Testes caixa preta - 1

Abordagens combinatórias

Criação: Abr/2001
Reformulação: Mar/2013

Referências

M. Pezzè, M. Young. **Teste e Análise de Software**. Bookman Companhia Editora, 2008, cap. 10 e 11.

P. Ammann, J. Offutt. **Introduction to Software Testing**. Cambridge University Press, 2008, cap.4.

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

NSF-SWENET. “Unit Testing”. SWENET Module. Obtido em maio/2005.

M. Grochtmann, K. Grimm. “Classification trees for partition testing”. Journal of Software Testing, Verification and Reliability, 3(2), 1993, pp63-82.

Tópicos

- Visão Geral
- Abordagens
  - Partição de equivalência
  - Análise de valores-limite
  - Particionamento em categorias
  - Combinação em pares
  - Testes aleatórios

Testes caixa preta - características

- Também chamados de *testes funcionais*, pois a especificação funcional é usada para derivar os casos de teste
- Especificação funcional muitas vezes pode ser completada ou criada pelo projetista de testes
  - ☺ Efeito colateral benéfico: ajuda a revelar falhas na especificação

- Projeto de casos de testes pode começar desde cedo
  - E continua ao longo do desenvolvimento
- Aplicáveis em todas as fases de testes: unidades → sistemas

- Prescindem do código fonte:
  - Úteis quando código fonte não está disponível (ex.: uso de componentes de terceiros)
  - Podem ser usados quando código muito complexo (ex.: testes de sistemas

Motivação - 1

- Especificação da função RAIZ:
  - A função aceita como entrada um valor inteiro
  - A função calcula, para esse inteiro dado, o seguinte valor e exibe o resultado:

$$\sqrt{(X-1)\times(X+2)}$$

  - Caso o valor da expressão seja negativo, a mensagem: “Erro- valor inválido para X” é exibida.

Qual o nº potencial de testes para esta função?

Motivação - 2

Qual o nº potencial de testes para esta página?

Problema

- Espaços de entrada podem ser muito grandes, até mesmo infinitos mas ...
- Cronogramas e orçamentos são finitos
- Como selecionar casos de teste com maior potencial para revelar a presença de falhas?
  - Lembrar que, aqui, consideramos que não se tem acesso ao código
- Como determinar quando parar de derivar casos de teste?

Exemplo - 1

[Binder00, cap. 3.3.3]

```
int exemplo (int j) {
        j = j - 1; // deveria ser j = j + 1
        j = j / 30000;
        return j;
}
```

- Supondo j um *int* de 16 bits: j$\in$ [-32.768, 32.767] $\Rightarrow$ total de 65.536 valores possíveis
- Se não há tempo de testar todos esses valores, que entradas escolher?

Exemplo - 2

[Binder00, cap. 3.3.3]

```
int exemplo (int j) {
        j = j - 1; // deveria ser j = j + 1
        j = j / 30000;
        return j;
}
```

Entrada (j)	Saída Esperada	Saída Obtida
1	0	0
42	0	0
40000	1	1
-64000	-2	-2

[CBSoft'2011 – Tutorial]

- Nenhum dos casos de teste acima revelam a presença da falha
- Para quais valores a falha é revelada?

Exemplo - 3

[Binder00, cap. 3.3.3]

```
int exemplo (int j) {
        j = j - 1; // deveria ser j = j + 1
        j = j / 30000;
        return j;
}
```

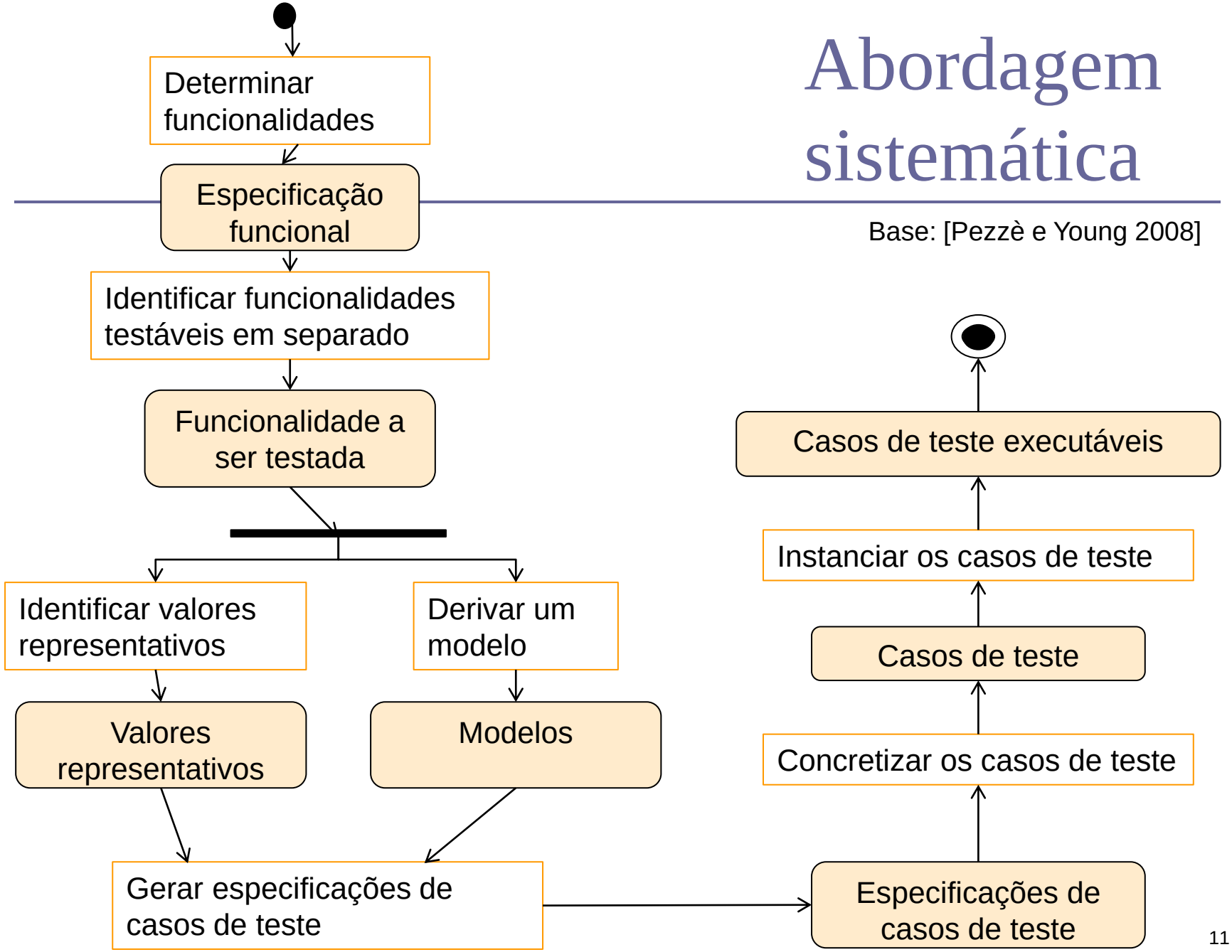
Entrada (j)	Saída Esperada	Saída Obtida
-30000	0	-1
-29999	0	-1
30000	1	0
29999	1	0

[CBSoft'2011 – Tutorial]

- Qual a chance desses valores serem selecionados?
- Depende da abordagem utilizada

Abordagem sistemática

Base: [Pezzè e Young 2008]



Identificar funcionalidades

- Decompor o sistema em funcionalidades distintas (unidades funcionais)
  - ☞ Unidades funcionais ≠ unidades de projeto
  - Unidade funcional: funcionalidades percebidas pelos usuários e que podem ser testadas independentemente
- Por que decompor: dividir para vencer
  - Simplifica a geração de casos de teste
  - Permite que cada funcionalidade seja examinada sistematicamente
  - Facilita a localização de falhas

Identificar classes de valores

- Cada unidade funcional tem entradas (e saídas)
  - Quais valores devem ser selecionados para cada entrada?
- Identificar valores representativos → requisitos de teste
- Como identificar esses valores?

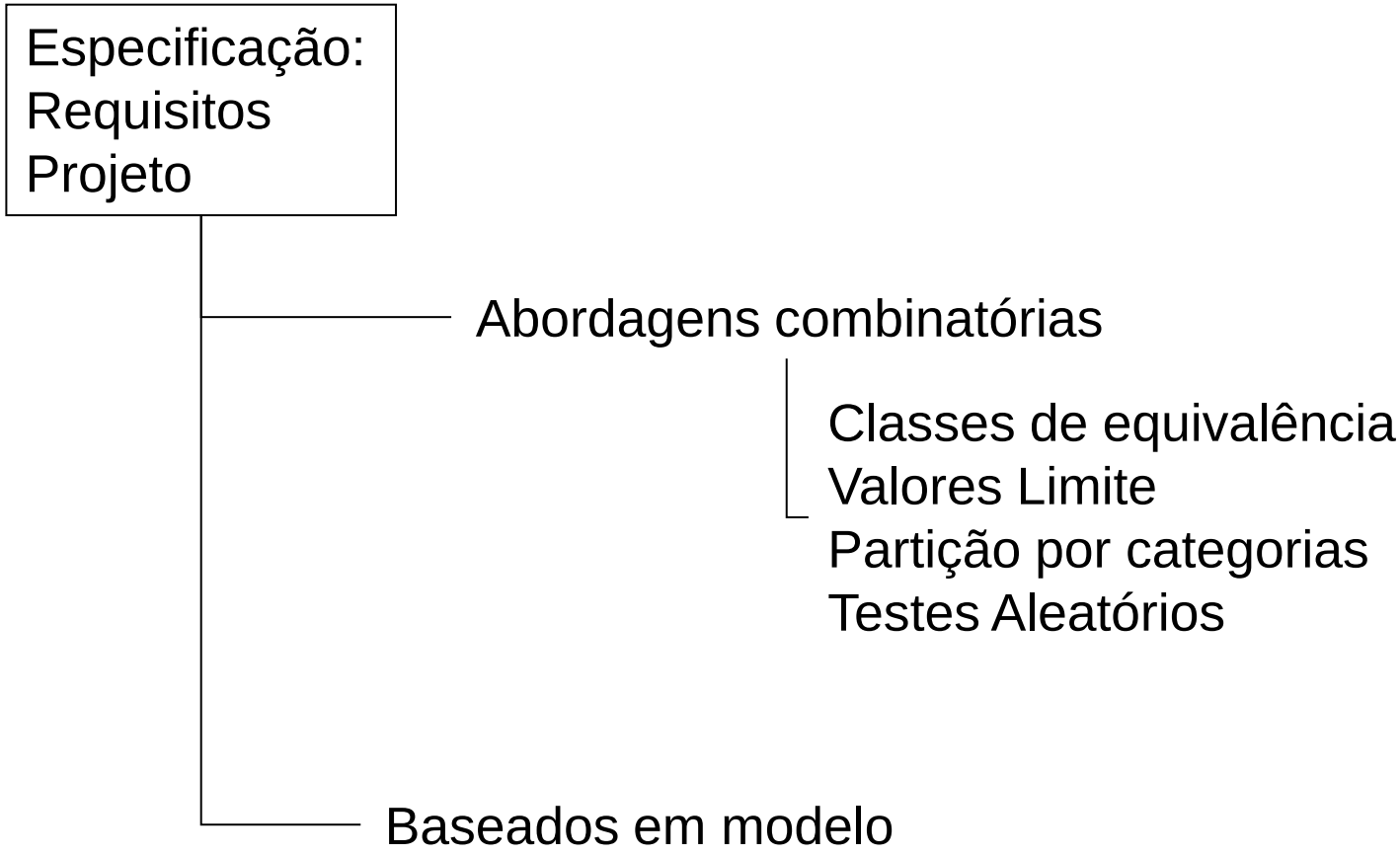
Gerar especificações de testes

- Casos de teste gerados são abstratos ≈ especificações de casos de teste
- Especificação de casos de teste = combinação de valores das entradas de uma unidade funcional
  - Nº elevado de combinações
  - Algumas combinações podem ser inviáveis
- Como limitar o nº de combinações?

Concretizar os casos de teste

- Transformar especificações de teste:
  - Selecionar 1+ casos de teste para cada especificação
  - Instanciar (se for o caso), i.e, atribuir valores específicos às entradas
  - Criar código para execução dos casos de teste

Algumas técnicas de testes caixa preta



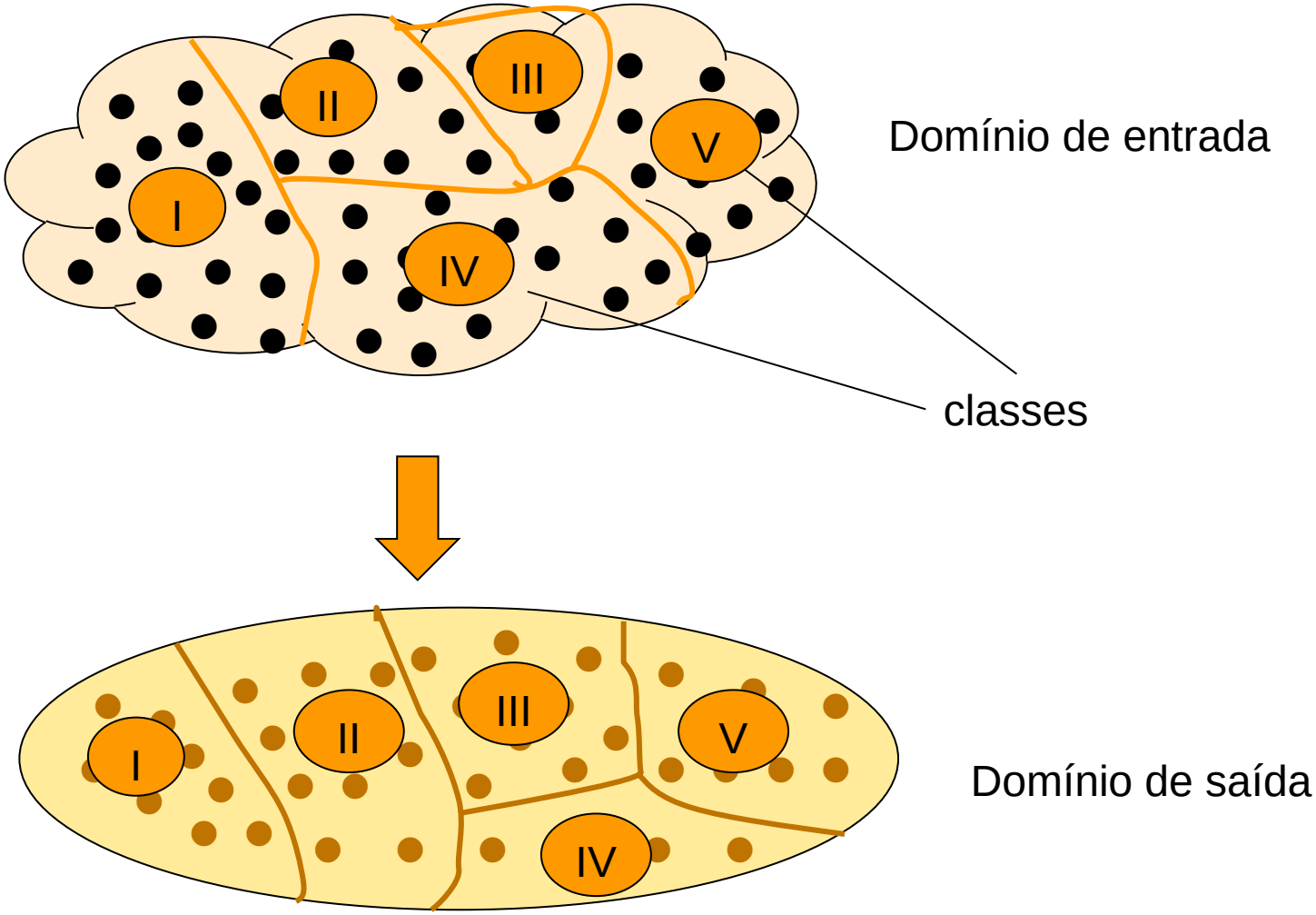
Abordagens combinatórias

- Usam critérios para dividir o domínio de cada entrada de uma UF em **partições**
- Produzem combinações de valores de cada entrada
- Algumas abordagens:
  - Partições em classes de equivalência
  - Partições por categorias
  - Combinações em pares

Testes baseados em partições : princípios

- O domínio de cada entrada da unidade funcional (UF) é dividido em partições de
  - Uma partição divide o domínio em **classes de equivalência**
  - supõe-se que dados pertencentes a uma classe de equivalência têm capacidade de revelar os mesmos tipos de falhas
  - uma classe de equivalência representa um conjunto de estados válidos e inválidos para uma dada condição de entrada
  - Classes de equivalência são **disjuntas**
  - A união das classes de equivalência = domínio **completo**

Testes baseados em partições: esquema



Testes

- Critério de cobertura:
  - cada partição deve ser considerada ao menos 1 vez
  - Cada partição é um requisito de testes

- Geração de testes:
  - selecionar um ou mais dados de cada partição

Exemplo

- Um programa recebe como entrada um arquivo, que pode ou não estar ordenado

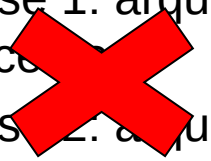
- Partição: ordenação do arquivo
  - Classe 1: arquivo em ordem crescente
  - Classe 2: arquivo em ordem decrescente
  - Classe 3: arquivo não ordenado

Esse particionamento é válido?

Exemplo

- Um programa recebe como entrada um arquivo, que pode ou não estar ordenado

- Partição: ordenação do arquivo
  - Classe 1: arquivo em ordem crescente
  - Classe 2: arquivo em ordem decrescente
  - Classe 3: arquivo não ordenado



- Partição 1: arquivo em ordem crescente
  - Classe 1: sim
  - Classe 2: não

  ou
- Partição 2: arquivo em ordem decrescente
  - Classe 1: sim
  - Classe 2: não

Partição em classes de equivalência: passos

1. Decompor o programa em unidades funcionais (UF)
2. Identificar as variáveis que determinam o comportamento de cada UF
3. Particionar os valores de cada variável em classes de equivalência (válidas e inválidas)
4. Especificar os casos de teste:
   - eliminar as classes impossíveis ou os casos desinteressantes
   - selecionar casos de testes cobrindo as classes válidas das diferentes variáveis
   - para cada classe inválida escolha um caso de teste que cubra 1 e somente 1 de cada vez

Determinação das classes de equivalência

Definição da variável de entrada	Classes de equivalência
Intervalo	• Uma classe válida para valores pertencentes ao intervalo • Uma classe inválida para valores menores que o limite inferior • Uma classe inválida para valores maiores que o limite superior
Lista de valores válidos	• Uma classe válida para os valores incluídos na lista • Uma classe inválida para todos os outros valores

Determinação das classes de equivalência

Definição da variável de entrada Número de valores válidos	**Classes de equivalência** • Uma classe válida para número de valores igual ao número previsto • Uma classe inválida para número de valores = 0 • Uma classe inválida para número de valores maior ou menor que o valor previsto
Restrições (expressão lógica; sintaxe; valor específico; compatibilidade com outras variáveis)	• Uma classe válida para os valores que satisfazem às restrições • Uma classe inválida para os outros valores

Exemplo – raiz quadrada

1. Identificar UF:

   Supor uma função que calcula o valor de

$$\sqrt{(X-1)\times(X+2)}$$

2. Valores representativos:
   - Valores válidos para X:
     - X ≤ -2
     - X ≥ 1

3. Partição em classes de equivalência

Condições de entrada	Classes válidas	Classes inválidas
X ≤ -2	C1. X ≤ -2	C3. -2 < X < 1
X ≥ 1	C2. X ≥ 1	

NSF-SWENET

Exemplo - raiz quadrada

- Gerar especificações de casos de teste:
  - selecionar casos de testes cobrindo as classes válidas das diferentes variáveis

		C1	C2	C3
CT1	$X \leq -2$	✓		
CT2	$X \geq 1$		✓	
CT3	$-2 < X < 1$			✓

Especificação das entradas de teste

variável	valor resultado
X	-5
X	5
X	0

Instanciação dos casos de teste

Exemplo 2

- Função MDC, que calcula o máximo divisor comum de dois inteiros (ambos não podem ser negativos)
  - MDC(a,b) = c onde
    - c é um inteiro positivo
    - c é divisor comum de a e b (i.e., c divide a e c divide b)
    - c é maior que todos os divisores comuns de a e b.
  - Exemplo:
    - MDC(45, 27) = 9
    - MDC (7,13) = 1
    - MDC(-12, 15) = 3
    - MDC(13, 0) = 13
    - MDC(0, 0) indefinido

Exemplo 2 – Valores representativos

O algoritmo do MDC pode aceitar quaisquer inteiros como entrada.

Neste exemplo vamos considerar que 0, inteiros positivos e inteiros negativos são valores especiais, i.e., são tratados diferentemente pelo programa.

Exemplo 2 – Classes de Equivalência

Variáveis	Classes de equivalência		
a	C1. a < 0	C2. a = 0	C3. a > 0
b	C4. b < 0	C5. b = 0	C6. b > 0

Exemplo 2 – Casos de teste

Classes de equivalência

Condições:		1	2	3	4	5	6
a<0, b<0	1	✓			✓		
a<0, b=0	2	✓				✓	
a<0, b>0	3						
.	4						
.	5						
.	6						
	7						
	8						
	9						

casos de teste

1.
2.
3.
4.
5.
6.
7.
8.
9.

Como combinar as classes?
São necessárias as 9 combinações?

Análise de valores-limite

- Critério de seleção que identifica valores nos limites das classes de equivalência
- Exemplos:
  - valor mínimo (máximo) igual ao mínimo (máximo) válido
  - uma unidade abaixo do mínimo
  - uma unidade acima do máximo
  - arquivo vazio
  - arquivo maior ou igual à capacidade máxima de armazenamento
  - cálculo que pode levar a “overflow” (“underflow”)
  - erro no primeiro (último) registro

Exemplo 1 - Análise de valores limites

- No exemplo 1, após determinar as classes de equivalência, devemos fazer uma análise de valores-limites para a escolha dos valores de cada classe (ou partição) de equivalência. Assim, considerando que a função roda em um processador de 32 bits, temos:

C1. $X \leq -2$	-2^{31}, -100, -2.1, -2
C2. $X \geq 1$	1, 1.1,100, $2^{31}-1$
C3. $-2 < X < 1$	-1.9, -1, 0, 0.9

Exemplo 2 - Análise de valores limites

- UF: MDC
- Valores limites

$$a = \begin{cases} C1.\ -2^{31}, -1 \\ C2.\ 0, \\ C3.\ 1, 2^{31}-1 \end{cases}$$

$$b = \begin{cases} C4.\ -2^{31}, -1 \\ C5.\ 0, \\ C6.\ 1, 2^{31}-1 \end{cases}$$

MDC – Plano de Testes (2)

Descrição dos testes / Dados	Resultados Esperados	Resultados Obtidos
Classes de equivalência		
a < 0 e b < 0 ➔ (-27, -45)	9	
a < 0 e b > 0 ➔ (-72, 100)	4	
a > 0 e b < 0 ➔ (121, -45)	1	
a > 0 e b > 0 ➔ (420, 252)	28	
a = 0 e b < 0 ➔ (0, -45)	45	
a = 0 e b > 0 ➔ (0 , 45)	45	
a > 0 e b = 0 ➔ (-27, 0)	27	
a > 0 e b = 0 ➔ (27, 0)	27	
a = 0 e b = 0 ➔ (0 , 0)	exceção	
Valores-limites		
(1 , 0)	1	
(-1 , 0)	1	
(0 , 1)	1	
(0 , -1)	1	
(0 , 0) (redundante)	exceção	
(1, 1)	1	
(1, -1)	1	
(-1, 1)	1	
(-1, -1)	1	

Ainda falta algum teste? Complete ...

Exercício

- Dado o método abaixo:

```
public int TipoTriang(int L1, int L2, int L3)
// Entradas: L1, L2 e L3 são os lados de
//                um triângulo
// Saídas produzidas:
//   TipoTriang = 1 se triângulo escaleno
//   TipoTriang = 2 se triângulo isósceles
//   TipoTriang = 3 se triângulo equilátero
//   TipoTriang = 4 se  não é triângulo
```

☞ Considera-se que triângulo é válido quando nenhum de seus lados é negativo

Fonte:[Ammann e Offutt 2008]

- Partição 1: relação dos lados de um triângulo com o valor 0

Entradas	Classe 1	Classe 2	Classe 3
L1	L1 < 0	L1 = 0	L1 > 0
L2	L2 < 0	L2 = 0	L2 > 0
L3	L3 < 0	L3 = 0	L3 > 0

- Essa partição é válida?
- Crie casos de teste para este programa.
- Os casos de teste refletem a funcionalidade do programa?
- Crie uma partição que reflita melhor a funcionalidade .

Partições por categorias

- Método de partições por categorias (CPM), de Ostrand & Balcer (1988)
  - Objetivos:
    - Sistematizar o particionamento, o que não ocorre no método Partições de Equivalência
    - Sistematizar a derivação das combinações de entradas
- Identificar entradas para a UF:
  - Entradas → fatores:
    - **Condições ambientais** que afetam a execução de uma unidade funcional
    - **Parâmetros**: entradas para uma UF, fornecidas pelo usuário ou por outro programa

Partições por categorias

- Identificar classes de valores de entrada:
  - **Categorias**:
    - Características dos fatores que afetam a execução da UF
    - São obtidas da especificação da UF: como a UF se comporta de acordo com uma dada característica do fator?
  - **Escolhas** (*choices*):
    - Partições em que são divididos os valores das categorias
    - Correspondem às classes de equivalência
  - **Restrições**:
    - Identificar restrições entre escolhas, que são descritas na forma de anotações:
    - **Propriedades ([property A, B, ...])** – designam 1+ propriedades
    - **Seletores** ([**if A and B and ...**]) – **conjunção** de propriedades associadas a outras escolhas
    - Anotações especiais para situações especiais ou de erro: **[single]** e **[error]**

Exemplo – identificação das categorias

- UF: um programa que lê um arquivo A contendo dois inteiros m e n, e produz como saída 1/(m+n)

CPM

- Categorias:
  - Status de A
  - m+n
- Escolhas:
  - Status de A = {NãoExiste; Vazio; NãoVazio}
  - m+n = {=0; ≠0}
- Restrições entre escolhas:
  - Escolha m+n só faz sentido se Status de A é NãoVazio

CPM anotado

- Categorias:
  - Status de A
  - m+n
- Escolhas:
  - Status de A =
    NãoExiste **[error]**
    Vazio **[error]**
    NãoVazio **[property TemDados]**
  - m+n =
    =0 **[if TemDados]**
    ≠0 **[if TemDados]**

Partições por categorias – Geração de combinações

- Critério:
  - exercitar cada categoria, selecionando uma escolha de cada por vez
- Uma escolha c só pode ser combinada com outras se as restrições permitirem.
  - O método requer a enumeração exaustiva de todas as combinações.
  - O número de combinações cresce exponencialmente com o número de fatores
  - No exemplo temos: 2 fatores, um com 3 categorias, outro com 2 categorias ➔ $3^1 \times 2^1 = 6$ combinações
  - As restrições ajudam a gerar um número reduzido:
    - A restrição **[erro]** indica uma classe de valores que só precisa ser testada uma vez, em combinação com classes válidas de outros fatores

Exemplo – casos de teste

CT	Status de A	m+n	Resultado
1	NãoExiste	=0	Erro
2	Vazio	=0	Erro
3	NãoVazio	=0	Msg erro
4	NãoVazio	≠0	1/(m+n)

- Na proposta de Ostrand & Balcer:
  - Ferramenta cria as especificações de testes por enumeração
  - Testador elimina combinações que violam as restrições

Algumas limitações do método de partições por categorias

[Grochtmann & Grimmm 93]

- Sem usar restrições: nº de casos de teste pode ser muito grande
- Uso de restrições para reduzir nº de casos de teste: depende muito do testador
- Não há hierarquias entre categorias
- Pouco auxílio para eliminar casos de teste que não satisfaçam às restrições

Árvore de classificação – proposta inicial

- Método baseado em árvore de classificação (CTM), proposto por Grochtmann & Grimm (1993) como alternativa ao CPM
  - Categorias → classificações
  - Escolhas → classes
  - Classificações e classes organizadas em estrutura hierárquica → árvore
  - Tabela de combinação para auxiliar a derivação dos casos de teste

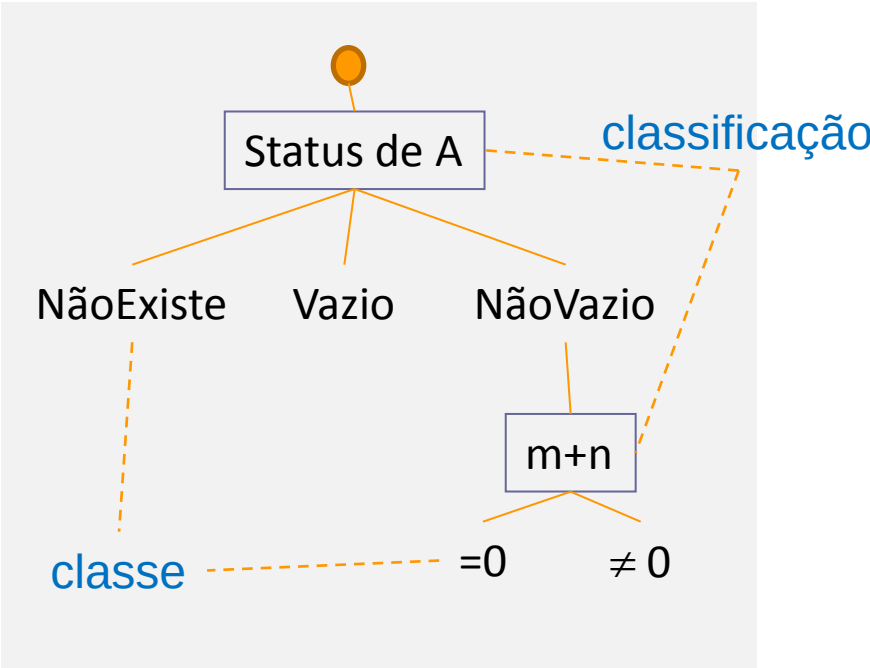
Exemplo

- Um programa lê um arquivo A contendo dois inteiros m e n, e produz como saída 1/(m+n)

CPM

- Categorias:
  - Status de A
  - m+n
- Escolhas:
  - Status de A = {NãoExiste; Vazio; NãoVazio}
  - m+n = {=0; ≠0}
- Restrições entre escolhas:
  - Escolha m+n só faz sentido se Status de A é NãoVazio

CTM



Como derivar casos de teste

- Selecionar 1 e somente 1 filho de uma classificação de alto nível
- Para cada classificação filha, selecione recursivamente um e somente um filho

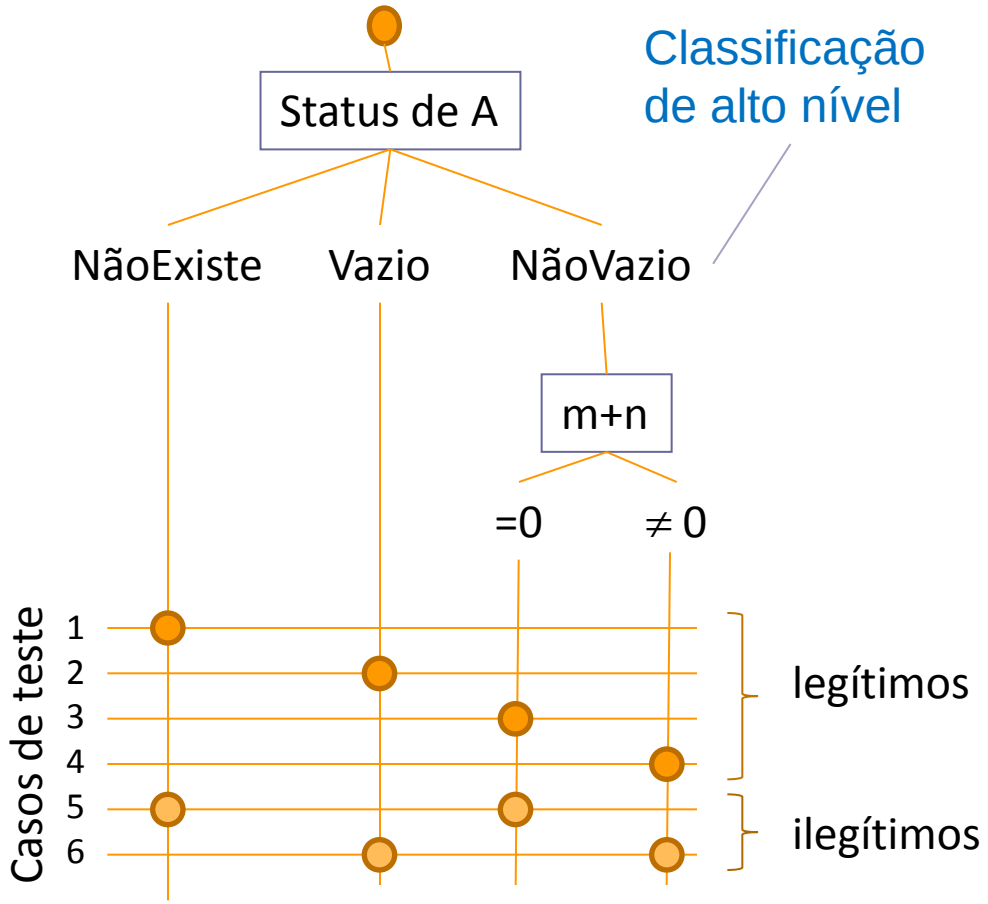


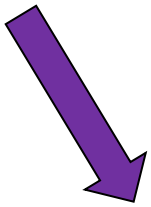
Tabela de combinações →

☞ Objetivo: minimizar nº de casos de teste ilegítimos

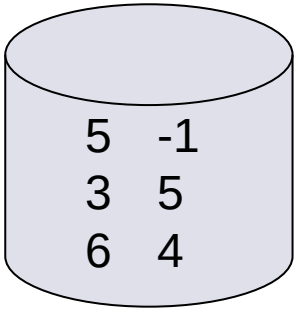
Exemplos de casos de teste

Especificação do caso de teste

Status de A: NãoVazio
m+n: $\neq 0$



Instâncias do caso de teste

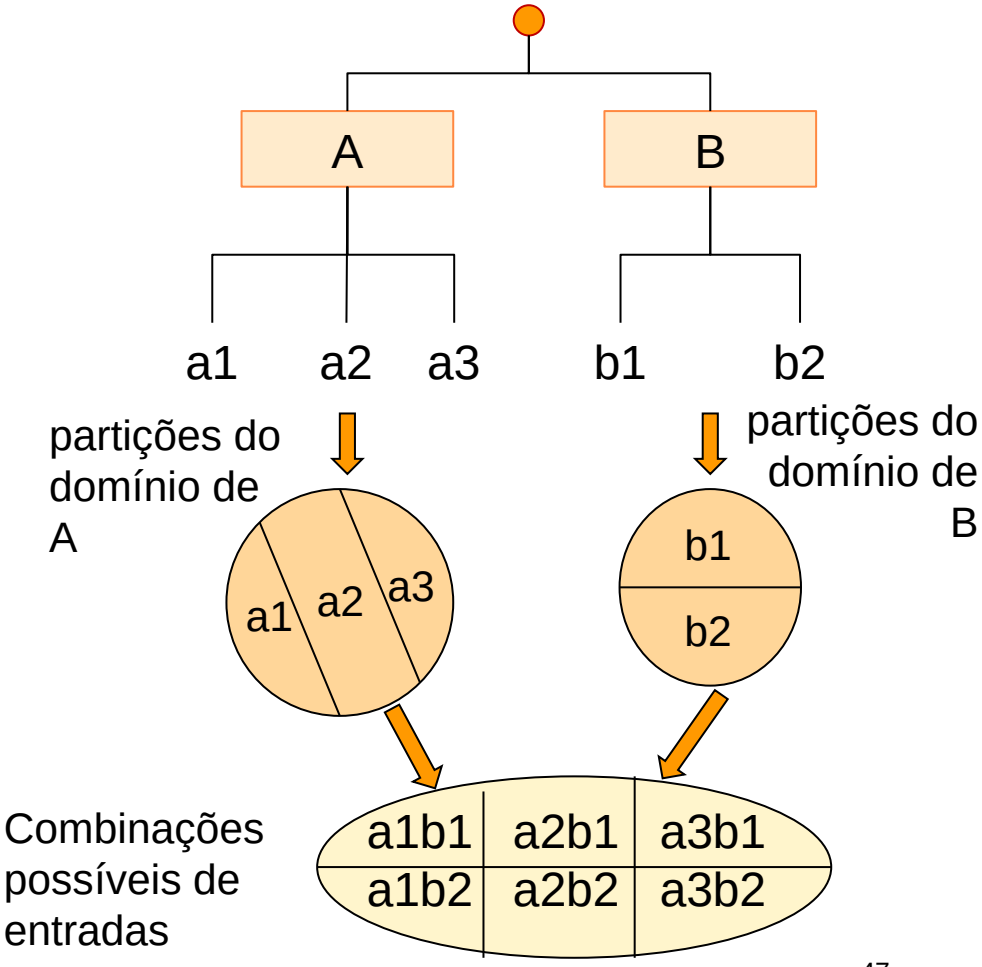


Resultados esperados

0.25
0.125
0.1

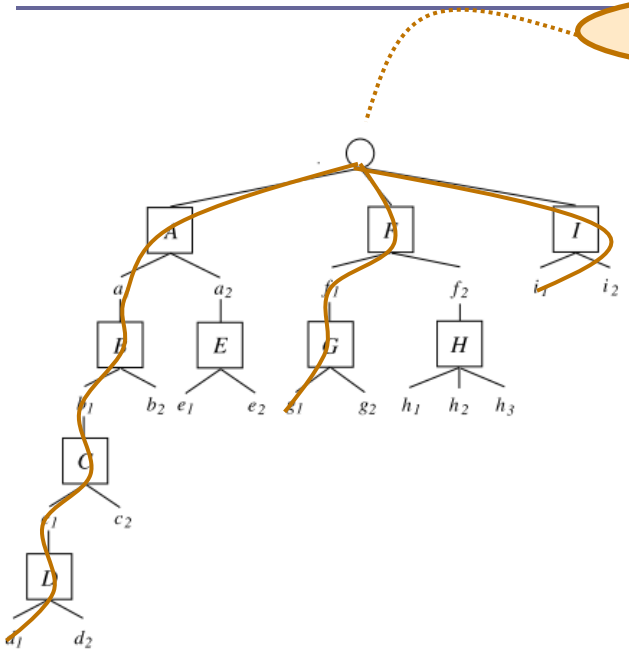
Árvore de classificação e partições

- As classificações na árvore induzem a partições do domínio de entrada que são:
  - Disjuntas: intersecção entre elas é vazia
  - Completas: a união delas é o domínio de entrada



Inspirado em [Grochtmann e Grimm 93]

Geração de casos de testes



No.	Potential Test Cases	No.	Potential Test Cases	No.	Potential Test Cases
1	$a_1, b_1, c_1, d_1, f_1, g_1, i_1$	21	$a_1, b_1, c_2, f_1, g_1, i_1$	41	a_2, e_1, f_1, g_1, i_1
2	$a_1, b_1, c_1, d_1, f_1, g_1, i_2$	22	$a_1, b_1, c_2, f_1, g_1, i_2$	42	a_2, e_1, f_1, g_1, i_2
3	$a_1, b_1, c_1, d_1, f_1, g_2, i_1$	23	$a_1, b_1, c_2, f_1, g_2, i_1$	43	a_2, e_1, f_1, g_2, i_1
4	$a_1, b_1, c_1, d_1, f_1, g_2, i_2$	24	$a_1, b_1, c_2, f_1, g_2, i_2$	44	a_2, e_1, f_1, g_2, i_2
5	$a_1, b_1, c_1, d_1, f_2, h_1, i_1$	25	$a_1, b_1, c_2, f_2, h_1, i_1$	45	a_2, e_1, f_2, h_1, i_1
6	$a_1, b_1, c_1, d_1, f_2, h_1, i_2$	26	$a_1, b_1, c_2, f_2, h_1, i_2$	46	a_2, e_1, f_2, h_1, i_2
7	$a_1, b_1, c_1, d_1, f_2, h_2, i_1$	27	$a_1, b_1, c_2, f_2, h_2, i_1$	47	a_2, e_1, f_2, h_2, i_1
8	$a_1, b_1, c_1, d_1, f_2, h_2, i_2$	28	$a_1, b_1, c_2, f_2, h_2, i_2$	48	a_2, e_1, f_2, h_2, i_2
9	$a_1, b_1, c_1, d_1, f_2, h_3, i_1$	29	$a_1, b_1, c_2, f_2, h_3, i_1$	49	a_2, e_1, f_2, h_3, i_1
10	$a_1, b_1, c_1, d_1, f_2, h_3, i_2$	30	$a_1, b_1, c_2, f_2, h_3, i_2$	50	a_2, e_1, f_2, h_3, i_2
11	$a_1, b_1, c_1, d_2, f_1, g_1, i_1$	31	a_1, b_2, f_1, g_1, i_1	51	a_2, e_2, f_1, g_1, i_1
12	$a_1, b_1, c_1, d_2, f_1, g_1, i_2$	32	a_1, b_2, f_1, g_1, i_2	52	a_2, e_2, f_1, g_1, i_2
13	$a_1, b_1, c_1, d_2, f_1, g_2, i_1$	33	a_1, b_2, f_1, g_2, i_1	53	a_2, e_2, f_1, g_2, i_1
14	$a_1, b_1, c_1, d_2, f_1, g_2, i_2$	34	a_1, b_2, f_1, g_2, i_2	54	a_2, e_2, f_1, g_2, i_2
15	$a_1, b_1, c_1, d_2, f_2, h_1, i_1$	35	a_1, b_2, f_2, h_1, i_1	55	a_2, e_2, f_2, h_1, i_1
16	$a_1, b_1, c_1, d_2, f_2, h_1, i_2$	36	a_1, b_2, f_2, h_1, i_2	56	a_2, e_2, f_2, h_1, i_2
17	$a_1, b_1, c_1, d_2, f_2, h_2, i_1$	37	a_1, b_2, f_2, h_2, i_1	57	a_2, e_2, f_2, h_2, i_1
18	$a_1, b_1, c_1, d_2, f_2, h_2, i_2$	38	a_1, b_2, f_2, h_2, i_2	58	a_2, e_2, f_2, h_2, i_2
19	$a_1, b_1, c_1, d_2, f_2, h_3, i_1$	39	a_1, b_2, f_2, h_3, i_1	59	a_2, e_2, f_2, h_3, i_1
20	$a_1, b_1, c_1, d_2, f_2, h_3, i_2$	40	a_1, b_2, f_2, h_3, i_2	60	a_2, e_2, f_2, h_3, i_2

Chen, T.Y.; Pak-Lok Poon; Tse, T. H., "A choice relation framework for supporting category-partition test case generation," *Software Engineering, IEEE Transactions on* , vol.29, no.7, pp.577,593, July 2003

Algumas ferramentas

- CTE (Classification-Tree Editor):
  - Permite editar árvore e gerar casos de testes (Grochtmann e Grimm)
  - Extensões: CTE/ES (para sistemas embarcados) e CTE XL (com lógica estendida)
  - CTE-XL (CTE with eXtended Logic): usa expressões lógicas para especificar restrições entre classes, e com isso reduzir geração de testes ilegítimos
- ADDICT:
  - Apoio ao método de Chen, Poon & Tse para construção da árvore reduzida
- EXTRACT (Extracting black-boX Test cases fRom Annotated Classification Trees)
  - Permite criar, editar, exibir e armazenar ACT, proposta por Yu et al.
  - Também permite gerar casos de teste

Outra estratégia de combinação

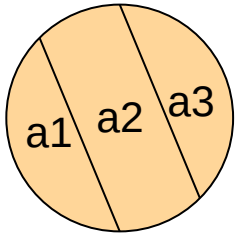
- Como selecionar combinações de partições ou classes para derivar as especificações de casos de teste?
  - Partição em classes de equivalência: cobertura de classes individuais
    - Não leva em conta combinações interessantes de valores de entrada
  - Particionamento em categorias/árvore de classificação: uso de enumeração exaustiva
    - Nº de combinações explode rapidamente → uso de restrições (reais ou arbitrariamente escolhidas pelo testador)
  - Estratégia intermediária?

Testes de combinações por pares

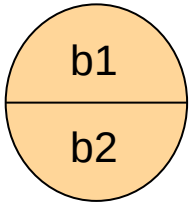
- Geram pares de classes para diferentes parâmetros
- Caso particular de estratégia n-árias, n<nº de entradas/fatores
- Nº de combinações cresce logaritmicamente com o nº de entradas, ao invés de crescer exponencialmente

Exemplo – enumeração exaustiva

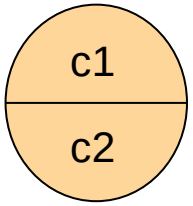
Partições para a entrada A

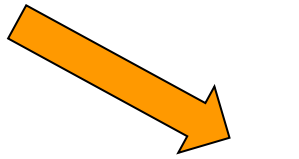


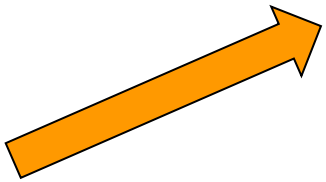
Partições para a entrada B



Partições para a entrada C

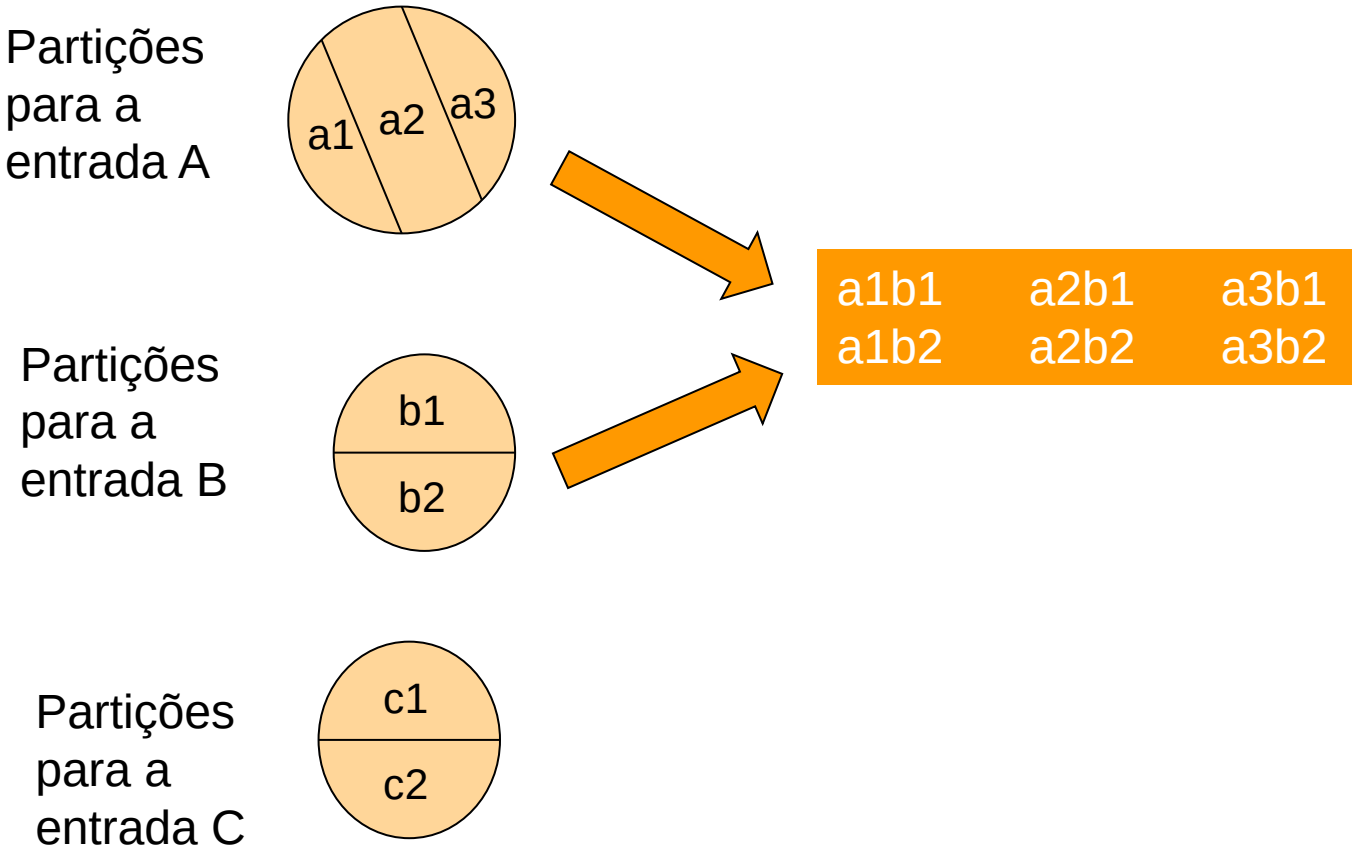






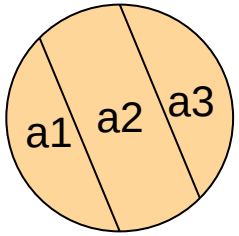
a1b1c1	a2b1c1	a3b1c1
a1b1c2	a2b1c2	a3b1c2
a1b2c1	a2b2c1	a3b2c1
a1b2c2	a2b2c2	a3b2c2

Exemplo – combinação de pares

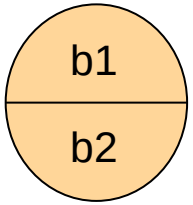


Exemplo – combinação de pares

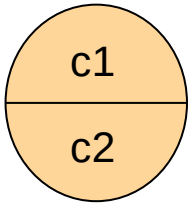
Partições para a entrada A

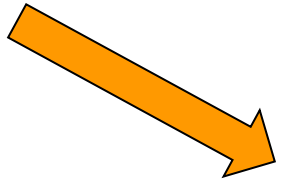


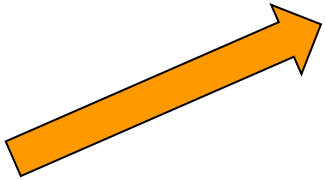
Partições para a entrada B



Partições para a entrada C



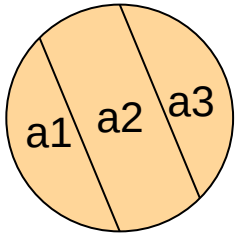




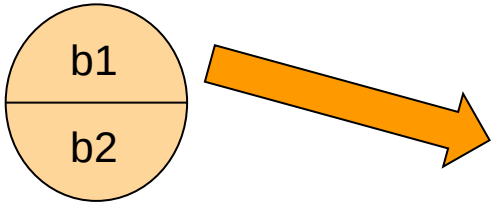
a1c1	a2c1	a3c1
a1c2	a2c2	a3c2

Exemplo – combinação de pares

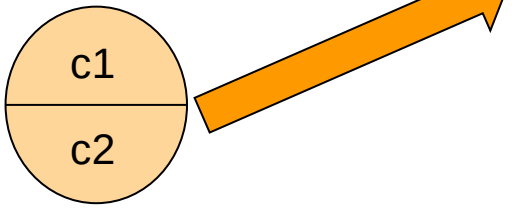
Partições para a entrada A



Partições para a entrada B



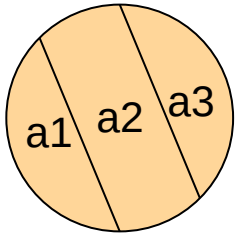
Partições para a entrada C



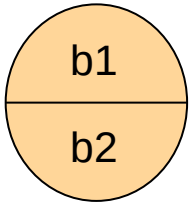
b1c1	b2c1
b1c2	b2c2

Exemplo – combinação de pares

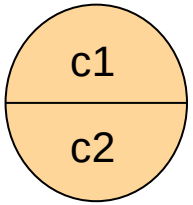
Partições para a entrada A



Partições para a entrada B



Partições para a entrada C



a1b1	a2b1	a3b1
a1b2	a2b2	a3b2

a1c1	a2c1	a3c1
a1c2	a2c2	a3c2

b1c1	b2c1
b1c2	b2c2

cobre os pares: a1b1, a1c1 e b1c1

a1b1c1	a2b1c2	a3b1c1
a1b1c2	a2b2c2	a3b2c2
a1b2 –	a2 – c1	

- todos os pares precisam ser cobertos
- cada caso de teste pode cobrir mais de um par
- "–" → não importa, qqr partição pode ser escolhida

Testes aleatórios

- Também conhecidos como RT (Random Testing): os testes são gerados aleatoriamente
- Fácil de implementar
- Muito usados para **testes de segurança**, para detectar vulnerabilidades
- Também muito usados em **testes de robustez**: como se comporta o sistema diante de entradas inesperadas?
- Existem diversas ferramentas:
  - Fuzz, Ridle

Testes aleatórios - limitações

- Dificuldade em satisfazer algum critério de cobertura
  - Gerar casos de teste para cobrir partes específicas que não foram testadas ainda
- Dificuldade em testar singularidades:
  - Ex.: testar programa que lê 3 lados de um triângulo e indica o tipo: isósceles, escaleno, equilátero
  - Difícil gerar 2 ou 3 entradas iguais

Exercício

- Dado o método abaixo:

```
public int TipoTriang(int L1, int L2, int L3)
// Entradas: L1, L2 e L3 são os lados de
//                um triângulo
// Saídas produzidas:
//   TipoTriang = 1 se triângulo escaleno
//   TipoTriang = 2 se triângulo isósceles
//   TipoTriang = 3 se triângulo equilátero
//   TipoTriang = 4 se  não é triângulo
```

☞ Considera-se que triângulo é válido quando nenhum de seus lados é negativo

	C1	C2	C3	C4
L1	>1	=1	=0	<0
L2	>1	=1	=0	<0
L3	>1	=1	=0	<0

- As partições são disjuntas e completas?
- Crie casos de teste usando enumeração exaustiva de todas as combinações de partições
- Crie casos de teste usando combinação por pares

[Ammann e Offutt 2008]