Testes de Software

Fundamentos

criado: Setembro / 2001

alterado: Set / 2009

Tópicos

- Objetivos
- Fases
- Processo

Referências

G.J.Myers. *The Art of Software Testing*. John Wiley & Sons, 1979.

B.Beizer. *Software Testing Techniques*. International Thomson Computer Press, 2ª ed, 1990.

E.Martins, *Verificação e Validação de Software*. Notas de Curso.

R.S.Pressman. *Software Engineering. A Practitiner's Approach*. 4ª edição, 1997.

R.Binder. *Testing OO Systems*. Addison Wesley, 2000.

W. de Pádua Paula Fº. *Engenharia de Software*. Ed. LTC, 2ª ed., 2002.

I. Sommerville. *Software Engineering*. 8ª ed, 2007.

M.E.Delamaro et al. *Introdução ao Teste de Software*. 2007.

P. Ammann, J. Offutt. *Introduction to Software Testing*. 2008.

Técnicas de V&V

- Verificação dinâmica
  - envolve a execução do produto (código ou modelo executável)
  - visa encontrar falhas ou erros no produto

  exemplos:
    - simulação
    - execução simbólica

    ➲ **testes**

- Qual o objetivo dos testes?
- Quando começam?
- Quem deve aplicá-los?
- Quando terminam?

Qual o objetivo dos testes?

- **Executar o produto de sw com o intuito de detectar a presença de falhas [Myers]**
- **Pode mostrar a presença de falhas em um sw, mas nunca a sua ausência [Dijkstra]**

(IEEE) processo de execução de um sistema ou componente sob condições especificas para detectar diferenças entre os resultados obtidos e os esperados

☞ Teste não prova que o sw está correto!

Não é objetivo dos testes ...

- Construção e avaliação de protótipos
- Verificação (estática ou dinâmica) de modelos
- Revisão de documentos ou código
- Análise estática automatizada de código
- Análise dinâmica de código visando descobrir problemas tais como vazamento de memória, entre outros
- Depuração (*debugging*) de programas

☞ estas atividades complementam os testes mas não serão consideradas como testes

Princípios [Myers79]

- Dados de teste devem ser definidos para dados válidos, inválidos e inoportunos
- Evite testar seus próprios programas, a menos que seja com auxílio de uma ferramenta
- Determine se o sw faz o que é esperado, mas também se não faz algo indesejável
- Nunca planeje testes assumindo que não serão encontradas falhas

Princípios [Myers79]

- Nunca jogue fora casos de teste, a não ser que vá jogar o sw fora também
- A probabilidade de detectar falhas em uma parte do sw é proporcional ao nº de falhas já detectadas
- Um bom caso de teste é aquele que tem alta probabilidade de detectar novas falhas
- Verifique cuidadosamente os resultados de cada caso de teste
- Testes devem ser planejados desde o início do desenvolvimento

Mais princípios

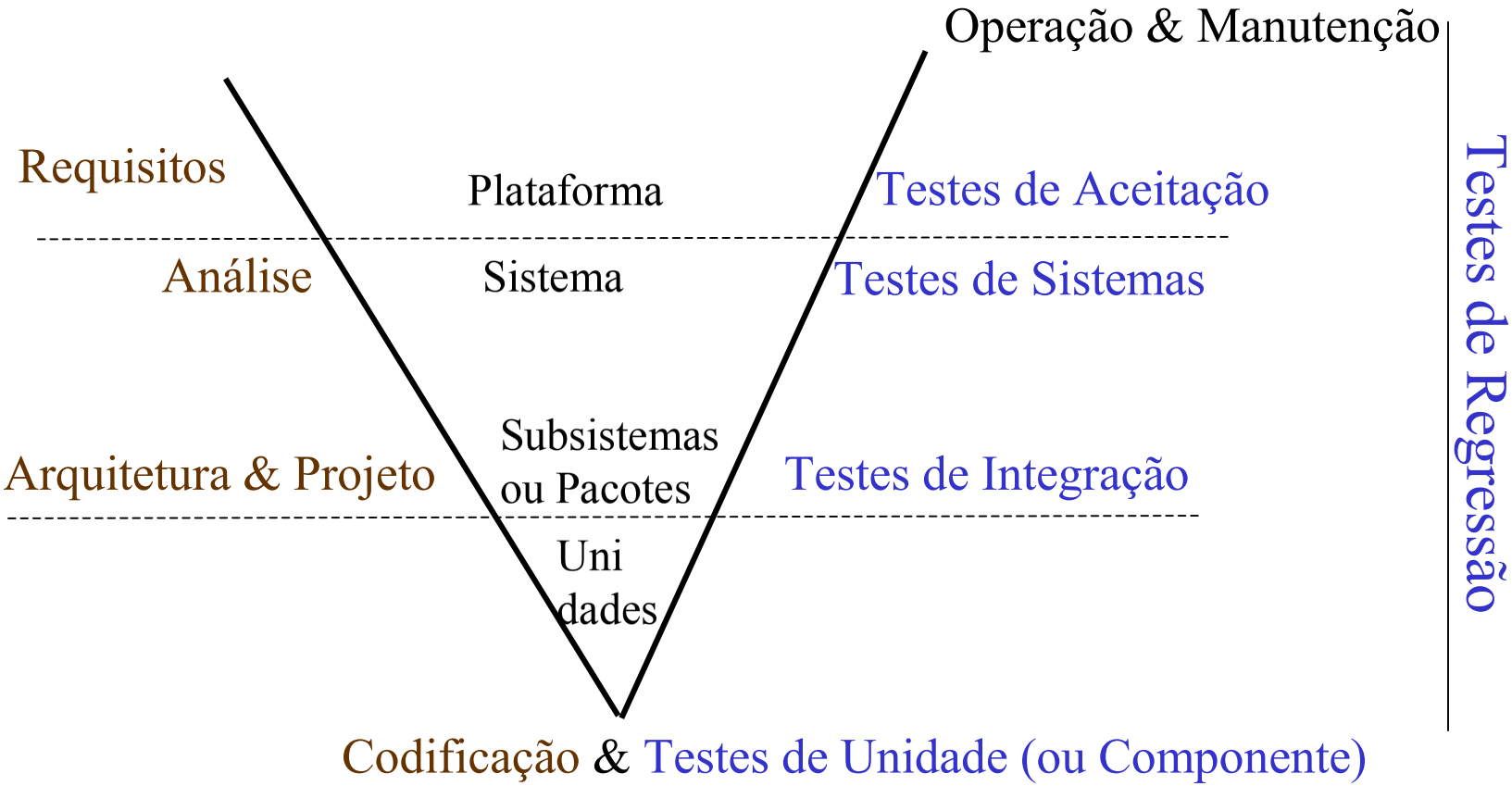
- Casos de teste não são substitutos para a especificação:
  - Casos de teste precisam da especificação para serem construídos
  - Casos de teste precisam da especificação para a análise dos resultados
- Execuções com defeito devem ser convertidas em casos de teste
  - Servirão para testar se as correções apropriadas foram feitas
  - Falhas “ressuscitam”, i.e, podem reaparecer em futuras modificações do software

[B. Meier. Seven principles of software testing. IEEE Computer, Ago/2008, pp99-101]

Quando começam os testes?

- As atividades de teste devem ser iniciadas cedo no ciclo de vida
- As atividades de testes devem ser integradas às atividades de desenvolvimento
- Procedimentos de teste podem ser descritos desde a fase de Especificação do Sistema

Testes no ciclo de vida do software



Escopo dos testes

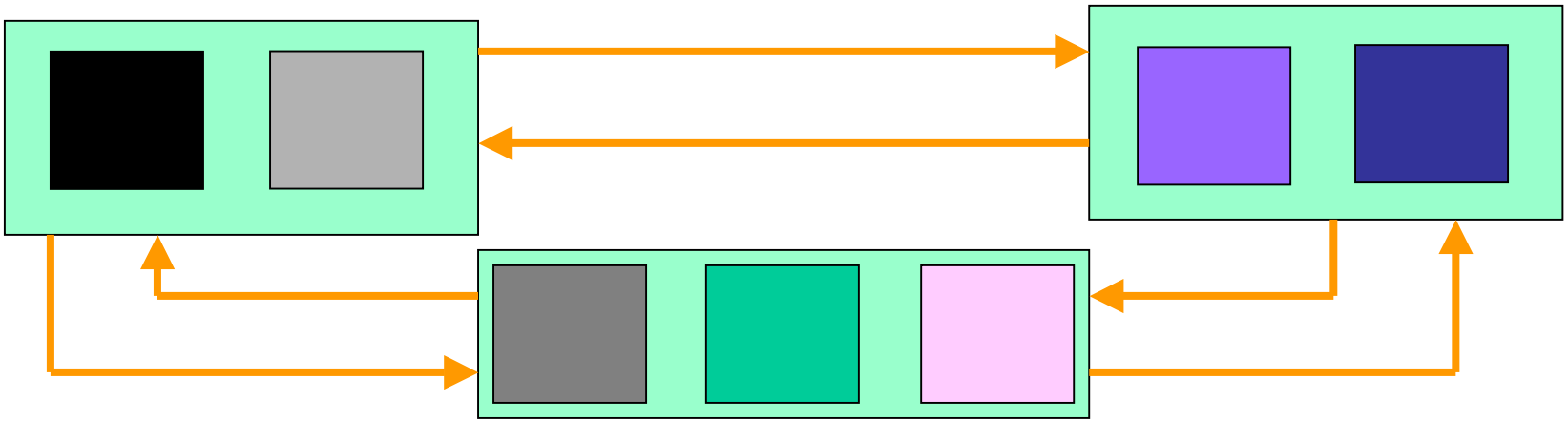
- **Escopo** de testes é a coleção de unidades de interesse
- Uma **unidade** pode ser uma função (ou um método), uma classe, um grupo de funções ou classes, e até mesmo um componente executável com uma interface de programação (API) bem definida
- Dependendo do escopo, os testes podem ser divididos nas seguintes fases:
  - Testes de Unidades
  - Testes de Integração
  - Testes de Sistemas
  - Testes de Aceitação

Testes de Unidades

- Uma unidade pode ser :
  - Módulo ou Função
  - Classe
  - Pequenos grupos de classes (clusters)
  - Componente
  - Um serviço

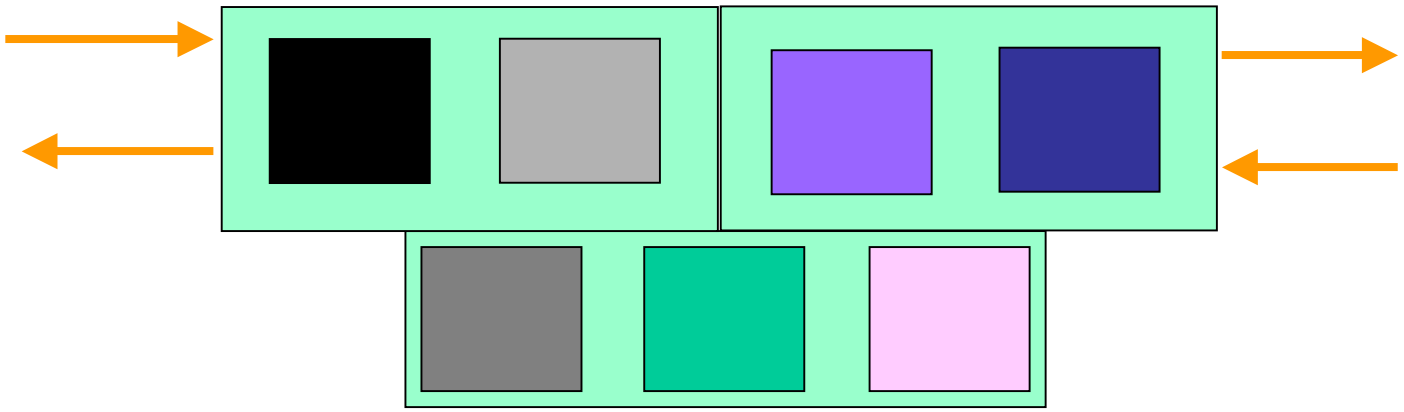
Testes de Integração

- Grandes grupos de classes
- Subsistemas
- Componentes
- Composição de serviços



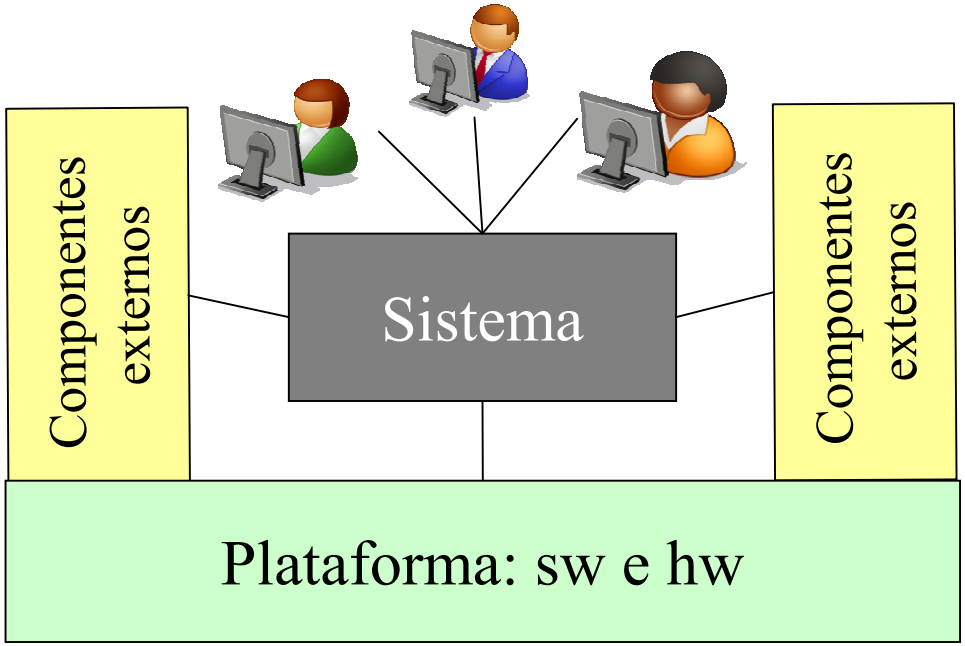
Teste de Sistemas

- Aplicações
- Frameworks



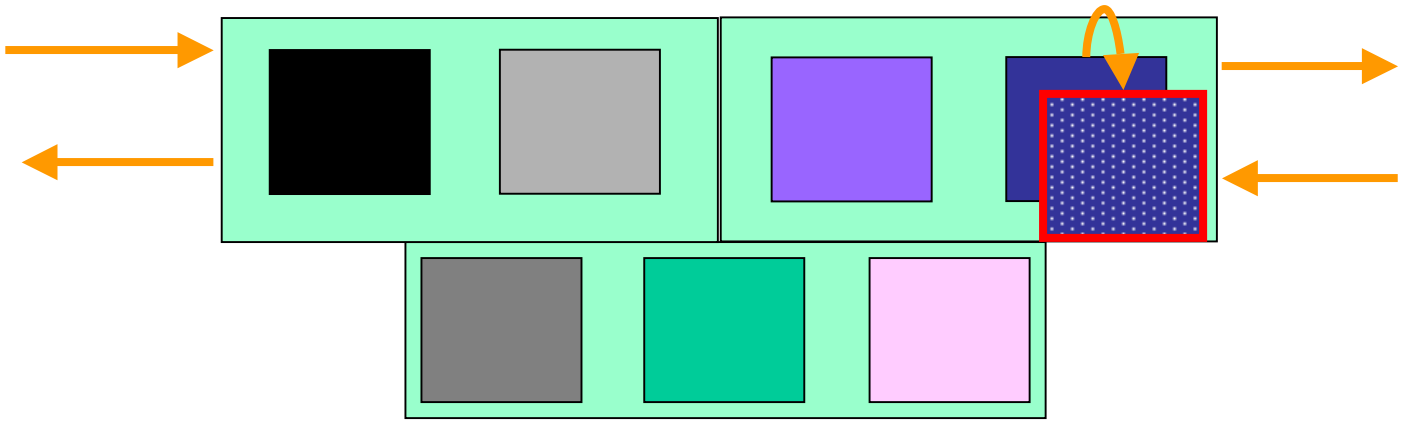
Testes de Aceitação

- Validação do sistema:
  - o sistema está pronto para uso?
  - O sistema atende aos requisitos de operação?

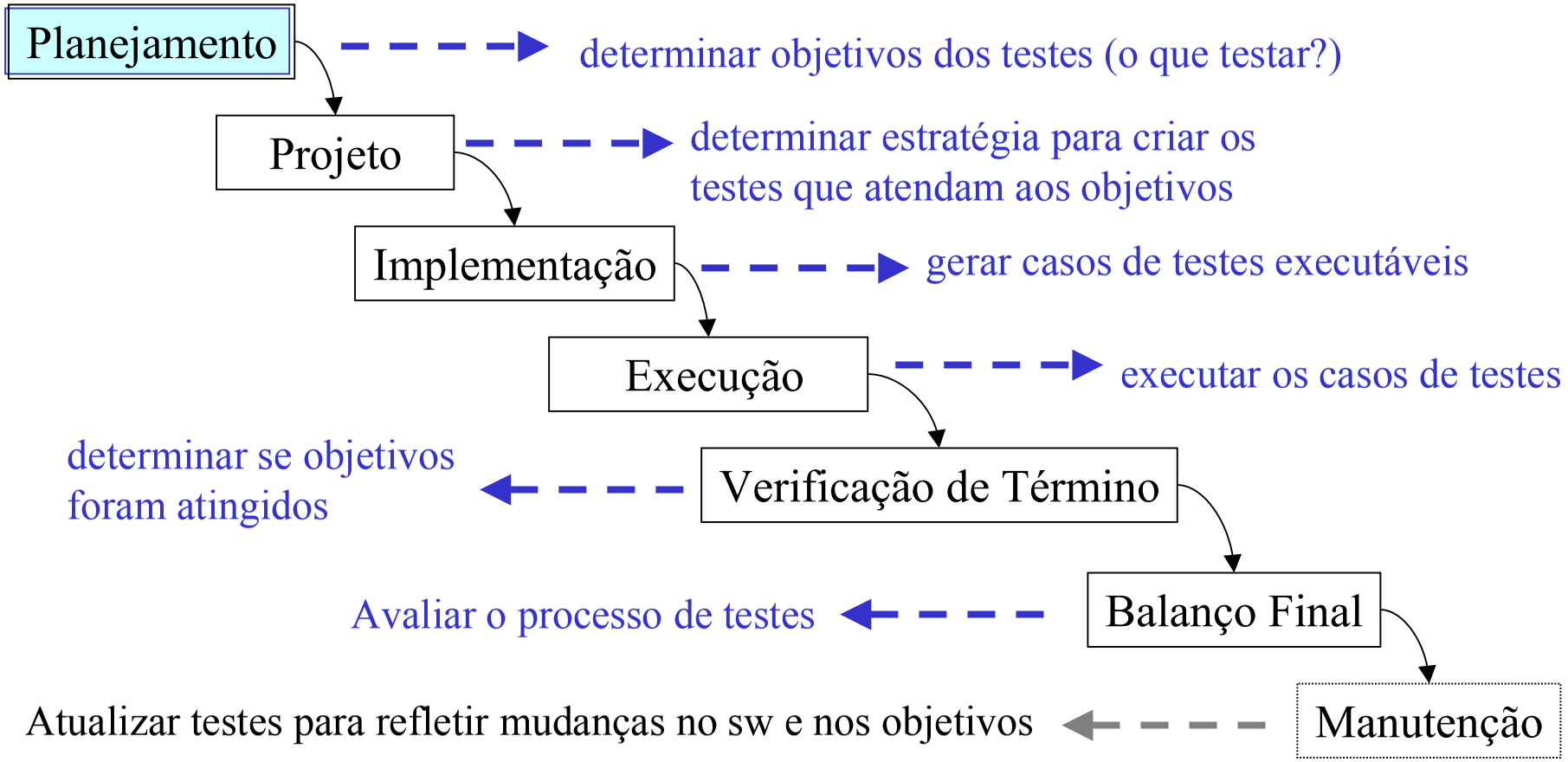


Teste de Regressão

- Verificar Impacto de Mudanças



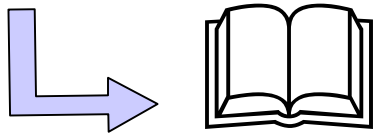
O processo de testes



(baseado em IEEE Std. 1008/87 - Std. for Sw Unit Testing [Paula00])

Planejamento

- Visa responder as seguintes questões:
  - Quem? → equipe de testes; usuários
  - O quê? → requisitos? casos de uso? módulos? ...
  - Quando? → cronograma
  - Onde? → local; ambiente hw e sw
  - Porquê? → critérios de completude
  - Como? → métodos e técnicas



Plano de Teste

Planos de Testes

- Saída da fase de planejamento. Deve-se criar um plano de teste para cada fase:
  - Testes de Unidades
  - Testes de Integração
  - Testes de Sistemas
  - Testes de Aceitação

Decidindo o quê testar

- Dado que **tempo**, **recursos** ou **pessoal** são escassos, os sistemas cada vez mais complexos, como garantir a qualidade mesmo assim? ⇨ testes baseados em riscos
- Realizar análise de riscos para
  - priorizar esforços
  - alocar melhor os recursos

⇩

  - quais componentes devem ser mais cuidadosamente enfocados

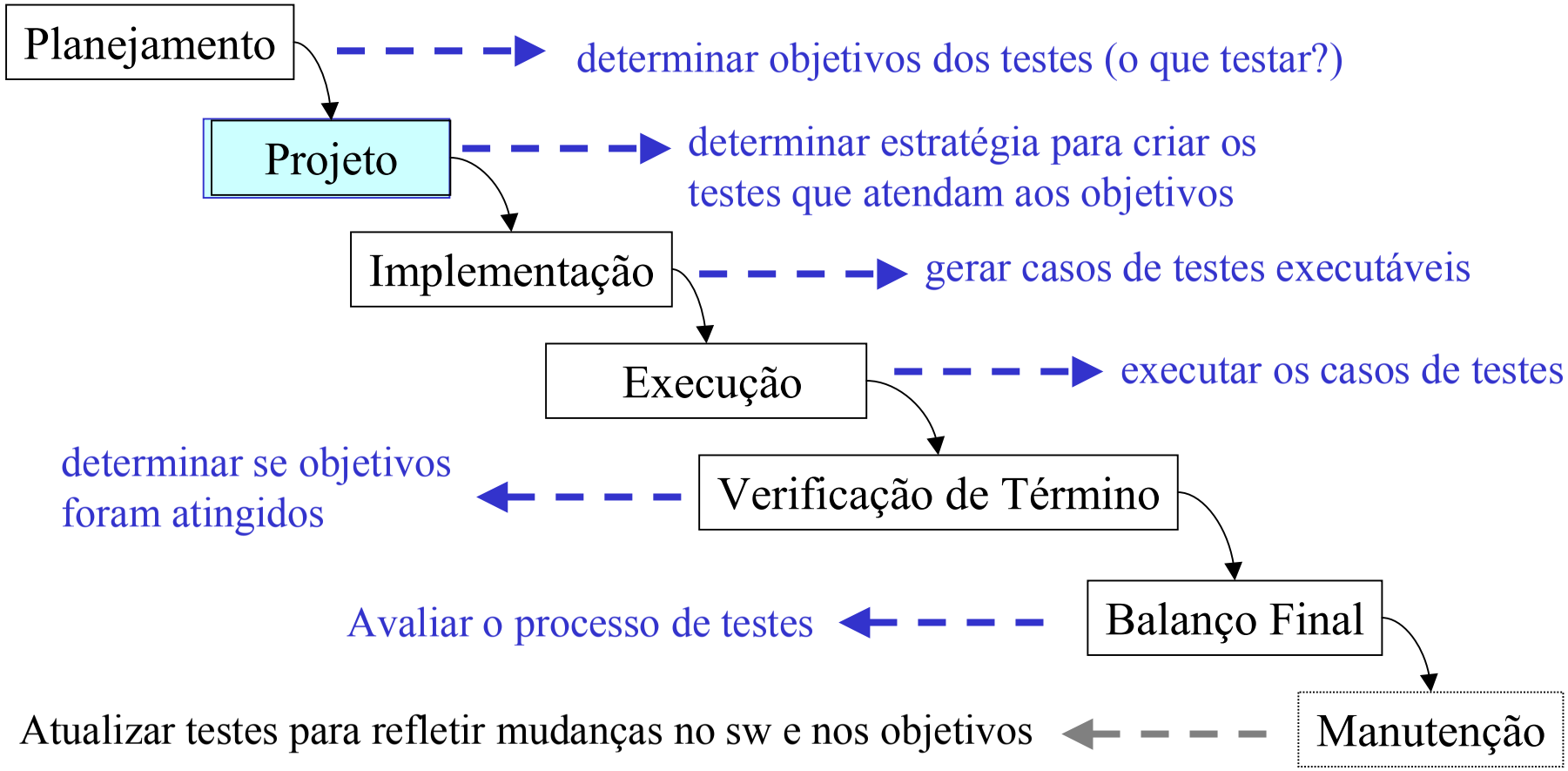
Testes baseados em riscos

- Princípio de Pareto (regra 80/20):
  - Vilfredo Pareto, economista italiano, trabalhando em seu jardim constatou que:
    - 80% das pêras → produzidas por 20% das árvores
    - Será que esse padrão se refletia em outras áreas?
      - 80% das terras → 20% da população
      - 80% dos lucros → 20% dos funcionários
      - 80% dos problemas → 20% dos clientes
      - 80% das decisões → 20% de uma reunião
      - ...

Aplicando regra 80/20 ao software

- No software:
  - 80% dos defeitos são decorrentes de falhas em 20% dos componentes.
  - ☞como achar esses 20% vitais?
    - ❑Resultados de outras atividades de V&V (revisões, ...):
      - ☞Quais os “campeões” de falhas severas ou moderadas??
    - ❑Métricas
      - ☞Que componentes apresentam as piores métricas??

O processo de testes



(baseado em IEEE Std. 1008/87 - Std. for Sw Unit Testing [Paula00])

Projeto de Testes

- Objetivo
  - buscar subconjunto finito de entradas (e estados) que irão **alcançar** e **ativar** as **falhas** (*faults* ou *bugs*) existentes, gerando **erros** que irão se **propagar** até as saídas, **levando à ocorrência** de **defeitos** (***failures***)

- Saída: especificação de testes

Especificação dos testes

- Saída da fase de Projeto de testes, mostrando detalhes sobre os testes a serem realizados
- Deve ficar separada do Plano de Testes para que possa ser reutilizada em diversos planos
- A especificação pode ser textual (testes manuais) ou codificada em alguma linguagem (testes automatizados)

Exemplo de descrição textual

procedimento de testes

Procedimento: Inclusão de Usuário
Identificação: PCT-01
Objetivo Verificar a inclusão de um usuário no bd BD_EX
Fluxo:
1. Abrir interface **Tela de Usuários**
2. Selecionar **Novo**
3. Inserir **Nome, Login, Senha**
4. "Clicar" **Salvar**
5. Selecionar **Pesquisar**

caso de teste

Identificação: CT01
Item a testar: caso de uso IncluirUsuário
Entradas:

Campo	**Valor**
Nome	Usuário1
Login	Usu1
Senha	uuu

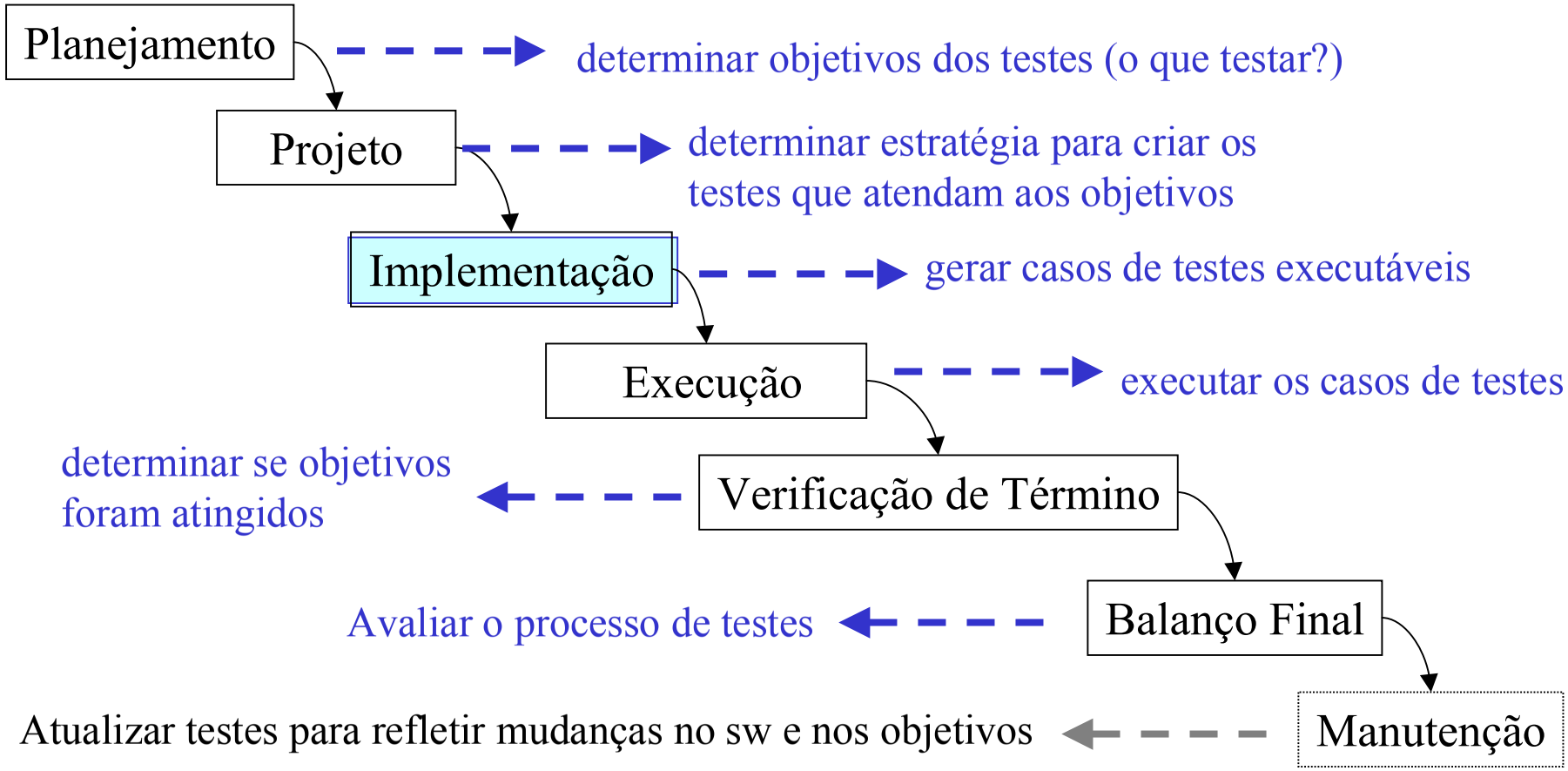
Saídas esperadas:

Campo	**Valor**
Nome	Usuário1
Login	Usu1

Procedimento: PCT-01
Dependências: banco de dados de teste deve estar vazio

[dePaula00]

O processo de testes



(baseado em IEEE Std. 1008/87 - Std. for Sw Unit Testing [Paula00])

Implementação dos testes

- Preparação do ambiente de testes, tornando disponíveis todos os recursos necessários
- Instalação e configuração dos itens a testar
- Instalação e configuração das ferramentas e componentes de teste
- Criação dos casos e procedimentos de testes executáveis, caso sua execução seja automática

Exemplo de implementação de testes

- Uso de JUnit
  - *Framework* que permite
    - Implementar e executar testes de unidade
    - Linguagem JAVA
    - Gratuito, muito utilizado, separa código de teste de código do produto
  - Onde obter mais informações:

    [JUnit] `http://www.junit.org`. Último acesso em 20/06/05.

JUnit

- Exemplo
  - Classe Calculadora.java

```
public class Calculadora {
        private float num1;
        private float num2;
        private float resultado;

        public Calculadora() {
        }

        public float soma(float a, float b) {
                return (a+b);
        }

        public float subtracao(float a, float b) {
                return (a-b);
        }

        public float divisao(float a, float b) {
                return (a/b);
        }

        public float multiplicacao(float a,
            float b) {
                return (a*b);
        }
}
```

JUnit

- Exemplo
  - Criando uma classe de teste

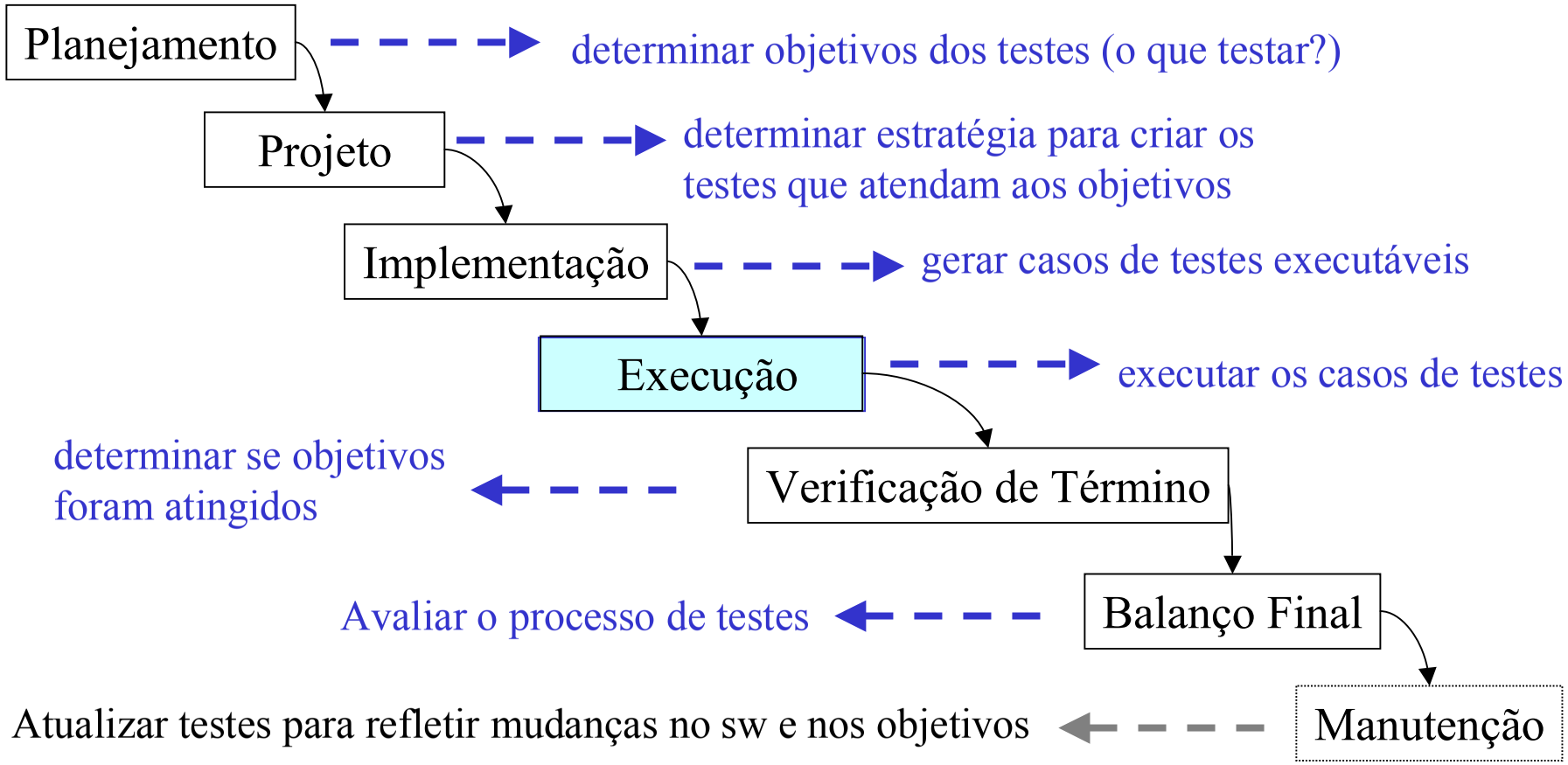
Instancia a classe em teste

Testa um método

Analisa o resultado

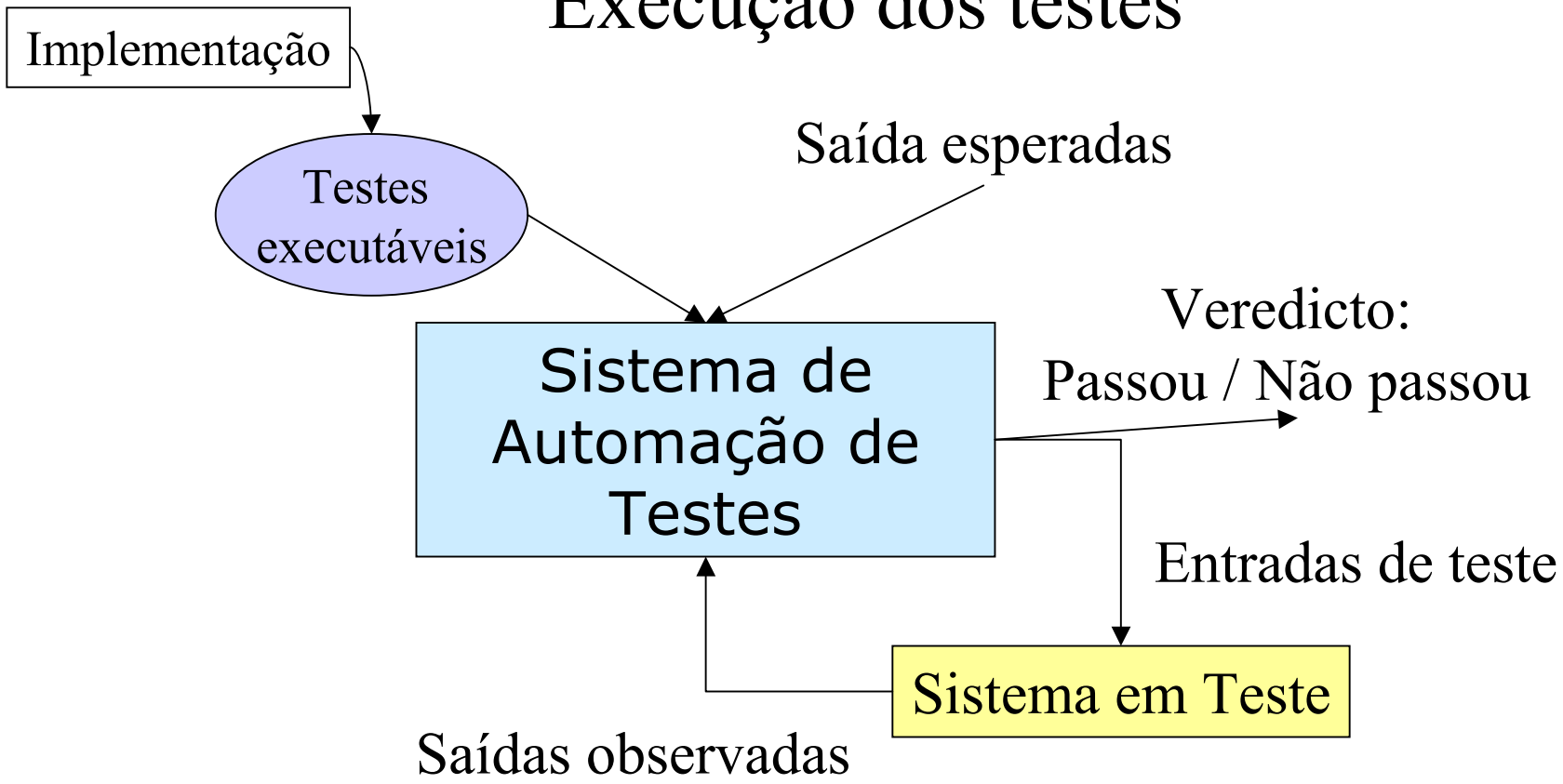
```
import junit.framework.TestCase;
import Calculadora;

public class CalculadoraTest extends TestCase {
        private Calculadora calculadora;

        public void testDivisao() {
                float resultado;
                Calculadora calculadora;

                calculadora = new Calculadora();
                resultado = this.calculadora.
                        divisao(1.0/5.0);
                assertEquals(resultado, 0.2);

        }

        public static void main(String args[]) {
                new CalculadoraTest("testDivisao");
        }

}
```

O processo de testes

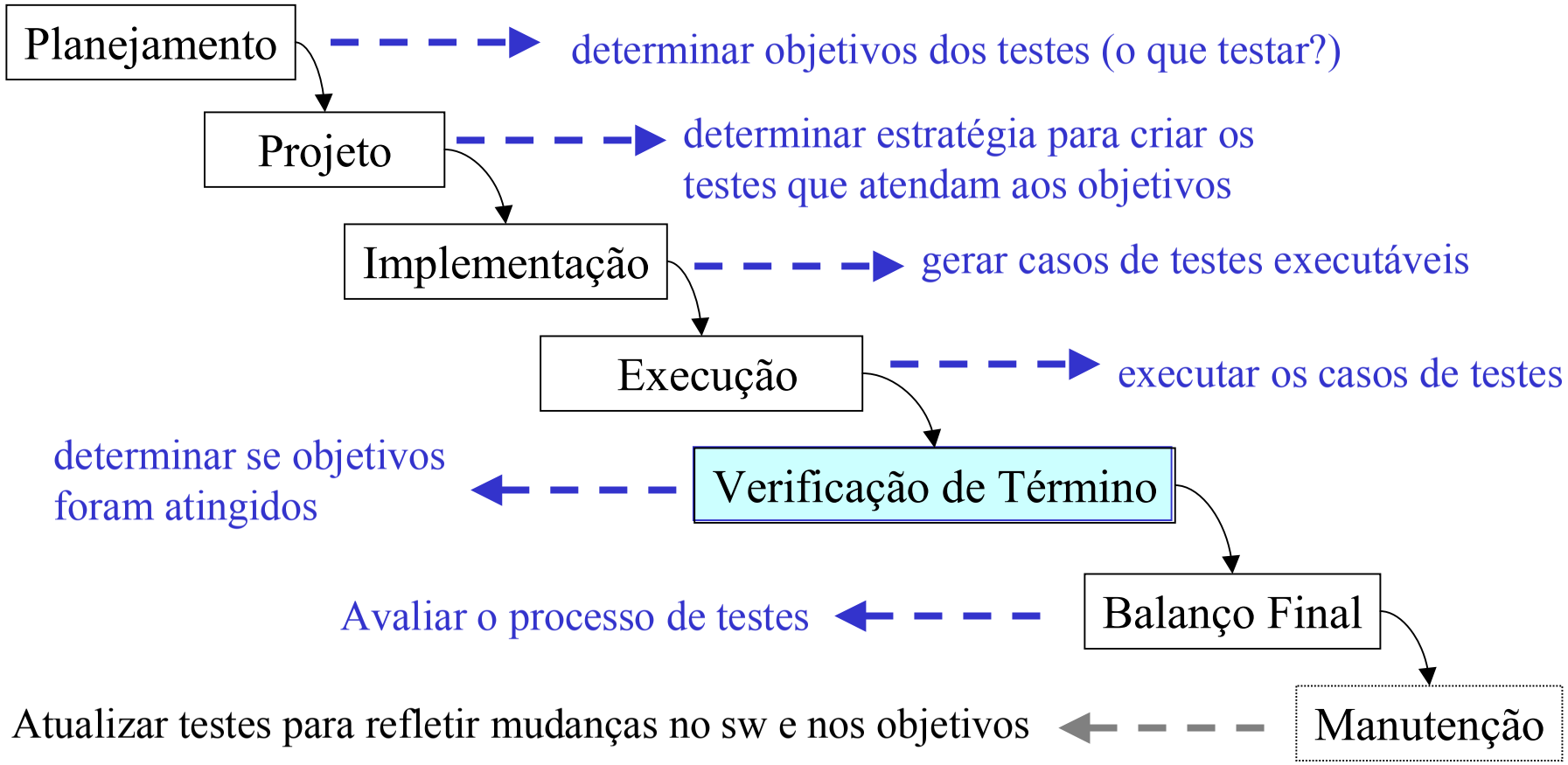


(baseado em IEEE Std. 1008/87 - Std. for Sw Unit Testing [Paula00])

Execução dos testes

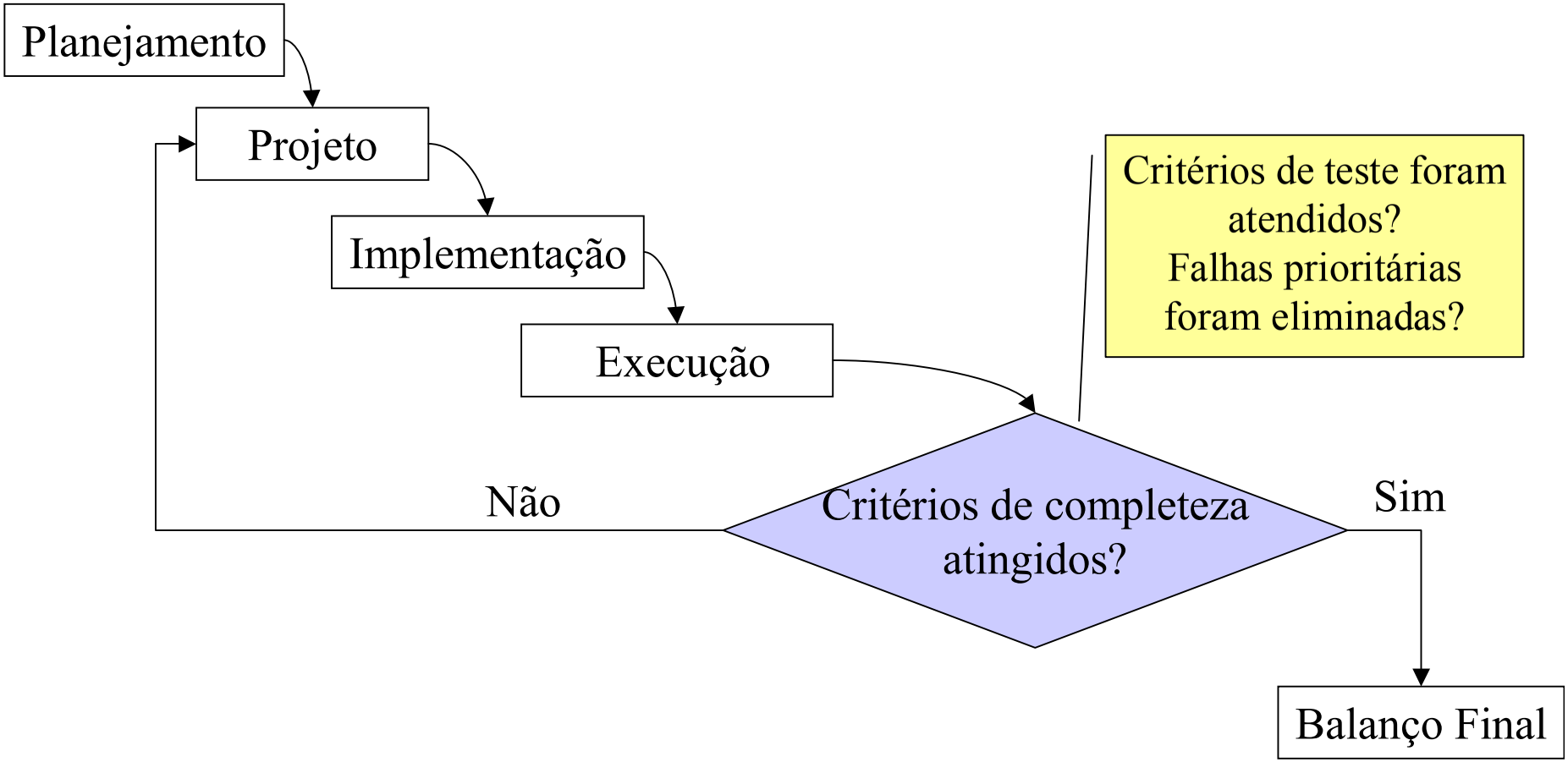


O processo de testes

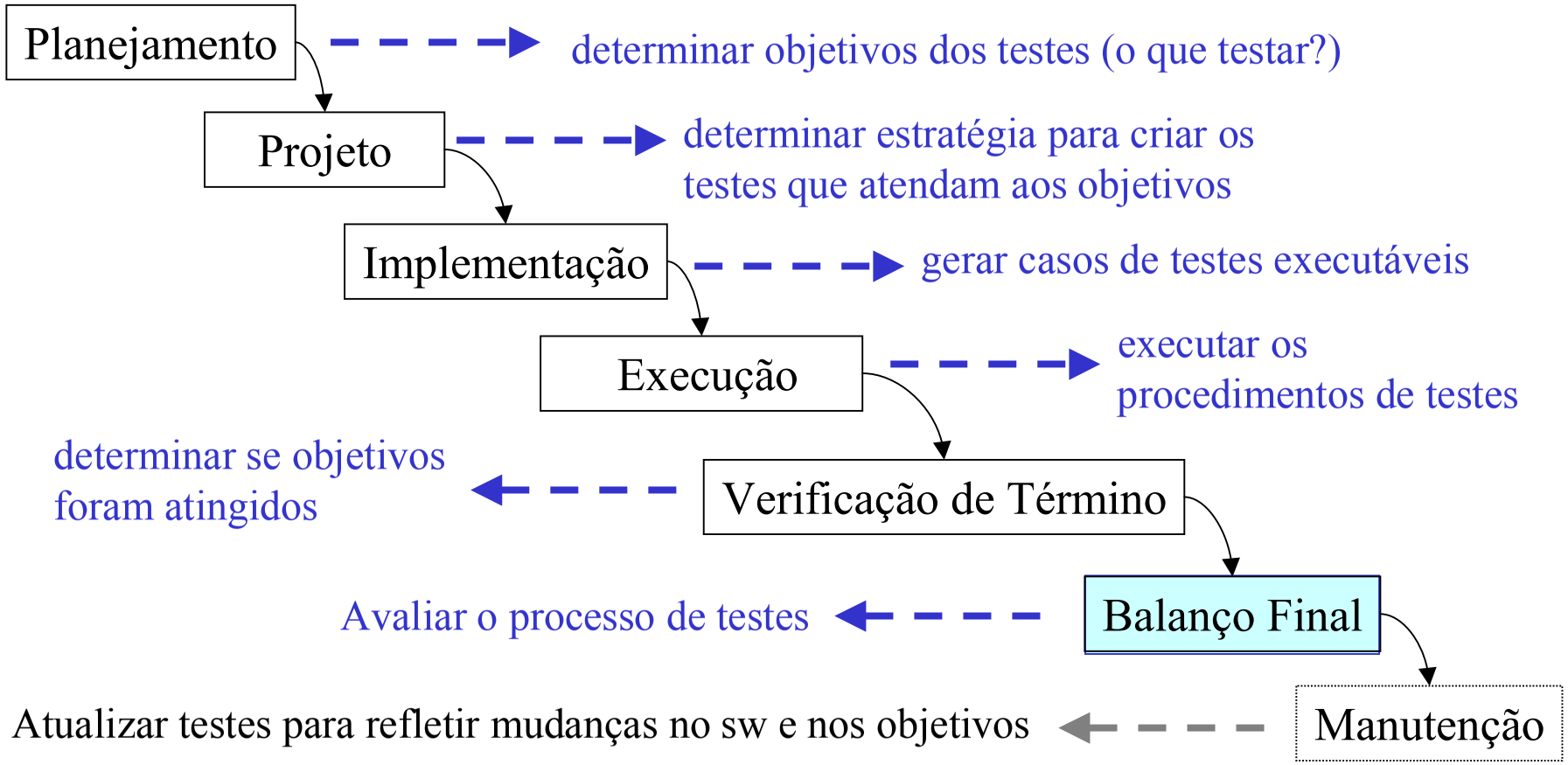


(baseado em IEEE Std. 1008/87 - Std. for Sw Unit Testing [Paula00])

Verificação de término



O processo de testes



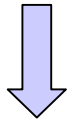
(baseado em IEEE Std. 1008/87 - Std. for Sw Unit Testing [Paula00])

Balanço final

- Obtenção de métricas:
  - Avaliação do produto
  - Avaliação do processo de testes
- Lições aprendidas
- Completar Relatório de Testes
  - Balanço dos defeitos observados
  - Relação entre as falhas corrigidas e os defeitos observados
  - Tempo estimado e real: realização dos testes + correções + retestes
  - Defeitos observados por caso de teste

Balanço final

- Análise de causa-raiz (Root Cause Analysis):
  - Quais as causas das falhas encontradas?
  - É possível evitar que elas voltem a aparecer em próximos projetos?
  - É possível detectá-las mais cedo?



Melhoria do processo

Testar não é tudo

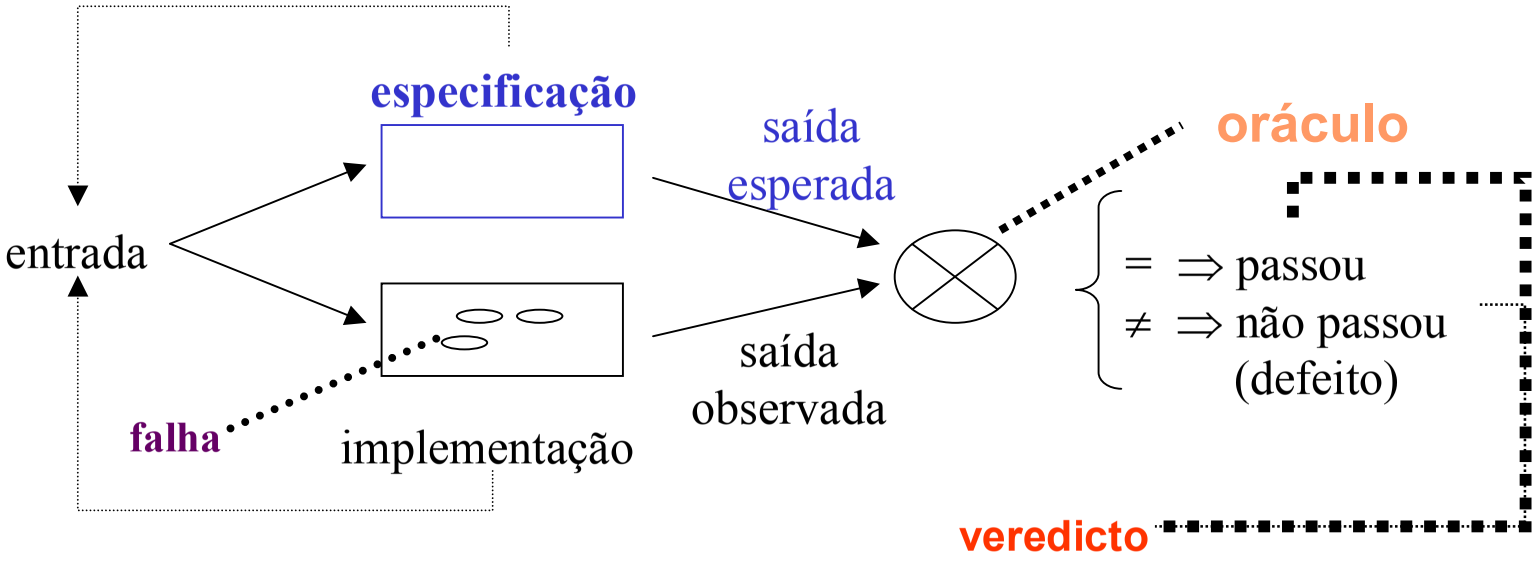
- Testar não é a única forma de detectar falhas em um sw
  - testes devem complementar outras formas de V&V e não substituí-las
  - há falhas que dificilmente seriam reveladas através de testes
    ex.: vazamento de memória (*memory leak*), baixa usabilidade, *deadlock*

- "Leis de Beizer" [Beizer90] :
  - **Paradoxo do Pesticida**: cada método/técnica usada deixa falhas residuais que não são detectáveis por esse método/técnica
  - **Barreira da Complexidade**: a complexidade do sw (e das falhas que este contém) cresce até um limite em que não podemos mais controlá-lo

Projeto de testes: Introdução

Objetivo

- Buscar as entradas (e estados) que irão **alcançar** e **ativar** as **falhas** (bugs) existentes, gerando **erros** que irão se **propagar** até as saídas, **levando à ocorrência** de **defeitos (*failures*)**



Como alcançar estes objetivos?

- Para observar defeitos nos testes, 3 condições são necessárias:
  - **Alcançabilidade**: o local com a falha deve ser alcançada durante a execução
  - **Infecção**: a falha é ativada, gerando um erro, i.e, estado incorreto do programa
  - **Propagação**: o estado incorreto deve se propagar até que o programa gere algumas saídas incorretas

Exemplo

- Suponha o seguinte trecho de código:

```
read (x)
y = x + x                    ➲ deveria ser y = x * x
if (y > x) then write ("o quadrado de x é maior
  que x")
          else write ("o quadrado de x não é
  maior que x");
```

para que valores de x a falha seria revelada?

Exemplo 1

☞Falha!

```
read (x)
y = x + x
  ➲ deveria ser y = x * x
if (y > x) then write ("o
  quadrado de x é maior
  que x")
          else write ("o
  quadrado de x não é
  maior que x");
```

x	y
1	***

Entrada: x = 1
Saída esperada: **"o quadrado de x não é maior que x"**

Exemplo 2

☞Falha!

```
read (x)
y = x + x
  ➲ deveria ser y = x * x
if (y > x) then write ("o
  quadrado de x é maior
  que x")
          else write ("o
  quadrado de x não é
  maior que x");
```

Alcançou instrução com falha

x	y
1	***
1	2

☞Erro!
Infecção

Entrada: x = 1
Saída esperada: **"o quadrado de x não é maior que x"**

Exemplo 3

☞Falha!

```
read (x)
y = x + x
  ➲ deveria ser y = x * x
if (y > x) then write ("o
  quadrado de x é maior
  que x")
        else write ("o
  quadrado de x não é
  maior que x");
```

x	y
1	***
1	2

Alcançou instrução com falha

☞Erro!

Infecção

Propagação do erro

Saída: **"o quadrado de x é maior que x"**

Defeito!

Entrada: x = 1

Saída esperada: **"o quadrado de x não é maior que x"**

Exercício

```
public static int
   OddOrPos(int[] x){
//retorna o nro. de
// elementos de x que são
// ímpares ou positivos
  int count=0;
  for (int i=0; i<x.length;
   i++)
  {
    if (x[i]%2==1||x[i]>0)
     { count++ }
  }
  return count;
}
```

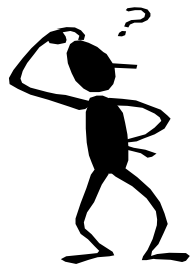
```
Teste: x = [-3,-2,0,1,4]
Saída esperada: 3
```

- Identifique a falha do programa.
- Ache um caso de teste que não encontre a falha.
- Se possível, encontre um caso de teste que encontre a falha mas que não produza um erro
- Se possível, encontra um caso de teste que produza o erro mas não leve a um defeito. (não se esqueça do contador de programa ou PC)
- Identifique o estado errôneo produzido pelo caso de teste dado (não esqueça o PC)

Projeto de Testes - Objetivo

- Objetivo
  - buscar **subconjunto finito** de entradas (e estados) que irão **alcançar** e **ativar** as **falhas** (bugs) existentes, gerando **erros** que irão se **propagar** até as saídas, **levando à ocorrência** de **defeitos**

- Dificuldade
  - como encontrar esse **subconjunto finito** ?



Ideal

- Dado que testes são úteis quando revelam a presença de falhas ...
- Dado que quanto mais se executar um software mais chances se tem de ativar as falhas ...

Quanto mais exaustivos os testes, mais falhas podem ser encontradas, certo?

Sobre a impossibilidade de testes exaustivos (1)

- Teste exaustivo (ou completo) consiste em ...
  ① *Exercitar o componente em teste (CeT) diante de todas as combinações possíveis de entradas?*

  – O nº de combinações de entradas de um sw pode ser enorme:

    ex.: um programa que lê uma cadeia de 10 caracteres requer 26^{10} combinações!

Sobre a impossibilidade de testes exaustivos (2)

- Teste exaustivo consiste em ...

  ② *Exercitar todos os caminhos de execução possíveis?*

- **caminho de execução** = seqüência de instruções começando em um ponto de entrada e terminando em um ponto de saída.
  - Possível se o programa não tem *loops* (explícitos ou implícitos), senão, o nº de seqüências de execução pode ser muito grande ou infinito

    ex.: o trecho de código abaixo :

```
for ( int i = 0; i < n; ++i ) {
  if ( x[i]div 2 == 0)     somapar = somapar + x [i]
  else somaimpar = somaimpar + x [i];
   }
```

possui $2^n + 1$ (para n> 0) seqüências possíveis de execução, considerando as repetições

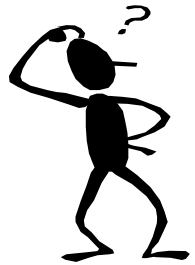
Projeto de Testes - Solução

- Objetivo
  - buscar **subconjunto finito** de entradas (e estados) que irão **alcançar** e **ativar** as **falhas** (bugs) existentes, gerando **erros** que irão se **propagar** até as saídas, **levando à ocorrência** de **defeitos**

- Dificuldade
  - como encontrar esse **subconjunto finito** ?

⇩

  - Uso de modelos
  - Uso de critérios para a seleção de testes



Uso de um modelo

- Modelos de desenvolvimento [Siegel96]
  - Requisitos: modelam o problema
  - Projeto: modelam a solução para o problema
  - Código fonte: modelam a implementação da solução
- Falhas
  - hipóteses sobre falhas prováveis (de sw ou de hw)

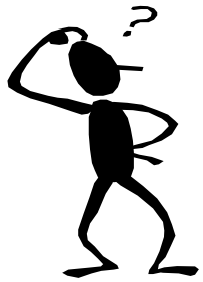
Modelos de base

Classificação das técnicas de projeto de testes

Modelo de base	Modelo de teste	Técnica de teste
Especificação de Requisitos de Projeto	Modelo do sistema ou componente	Baseado na especificação (caixa preta)
Código	Modelo do programa	Baseado na implementação (caixa branca)
Falhas	Modelo de Falhas	Baseado em falhas

Mais algumas questões na geração de testes

- Como selecionar as entradas a serem usadas nos testes ?



- Como decidir se um (componente de) sw já foi suficientemente testado ?

Critérios de teste

- Um **critério** de teste **C** determina um conjunto finito de elementos do modelo de teste que devem ser exercitados durante os testes
- Requisito de teste ($\mathbf{RT_C}$): elemento do modelo de teste que precisa ser coberto

Ex.:

modelo de teste = diagrama de casos de uso

critério (**C**) = todos os casos de uso devem ser exercitados ao menos 1 vez

requisitos de teste ($\mathbf{RT_C}$) = { todos os casos de uso do diagrama }

↳ também chamados de **elementos requeridos**

Cobertura de testes

- Medida de qualidade dos testes:
  - mede o quão completo é um conjunto de testes (***T***) com relação ao critério adotado
  - é dada na forma de uma proporção:

$$\frac{\text{nº de requisitos de teste } (RT_C) \text{ exercitados}}{\text{nº total } RT_C}$$

- O nível cobertura permite responder à questão: “quando terminam os testes?”
  Porque?

Sumário: principais pontos