	false, ??

- Casos de Teste baseados em Argumentos de Entrada
- Um argumento de entrada é aquele usado por uma operação
- O “tester” deve criar casos de teste usando argumentos de entrada para cada operação, de acordo com cada uma das seguintes condições de entrada:
  - Valores normais de cada classe de equivalência
  - Valores na fronteira de cada classe de equivalência
  - Valores fora das classes de equivalência
  - Valores inválidos:
    - Nota: tratar o estado do objecto como um argumento de entrada. Se, por exemplo, se testar uma operação de adição de acordo com um Conjunto de objectos, deve-se testar a adição com valores de todas as classes de equivalência do Conjunto, ou seja, com um Conjunto completo, com algum elemento no Conjunto e com um Conjunto vazio.

- Casos de Teste baseados em Argumentos de Saída
- Um argumento de saída é aquele que é alterado por uma operação.
- Um argumento pode ser de entrada ou de saída. O “tester” deverá seleccionar a entrada de modo a que a saída esteja de acordo com o seguinte:
  - Valores normais de cada classe de equivalência
  - Valores na fronteira para cada classe de equivalência
  - Valores fora das classes de equivalência
  - Valores inválidos:
    - Nota: tratar o estado do objecto como um argumento de saída. Se, por exemplo, se testar uma operação de exclusão em uma Lista, deve-se escolher valores de entrada de modo que a Lista fique cheia, contenha algum elemento ou fique vazia após a execução da operação (teste com valores de todas as respectivas classes de equivalência).