Seção A: Introdução

Tutorial: Geração de Dados de Teste
(Principais Conceitos e Técnicas)

Tutorialista: Profa. Silvia Regina Vergilio

Agenda da Seção A

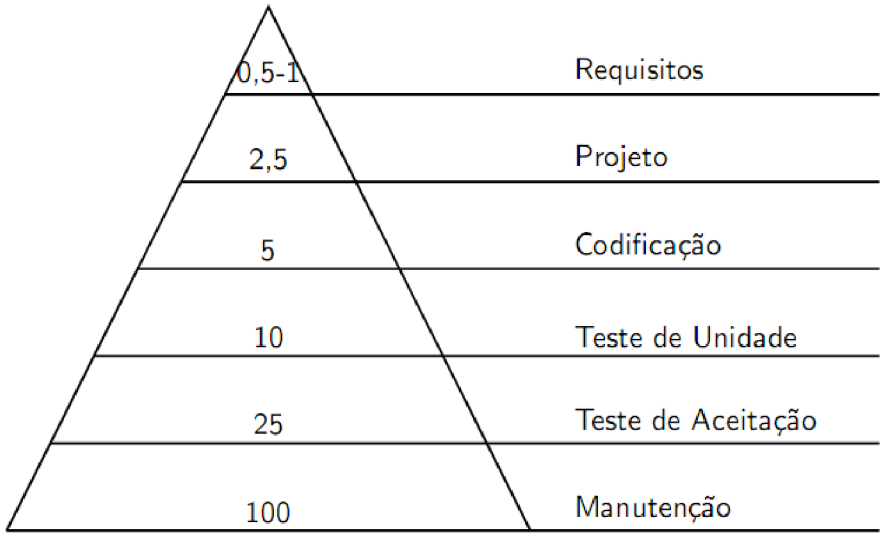
- Por que testar?
- O que é testar?
- Fases de teste
- Terminologia
- Partes de um caso de teste
- Conjunto de Teste
- Técnicas e critérios de teste
  - Critérios funcionais
  - Critérios estruturais
  - Critérios baseado em defeitos

Por que Testar? (1)

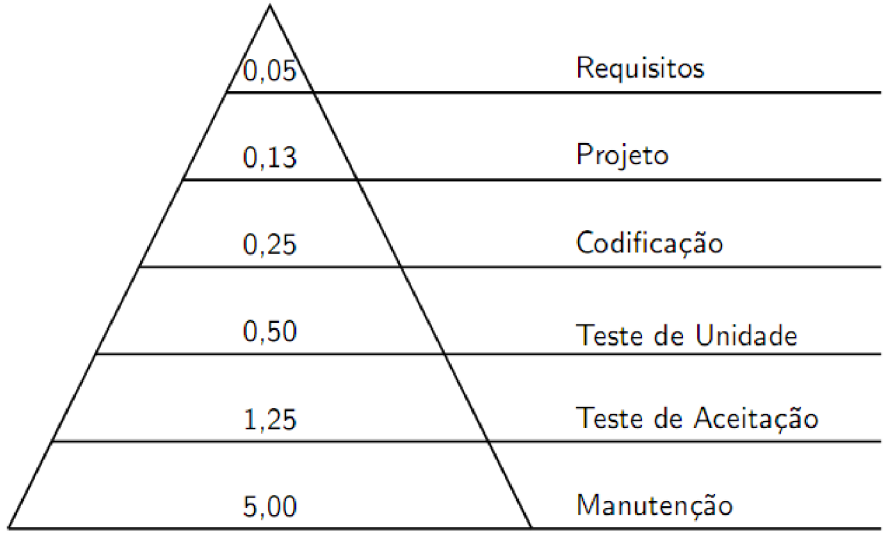
- Crescente interesse e importância do teste de software, principalmente devido a demanda por produtos de software de alta qualidade.
- Shull et al. (2002) alerta que quase não existem módulos livres de defeitos durante o desenvolvimento, e após a liberação dos mesmos, em torno de 40% podem estar livres de defeitos.
- Boehm e Basili (2001) também apontam que é quase improvável liberar um produto de software livre de defeitos.
- Além disso, quanto mais tarde um defeito é revelado, maior será o custo para sua correção.

Por que Testar? (2)

- Escala de Custo de Correção de Defeitos



Fonte: Boehm (1987)



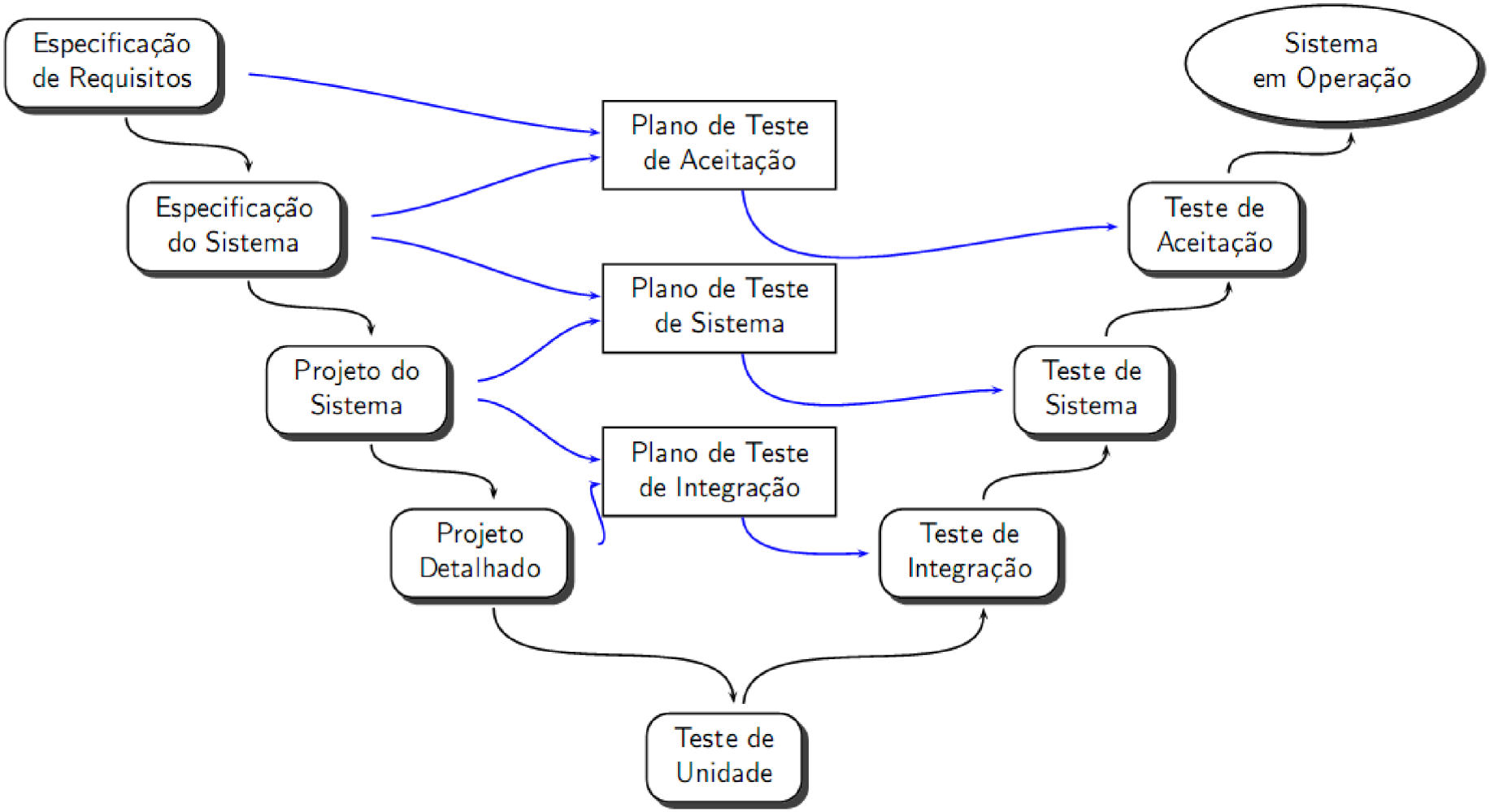
Fonte: Boehm e Basili (2001)

O que é testar?

- Segundo o *IEEE Standard Glossary* 610.12-1990 (R2002) (IEEE, 2002):
  - "Teste é o processo de operar um sistema ou componente sob condições específicas, observando e registrando os resultados, avaliando alguns aspectos do sistema ou componente."

- Craig e Jaskiel (2002) apresentam outra definição:
  - "Teste é um processo de engenharia concorrente ao processo de ciclo de vida do software, que faz uso e mantém artefatos de teste usados para medir e melhorar a qualidade do produto de software sendo testado."

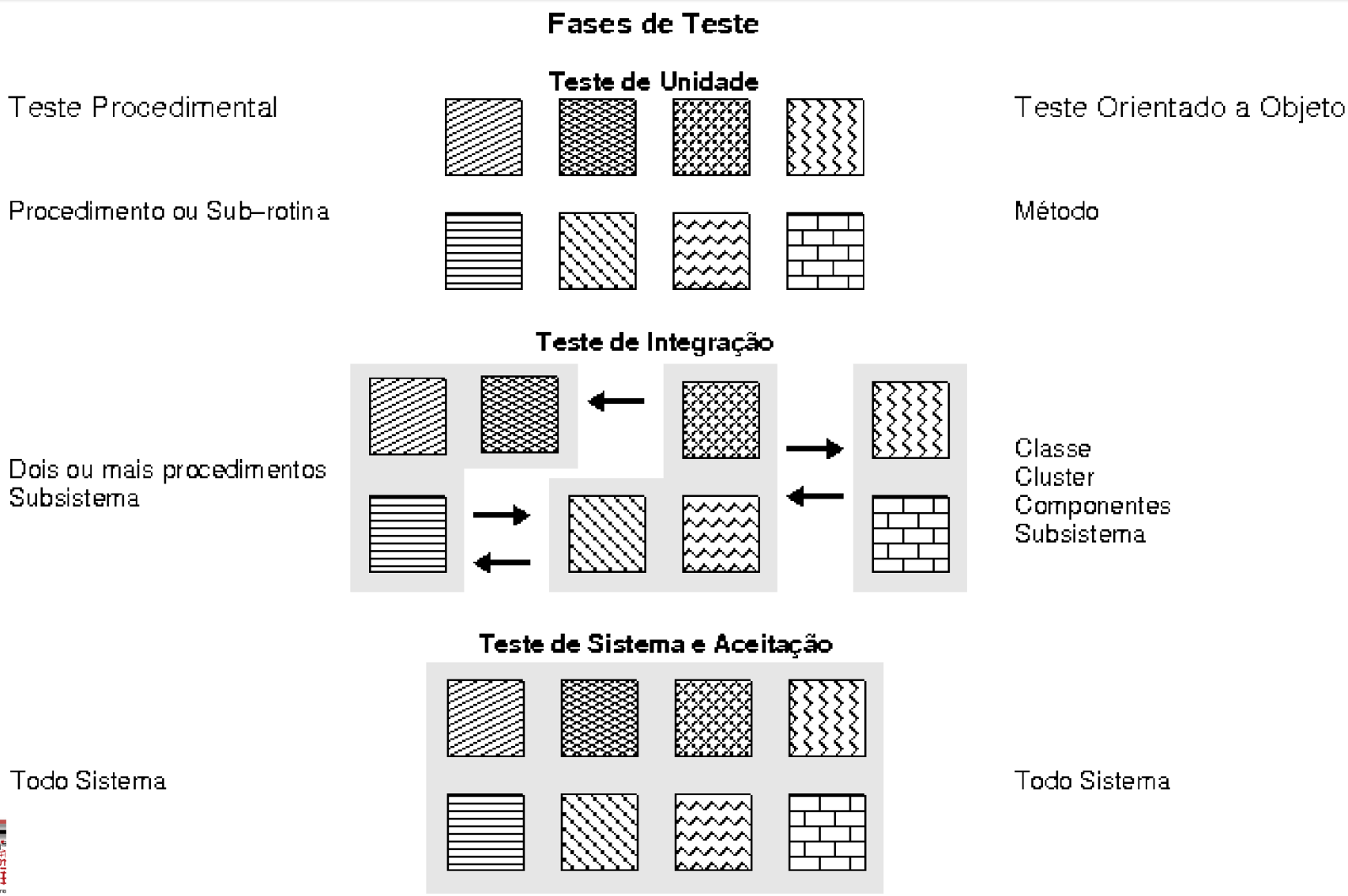
Modelo V


Especificação de Requisitos
Sistema em Operação
Plano de Teste de Aceitação
Especificação do Sistema
Teste de Aceitação
Plano de Teste de Sistema
Projeto do Sistema
Teste de Sistema
Plano de Teste de Integração
Projeto Detalhado
Teste de Integração
Teste de Unidade

Fases de Teste (1)

- A Atividade de Teste também é dividida em fases, conforme outras atividades de Engenharia de Software.
- Objetivo é reduzir a complexidade dos testes.
- Conceito de “dividir e conquistar”.
- Começar testar a menor unidade executável até atingir o programa como um todo.

Fases de Teste (2)



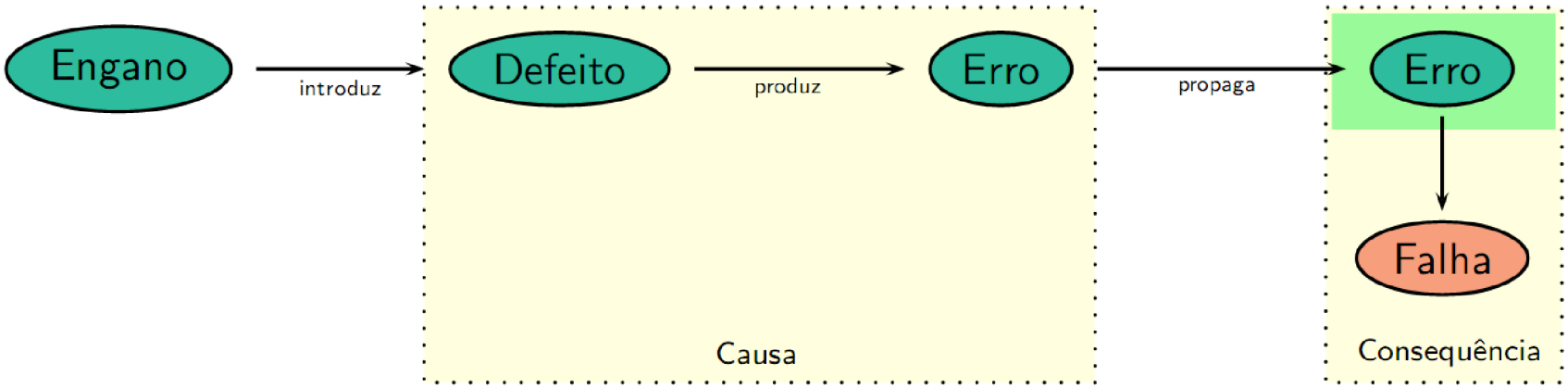
Terminologia (1)

- O *IEEE Standard Glossary* 610.12-1990 (R2002) (IEEE, 2002) diferencia os seguintes termos:

Engano (*Mistake*)	• Ação humana que produz um resultado incorreto. Exemplo: uma ação incorreta tomada pelo programador.
Defeito (*Fault*)	• Um passo, processo, ou definição de dados incorreta em um produto de software. No uso comum, os termos "erro", "bug", e "defeito" são usados para expressar esse significado. Exemplo: um comando ou instrução incorreta.
Erro (*Error*)	• Diferença entre o valor computado, observado ou medido e o valor teoricamente correto de acordo com a especificação.
Falha (*Failure*)	• Inabilidade do sistema ou componente realizar a função requerida, considerando as questões de desempenho exigidas.

Terminologia (2)

- Relação dos termos na terminologia padrão



Terminologia (3)

- Considere o seguinte comando de atribuição: (z = y + x)

1. **Engano** – o comando é modificado para (z = y − x), caracterizando um defeito.
2. **Defeito** – se ativado (executado) com x = 0, nenhum erro é produzido. Para valores de x ≠ 0, o defeito é ativado causando um erro na variável z.
3. **Erro** – o estado errôneo do sistema, quando propagado até a saída, causará uma **Falha**.

Partes de Um Caso de Teste (1)

- De forma simplificada, um caso de teste é um par ordenado formado por:
  - Dado de teste
  - Saída esperada

- O dado de teste corresponde às entradas necessárias para ocasionar a execução do produto em teste
- A saída esperada corresponde ao resultado que o produto em teste deveria produzir conforme a especificação, considerando a entrada fornecida

Partes de Um Caso de Teste (2)

- Mais formalmente
  - A tupla (d, $S(d)$) é um caso de teste, onde $d \in D$ é a entrada (dado de teste) e $S(d)$ representa a saída esperada de d de acordo com a especificação S.

Partes de Um Caso de Teste (3)

- **Entrada** (dado de teste)
  - Geralmente identificadas como dados fornecidos via teclado para o programa executar.
  - Entretanto, os dados de entrada podem ser fornecidos por outros meios, tais como:
    - Dados oriundos de outro sistema que servem de entrada para o programa em teste;
    - Dados fornecidos por outro dispositivo;
    - Dados lidos de arquivos ou banco de dados;
    - O estado do sistema quando os dados são recebidos;
    - O ambiente no qual o programa está executando.

Partes de Um Caso de Teste (4)

Saída esperada

- As saídas também podem ser produzidas de diferentes formas.
- A mais comum é aquele apresentada na tela do computador.
- Além dessa, as saídas podem ser enviadas para:
  - Outro sistema interagindo com o programa em teste;
  - Dados escritos em arquivos ou banco de dados;
  - O estado do sistema ou o ambiente de execução podem ser alterados durante a execução do programa.

Partes de Um Caso de Teste (5)

Oráculo

- Todas as formas de entrada e saída são relevantes.
- Durante o projeto de um caso de teste, determinar a saída esperada é função do oráculo (*oracle*);
- Oráculo corresponde a um mecanismo (programa, processo ou dados) que indica ao projetista de casos de testes se a saída obtida para um dado de teste é aceitável ou não.

Conjunto de Teste

- O termo **conjunto de teste** é usado para definir um conjunto de casos de teste.
  - O conjunto formado apenas por dados de teste é referenciado como **conjunto de dados de teste**.

Técnicas e Critérios de Teste (1)

- Basicamente, os testes podem ser classificados em:
  - **teste caixa-preta** (*black-box testing*),
  - **teste caixa-branca** (*white-box testing*), ou
  - **teste baseado em defeito** (*fault-based testing*).

- Esses tipos de teste correspondem às chamadas **técnicas de teste**.

Técnicas e Critérios de Teste (2)

- A técnica de teste é definida pelo tipo de informação utilizada para realizar o teste:

Técnica Caixa-Preta	• Os testes são baseados exclusivamente na especificação de requisitos do programa. Nenhum conhecimento de como o programa está implementado é requerido.
Técnica Caixa-Branca	• Os testes são baseados na estrutura interna do programa, ou seja, na implementação do mesmo.
Técnica Baseada em Defeitos	• Os testes são baseados em informações histórica sobre defeitos cometidos frequentemente durante o processo de desenvolvimento de software.

Técnicas e Critérios de Teste (3)

- A cada técnica de teste estão associados diferentes **critérios de teste**.
- Os critérios de teste:
  - Auxiliam a selecionar dentre um conjunto de entradas potencialmente infinito os "melhores" pontos, ou seja, os que têm maior probabilidade de revelar o maior número de defeitos com menor custo.
  - Sistematizam a forma como **requisitos de teste** devem ser produzidos a partir da fonte de informação disponível (especificação de requisitos, código fonte, histórico de defeitos, dentre outras).

Técnicas e Critérios de Teste (4)

- Os requisitos de teste são utilizados para:
  - Gerar casos de teste; e/ou
  - Avaliar a qualidade de um conjunto de teste existente.

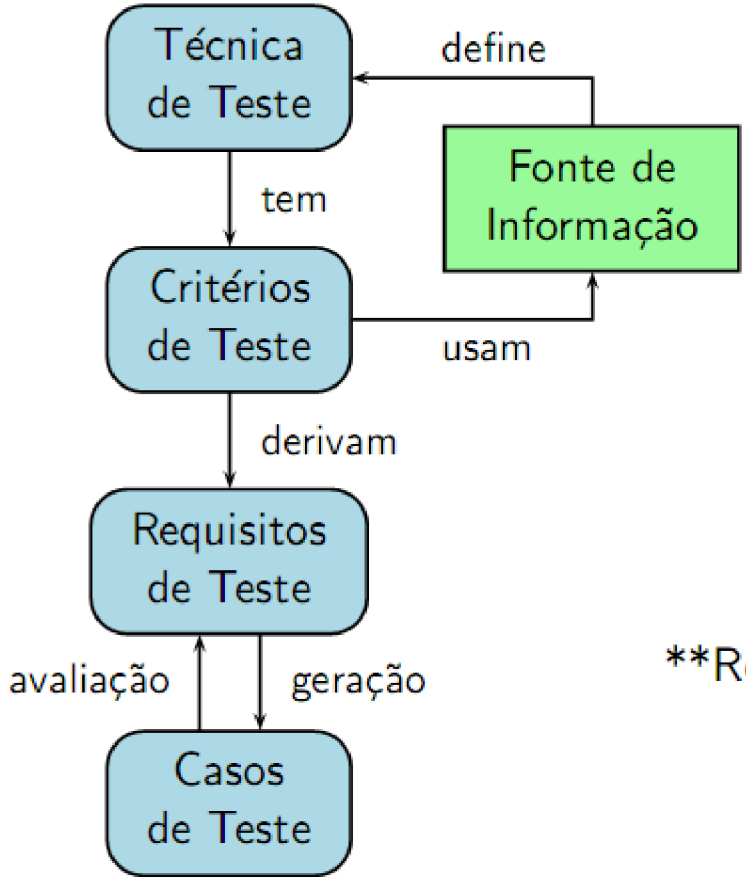
- Critérios de teste ajudam a decidir quando parar os testes.

Técnicas e Critérios de Teste (5)

- Dado um produto *P* e um conjunto de teste *T*, um critério de teste é definido como um:
  - Critério de adequação de caso de teste: predicado que avalia o quanto *T* é adequado ao teste de *P*;
  - Método de Seleção de caso de teste: procedimento para escolher casos de teste para testar *P*.

Critério de teste é "um predicado que define quais propriedades de um programa devem ser exercitadas visando obter um teste sistemático" (Goodenough e Gerhart, 1975).

Técnicas e Critérios de Teste (6)



**Requisitos não-executáveis.

Critérios Funcionais (1)

- Os passos básicos para se aplicar um critério de teste funcional é o seguinte:

A especificação de requisitos é analisada;

↓

Entradas válidas são escolhidas (com base na especificação) para determinar se o produto em teste se comporta corretamente. **Entradas inválidas** também são escolhidas para verificar se são detectadas e tratadas adequadamente;

↓

As saídas esperadas para as entradas escolhidas são determinadas;

↓

Os casos de testes são construídos;

↓

O conjunto de teste é executado;

↓

As saídas obtidas são comparadas com as saídas esperadas;

↓

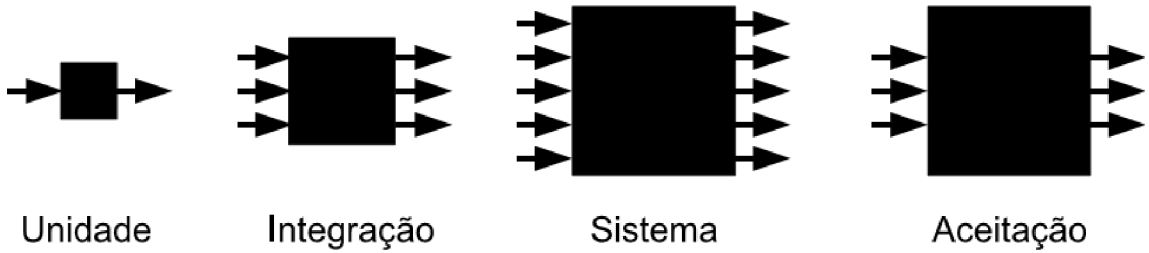
Um relatório é gerado para avaliar o resultado dos testes.

Critérios Funcionais (2)

- Alguns critérios funcionais:
  - Particionamento de Equivalência (*Equivalence Partition*);
  - Análise do Valor Limite (*Boundary Value Analysis*);
  - Tabela de Decisão (*Decision Table*);
  - Teste de Transição de Estado (*State-Transition Testing*);
  - Teste de Análise de Domínio (*Domain Analysis Testing*);
  - Teste de Caso de Uso (*Use Case Testing*).

Critérios Funcionais (3)

- Aplicabilidade dos critérios funcionais:
  - Por ser independente da implementação, critérios funcionais podem ser utilizados em todas as fases de teste;
  - A medida que se move do teste de unidade para o teste de sistema, entradas e saídas mais complexas são exigidas, mas a abordagem permanece a mesma;
  - Além disso, principalmente no nível de sistema, o teste funcional é de fundamental importância uma vez que a estrutura interna do sistema é muito complexa, inviabilizando o teste estrutural.



Critérios Funcionais (4)

- Desvantagens dos critérios funcionais:
  - Dependente de uma boa especificação de requisitos (a qual, em geral, não é bem elaborada);
  - Não permite determinar que partes essenciais ou críticas da implementação do programa em teste foram executadas;
  - Para encontrar todos os defeitos com teste funcionais, cada combinação de entrada possível necessita ser exercitada (válidas e inválidas).
    - Entretanto, o teste exaustivo é quase sempre impossível.

Critérios Funcionais (5)

- Vantagens dos critérios funcionais:
  - Pode ser utilizado em todas as fases de teste;
  - Independente do paradigma e linguagem de programação utilizados;
  - Teste funcional sistemático direciona o testador a escolher subconjuntos de teste que são eficientes e efetivos em identificar defeitos.
    - Melhor que o teste aleatório (Copeland, 2004);
  - Eficaz em detectar determinados tipos de defeitos:
    - Funcionalidade ausente, por exemplo.

Critérios Funcionais (6)

- Exemplo: Particionamento de Equivalência
  - Qualquer valor dentro do intervalo tem a mesma importância, ou seja, qualquer valor escolhido é adequado;
  - O mesmo se aplica para os demais intervalos de dados;
  - Tais intervalos determinam o que é chamado de classe de equivalência;
  - Qualquer valor no intervalo de uma classe é considerado equivalente em termos de teste. Assim sendo:
    - Se um caso de teste de uma classe de equivalência revela um defeito, qualquer caso de teste da mesma classe também revelaria e vice-versa.

Critérios Funcionais (7)

- Passos de Aplicação:
  1. Identificar as classes de equivalência (requisitos de teste do critério)
  2. Criar o menor número de casos de testes para cobrir as classes de equivalência válidas;
  3. Criar um caso de teste para cobrir cada classe de equivalência inválida (entradas inválidas são grandes fontes de defeitos);
  4. Casos de teste adicionais podem ser criados caso haja tempo e recursos suficientes
     - Com base em sua experiência, o(a) testador(a) pode criar casos de teste adicionais.

Critérios Funcionais (8)

- Exemplo:

Especificação (extraído de Maldonado et al. (2004)):

O programa deve determinar se um identificador é válido ou não em Silly Pascal (uma variante do Pascal). Um identificador válido deve começar com uma letra e conter apenas letras ou dígitos. Além disso, deve ter no mínimo um caractere e no máximo seis caracteres de comprimento.

Exemplos de Identificadores:

abc12 (válido); cont*1 (inválido); 1soma (inválido); a123456 (inválido)

Critérios Funcionais (9)

- Classes de Equivalência (requisitos de teste do critério):

Condições de Entrada	Classes Válidas	Classes Inválidas	
Tamanho t do identificador	$1 \leq t \leq 6$ (1)	$t < 1$ (2)	$t > 6$ (3)
Primeiro caractere c é uma letra	Sim (4)	Não (5)	
Só contém caracteres válidos	Sim (6)	Não (7)	

Exemplo de Conjunto de Teste:
T0 = {(a1, Válido), ("", Inválido), (A1b2C3d, Inválido), (2B3, Inválido), (Z#12, Inválido)}

Critérios Estruturais (1)

- Os passos básicos para se aplicar um critério de teste estrutural é o seguinte:

A implementação do produto em teste é analisada;

↓

Caminhos por meio da implementação são escolhidos;

↓

Valores de entradas são selecionados de modo que os caminhos selecionados sejam executados;

↓

As saídas esperadas para as entradas escolhidas são determinadas;

↓

Os casos de testes são construídos;

↓

As saídas obtidas são comparadas com as saídas esperadas.

↓

Um relatório é gerado para avaliar o resultado dos testes.

Critérios Estruturais (2)

- Alguns critérios estruturais:
  - Critérios Baseados em Fluxo de Controle:
    - Todos-Nós, Todas-Arestas, Todos-Caminhos
  - Critérios Baseados em Fluxo de Dados:
    - Todas-Definições, Todos-P-Usos, Todos-C-Usos, Todos-Usos, Todos-Du-Caminhos, Todos-Pot-Usos, Todos-Pot-Du-Caminhos, Todos-Pot-Usos/Du.

Critérios Estruturais (3)

- Aplicabilidade dos critérios estruturais:
  - Critérios da técnica estrutural também podem ser utilizados em todas as fases de teste.
  - São mais comuns no teste de unidade e de integração
    - Teste de caminhos:
      - Caminhos dentro de uma unidade
      - Caminhos entre unidades
      - Caminhos entre sub-sistemas
      - Caminhos entre o sistema todo

Critérios Estruturais (4)

- Desvantagens dos critérios estruturais:
  - O número de caminhos a serem executados pode ser infinito (semelhante ao teste exaustivo);
  - Assume fluxo de controle correto (ou próximo do correto).
    - Casos de testes são baseados em caminhos existentes: caminhos inexistentes não podem ser descobertos;
  - Determinação de caminhos não executáveis pode ser um problema.
    - Dificuldade de automatização;
  - Habilidades de programação avançadas exigidas para compreender o código e decidir pela executabilidade ou não de um caminho.

Critérios Estruturais (5)

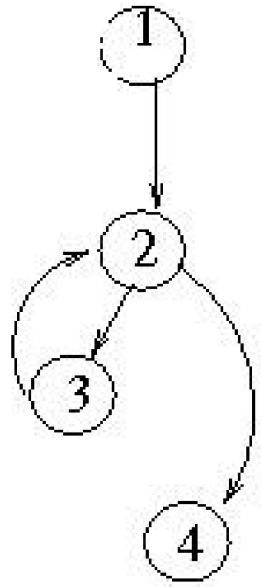
- Vantagens dos critérios estruturais:
  - Eficaz em determinar defeitos de lógica e de programação, especialmente no nível de unidade;
  - É possível garantir que partes essenciais ou críticas do programa tenham sido executadas:
    - Requisito mínimo de teste: garantir que o programa foi liberado tendo seus comandos executados ao menos uma vez por pelo menos um caso de teste.
  - É possível saber o que ainda não foi testado.

Critérios Estruturais (6)

- Grafo de Fluxo de Controle
  - Abstração utilizada pela maioria dos critérios estruturais para derivar os requisitos de teste
  - Grafo orientado com os nós e arcos:
    - Um **nó** representa uma ou mais instruções as quais são sempre executadas em sequência, ou seja, uma vez executada a primeira instrução de um nó todas as demais instruções daquele nó também são executadas;
    - Um **arco**, também chamado de ramo ou aresta, representa o fluxo de controle entre blocos de comandos (nós).

Critérios Estruturais (7)

```
    void soma()
    {
1     int i,n;
1     int s=0;
1     scanf("%d",&n);
1,2,3 for (i=0; i<n; i++)
3           s = s+pow (i,2);
4       printf("%d",s);
    }
```



Caminho 1 2 4

todos-arcos: {(1,2), (2,3), (3,2), (2,4)}

Critérios Baseados em Defeitos (1)

- Baseado em enganos típicos cometidos durante o processo de desenvolvimento
- Exemplos de critérios
  - Error Seeding (Budd, 1981)
  - Análise de Mutantes (Teste de Unidade) (DeMillo et al., 1978)
  - Mutação de Interface (Teste de Integração) (Delamaro et al., 2001)

Critérios Baseados em Defeitos (2)

- Teste de mutação
  - A ideia básica do critério é a **hipótese do programador competente**: um programador competente escreve programas corretos ou próximo do correto.
    - Assumindo a validade desta hipótese: defeitos são introduzidos no programa por meio de pequenos desvios sintáticos que fazem com que a execução do produto leve a um comportamento incorreto.
    - O Teste de Mutação realiza pequenas alterações sintáticas no produto em teste.
    - O testador deve construir um conjunto de teste que mostre que tais modificações criaram produtos que não são corretos (Agrawal et al., 1989).

Critérios Baseados em Defeitos (3)

- Teste de mutação
  - Uma segunda hipótese explorada pelo Teste de Mutação é o **efeito de acoplamento** (DeMillo et al., 1978).
    - Defeitos complexos são uma composição de defeitos simples.
    - Conjuntos de teste que detectam falhas simples são também capazes de identificar falhas complexas (Budd, 1980).
    - Uma única alteração sintática é aplicada ao programa *P* em teste, ou seja, cada mutante tem uma única diferença sintática em relação ao programa original.

Critérios Baseados em Defeitos (4)

- Dados um produto que se deseja testar *P* e um conjunto de teste *T*. Os passos de aplicação do Teste de Mutação são:

1. Execução do produto original
   - *P* é executado com *T*
   - Se uma falha for identificada, o teste termina.
   - Se nenhuma falha for identificada, *P* ainda pode conter defeitos que *T* não foi capaz de revelar.
2. Geração dos mutantes
   - P é submetido a um conjunto operadores de mutação que transformam *P* em $M_1, M_2, \ldots, M_n$, denominados mutantes de *P*.

Operadores de mutação são regras que representam enganos frequentes ou desvios sintáticos relacionados com uma determinada linguagem de programação.

Critérios Baseados em Defeitos (5)

3. Execução dos mutantes
   - Os mutantes são executados com o mesmo conjunto de teste *T*
   - Mutantes mortos - resultados diferentes de *P*
   - Mutantes vivos - resultados idênticos ao de *P*
   - O ideal é ter apenas mutantes mortos, indicando que o conjunto de teste *T* é adequado para o teste de *P* em relação ao Teste de Mutação.

4. Análise dos mutantes vivos
   - Mutantes vivos são analisados para identificar possível equivalência em relação a P, ou expor uma fraqueza do conjunto de teste *T*.

Critérios Baseados em Defeitos (6)

- Análise de Mutantes Vivos
  - Mutante equivalente
    - Um mutante M é dito equivalente a P se para qualquer dado de entrada $d \in D$, o comportamento de M é igual ao de P: $M(d) = P(d)$
  - Mutante revelador de defeito (*fault-reveling)*
    - Um mutante é dito ser revelador de defeito se para algum caso de teste t, tal que $P(t) = M(t)$, conclui-se que $P(t)$ não está de acordo com a especificação, ou seja, a presença de um defeito em P foi revelada

Critérios Baseados em Defeitos (7)

- Escore de Mutação
  - Medida objetiva para avaliar a adequação de *T* em relação ao Teste de Mutação

$$ms(P,T) = \frac{DM(P,T)}{M(P) - EM(P)}$$

  - Onde:
    - *DM*(*P*,*T*): número de mutantes mortos por *T*;
    - *M*(*P*): total de mutantes gerados;
    - *EM*(*P*): número de mutantes equivalentes a *P*.

Critérios Baseados em Defeitos (8)

- Desvantagens do teste de mutação
  - Principal problema é o grande número de mutantes gerados.
    - Mutantes precisam ser compilados e executados;
    - Mutantes vivos precisam ser analisados devido a possível equivalência.
  - Requer bom conhecimento da implementação do produto para facilitar a análise de mutantes vivos.

Critérios Baseados em Defeitos (9)

- Vantagens do teste de mutação
  - É fácil de ser estendido para qualquer produto "executável" seja no nível de especificação ou implementação.
  - É um dos critérios de teste mais eficaz em detectar defeitos.

Critérios Baseados em Defeitos (10)

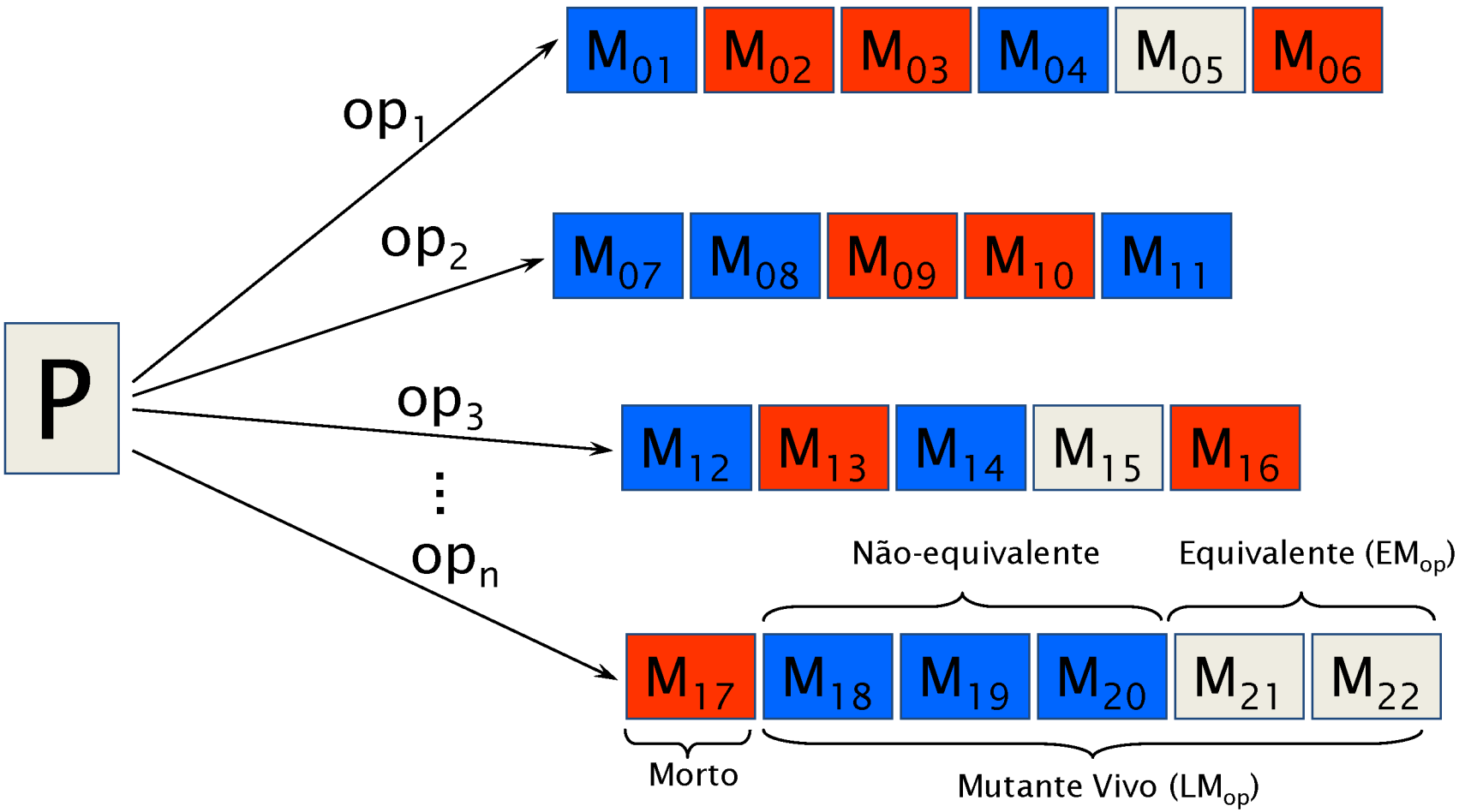
```
void soma()
{
 int i;
 int s=0;
 for (i=0; i<10; i++)
     s = s+pow(i,2);
 printf("%d",s);
}
```

Programa Original

```
void soma()
{
 int i;
 int s=0;
 for (i=0; i<=10; i++)
     s = s+pow(i,2);
 printf("%d",s);
}
```

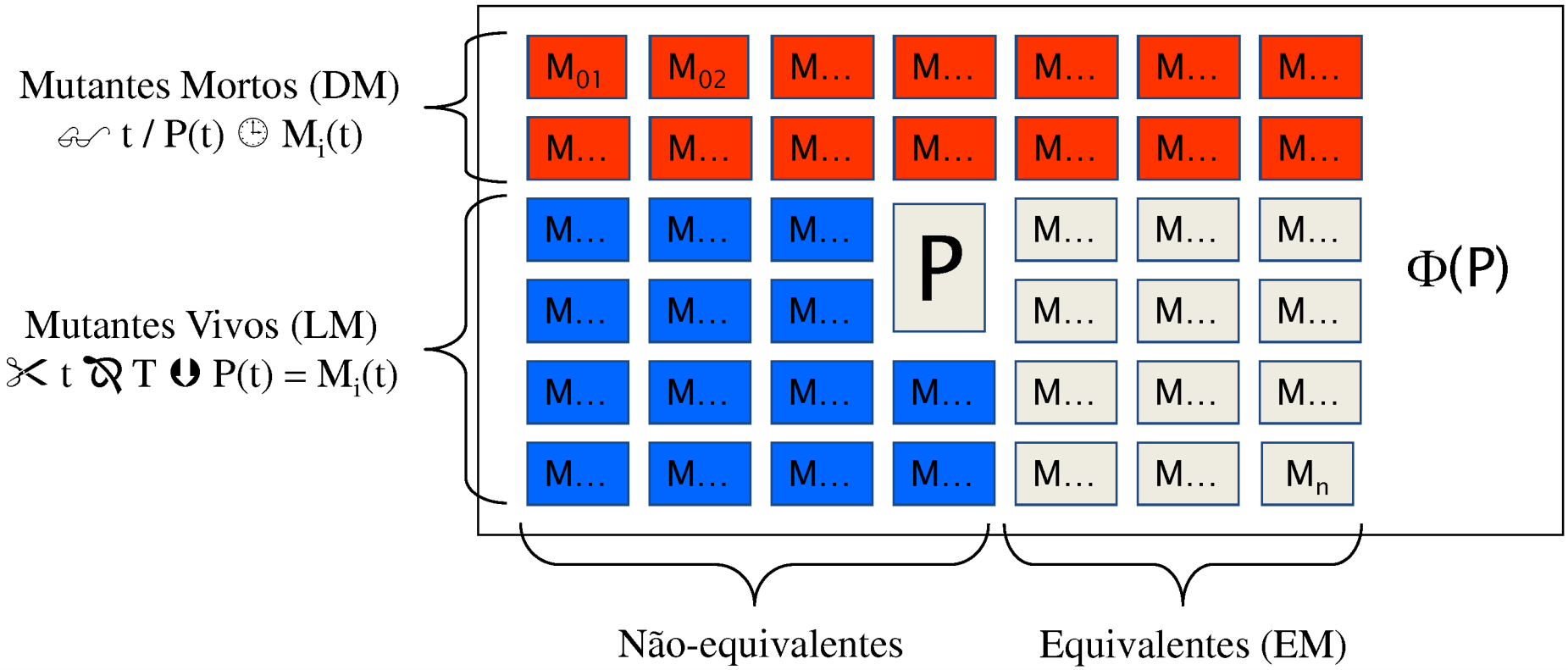
Programa Mutante

Critérios Baseados em Defeitos (11)



Critérios Baseados em Defeitos (12)

- Resultado Após a Execução de *P* e M_i
  - Escore de Mutação



Conjunto de Teste Adequado

- Quando um **conjunto de teste** ou de **dados de teste** *T* satisfaz todos os requisitos de um determinado critério de teste *C*, *T* é dito adequado a *C*, ou simplesmente, *T* é *C*-adequado.
- A partir de um *T* *C*-adequado é possível obter, em teoria, infinitos conjuntos de teste *C*-adequados, simplesmente incluindo mais dados/casos de teste em *T*.
- Posteriormente, técnicas de minimização de conjunto de teste podem ser usadas para reduzir a cardinalidade do conjunto.

Referências Bibliográficas

- H. Agrawal, R. A. DeMillo, R. Hathaway, W. Hsu, W. Hsu, E. W. Krauser, R. J. Martin, A. P. Mathur, and E. H. Spafford. Design of mutant operators for the C programming language. Technical Report SERC-TR41-P, Software Engineering Research Center, Purdue University, West Lafayette, IN, March 1989.
- P. Amey & B. Dion. Combining model-driven design with diverse formal verification. In: III European Congress of the Embedded Real Time Software – ERTS'2006, Toulouse, France, disponível on-line: http://www.esterel-technologies.com/technology/WhitePapers/. Acesso em: 09/08/2009., 2006, p. 1–8.
- B. W. Boehm & V. R. Basili. Software defect reduction top 10 list. Computer, v. 34, n. 1, p. 135–137, 2001.
- B. W. Boehm. Industrial software metrics top 10 list. IEEE Software, v. 4, n. 5, p. 84–85, 1987.
- T. A. Budd. Mutation Analysis of Program Test Data. PhD thesis, Yale University, New Haven, CT, 1980.
- T. A. Budd. Mutation Analysis: Ideas, Example, Problems and Prospects, chapter Computer Program Testing, pages 129–148. North-Holand Publishing Company, 1981.