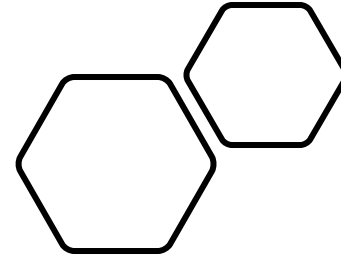


Teste de valor limite

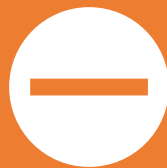


Prof. Bernardo Copstein

Baseado no livro “**Software Testing: From Theory to Practice**”

Maurice Aniche e Arie Van Deursen

Introdução



É normal cometer erros de programação quando se trabalha com condicionais envolvendo um “>” que deveria ser “>=”. Os programadores tendem a lidar melhor com a maioria dos valores de entrada, mas tendem a falhar quando as entradas “estão perto dos limites”;



A técnica de teste de valor limite objetiva endereçar este problema.

Limites entre classes/partições



Quando trabalhamos com teste baseado em especificação usamos o conceito de classes/partições.



Cada classe tem “fronteiras” com as outras classes, isto é, se eu vou alterando uma entrada em passos pequenos, em um dado momento a entrada passa a pertencer a próxima partição.



O ponto onde a entrada troca de partição é conhecido como “limite”

Exemplo

- **Requisito: calcular a quantidade de pontos de um jogador**
- Dado a pontuação (p) do jogador e suas vidas restantes (v) o programa retorna:
 - Se a pontuação do jogador esta abaixo de 50, então ele adiciona 50 aos pontos correntes.
 - Se a pontuação do jogador for maior que 50:
 - Se o número de vidas restantes for maior ou igual a 3, triplica-se a pontuação
 - Caso contrário soma-se 30 na pontuação atual

```
public int totalPoints(int p,int v){  
    if (p<50) return p+50;  
    return v<3 ? p+30 : p*3;  
}
```

Derivando partições (método “totalPoints”)

- **Partições:**

- **Menos pontos:** $p < 50$
- **Muitos pontos, poucas vidas:**
 $p \geq 50 \ \&\& \ v < 3$
- **Muitos pontos e muitas vidas:**
 $p \geq 50 \ \&\& \ v \geq 3$

Derivando casos de teste (método “totalPoints”)

- **Partições:**

- **Menos pontos:** $p < 50$
- **Muitos pontos, poucas vidas:**
 $p \geq 50 \ \&\& \ v < 3$
- **Muitos pontos e muitas vidas:**
 $p \geq 50 \ \&\& \ v \geq 3$

Valores de entrada (pontos, vidas)	Resultado esperado
P=30, V= 5	30+50
P=300, V=1	300+30
P=500, V=10	500*3

Implementando o driver (método “totalPoints”)

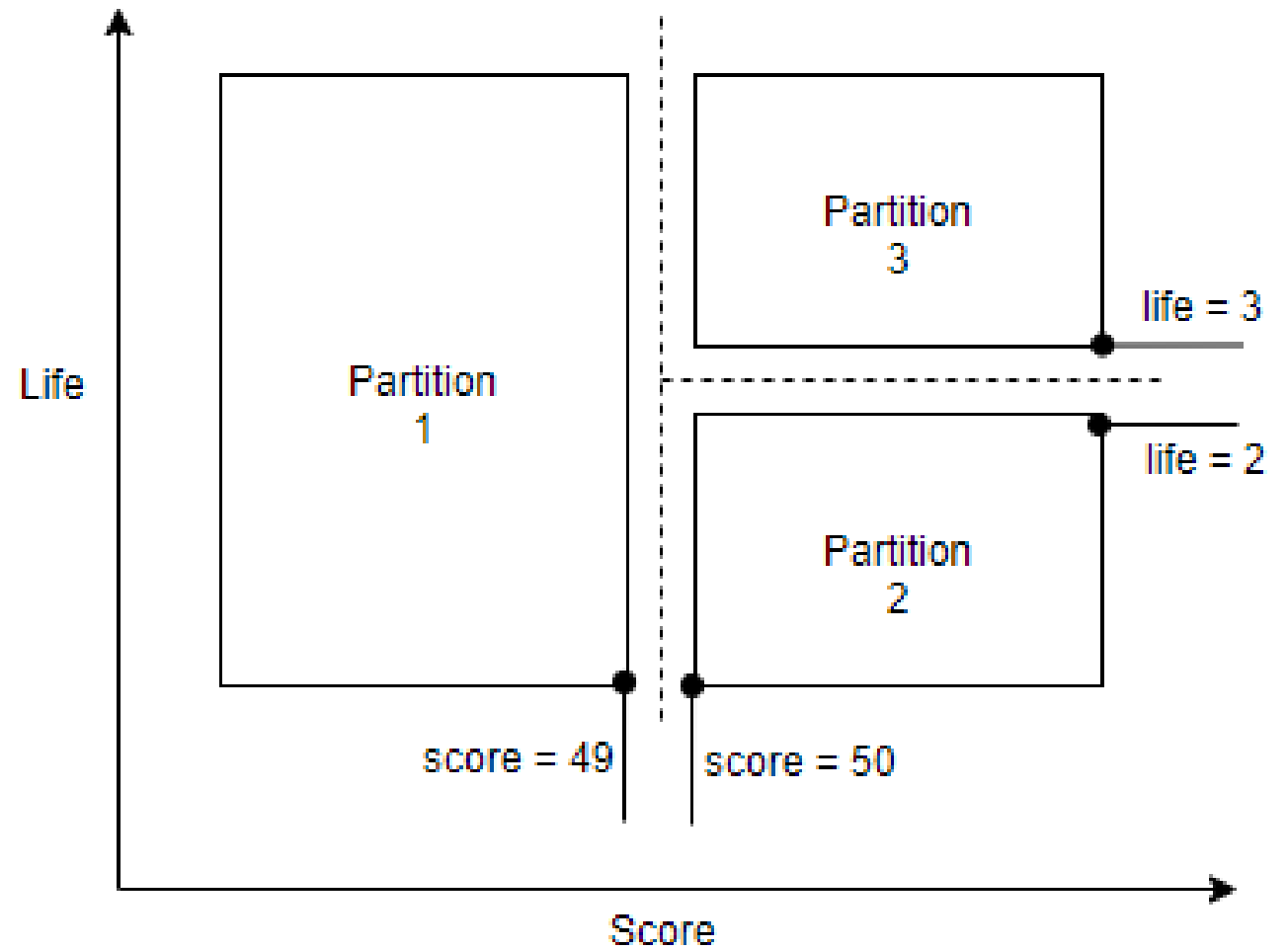
- **Partições:**

- **Menos pontos:** $p < 50$
- **Muitos pontos, poucas vidas:**
 $p \geq 50 \ \&\& \ v < 3$
- **Muitos pontos e muitas vidas:**
 $p \geq 50 \ \&\& \ v \geq 3$

```
public class PlayerPointsTest {  
    private final PlayerPoints pp = new PlayerPoints();  
    @Test  
    void lessPoints() {  
        assertEquals(30+50, pp.totalPoints(30, 5));  
    }  
    @Test  
    void manyPointsButLittleLives() {  
        assertEquals(300+30, pp.totalPoints(300, 1));  
    }  
    @Test  
    void manyPointsAndManyLives() {  
        assertEquals(500*3, pp.totalPoints(500, 10));  
    }  
}
```

Mas existem limites para verificar

- **Limite 1:** quando a pontuação é menor que 50, ela pertence a partição 1. Se for maior ou igual a 50, pertence as partições 2 e 3. Consequentemente quando a pontuação muda de 49 para 50 a partição muda (chamamos isso de teste L1)
- **Limite 2:** se a pontuação é maior ou igual a 50, observamos que se o número de vidas é menor que 3, então pertence a partição 2, caso contrário a partição 3. Então identificamos outro limite (chamaremos de teste L2).



Se um limite tem 2 entradas, cada uma exige dois casos de teste

- Para L1:
 - L1.1 → in={p=49, v=5}, out={99}
 - L1.2 → in={p=50, v=5}, out={150}
- Para L2:
 - L2.1 → in={p=500, v=3}, out={1500}
 - L2.2 → in={p=500, v=2}, out={530}

```
@Test
void betweenLessAndManyPoints() {
    assertEquals(49+50, pp.totalPoints(49, 5));
    assertEquals(50*3, pp.totalPoints(50, 5));
}

@Test
void betweenLessAndManyLives() {
    assertEquals(500*3, pp.totalPoints(500, 3));
    assertEquals(500+30, pp.totalPoints(500, 2));
}
```

Terminologia

On-point: o valor que esta exatamente no limite.

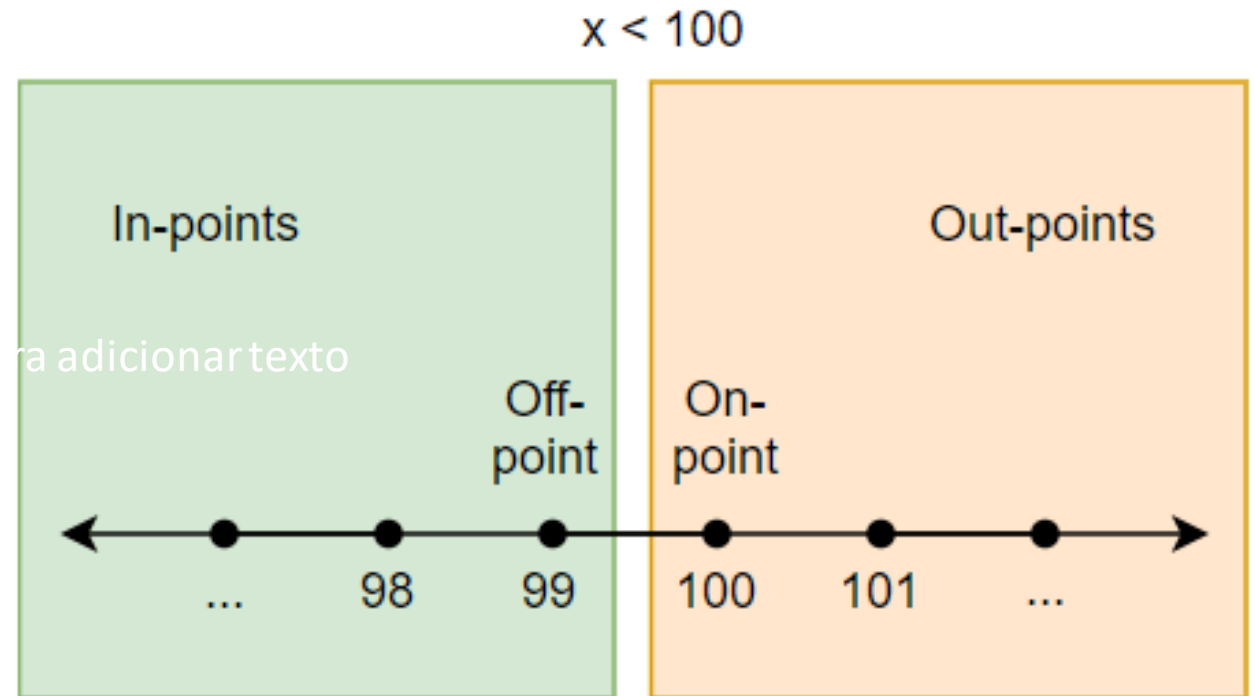
Off-point: é o valor mais próximo do limite antes de trocar a condição. Se o on-point torna a condição verdadeira, o off-point torna a condição falsa.

In-points: são todos os valores que tornam a condição verdadeira.

Out-points: são todos os valores que tornam a condição falsa.

Exemplo

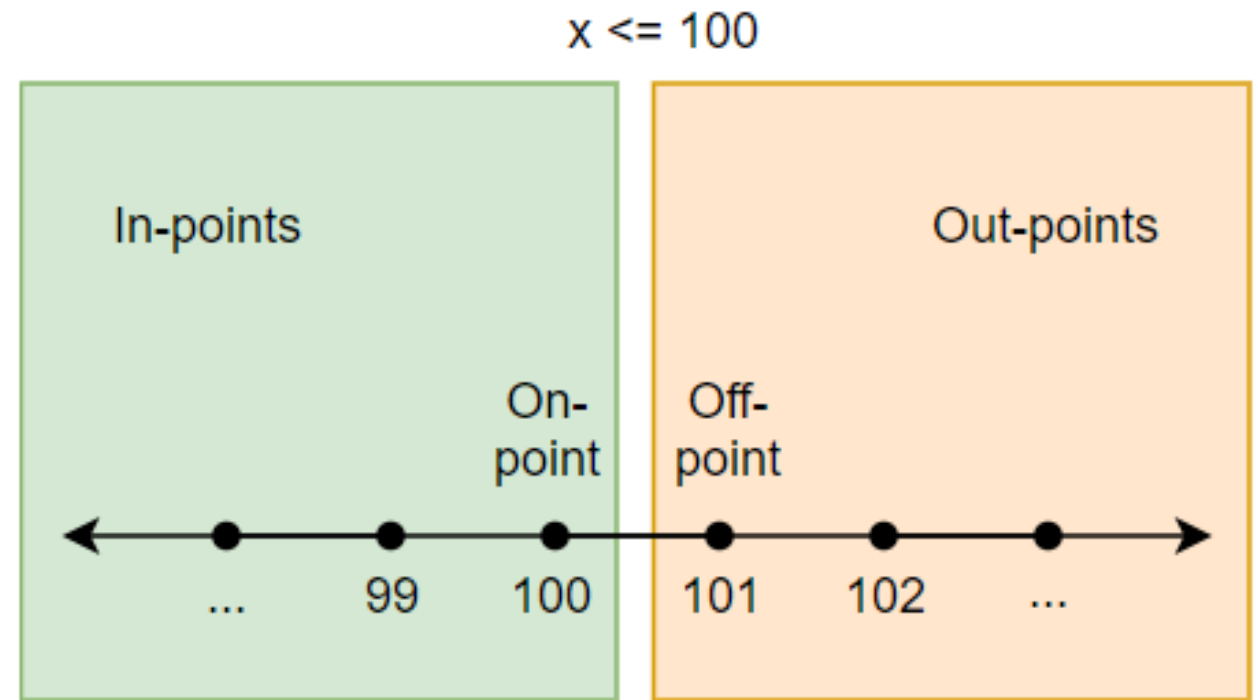
- Em uma venda acrescenta-se o custo do transporte quando o valor da venda é menor que 100 ($x < 100$).
- O on-point é 100, porque é o valor que esta precisamente na condição.
- O on-point torna a condição falsa (100 não é menor que 100), então o off-point deve ser o valor mais próximo que torna a condição verdadeira. Este será 99, pois $99 < 100$ é verdadeiro.
- Os in-points são os valores menores ou iguais a 99. Por exemplo, 37, 42, 56.
- Os out-points são os valores maiores ou iguais a 100. Por exemplo, 325, 1254, 101.



Se fosse um
pouquinho diferente

...

- Se a condição fosse $x \leq 100$ ao invés de $x < 100$:
 - O on-point ainda é 100: este é o valor que está precisamente na condição.
 - A condição é verdadeira para o on-point. Então o off-point é o número mais próximo do on-point que deixa a condição falsa. Então o off-point é 101.
- Observe que, dependendo da condição, um in-point pode ser um out-point e vice-versa.



Concluindo

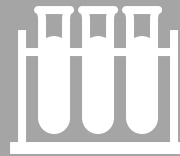
- Testadores devem prever casos de teste para:
 - O on-point
 - O off-point
 - Um único in-point (já que todos os pontos internos são equivalentes)
 - Um out-point



Derivando testes para condições múltiplas



Em muitos casos iremos encontrar condições múltiplas tais como :

$$a > 10 \ \&\& \ b < 20 \ \&\& \ c == 10 \ \&\& \ d \geq 50$$


Nestes casos o número de combinações pode crescer rapidamente

5 condições com 4 testes cada
Combinando todos os testes entre si
 $5^4 = 625$ testes !!



Para minimizar este problema iremos usar a **estratégia do domínio simplificado** proposta por Jeng and Weyuker.

A estratégia do domínio simplificado

- Seleciona-se os pontos on e off e cria-se um caso de teste para cada.
- Como se quer testar os limites separadamente, selecionam-se in-points para as demais variáveis/condições. Na prática queremos que todas as demais condições permaneçam verdadeiras de maneira que possamos testar a saída de uma de forma independente.
- É importante variar os in-points nos diferentes testes. Isso permite verificar se o programa fornece resultados corretos para vários in-points.

A matriz de condições limite

- Usada para mostrar os casos de teste de maneira estruturada

Boundary conditions for $x > a \ \&\& \ y > b$

			test cases (x, y)			
Condition	type	t1	t2	t3	t4	
> a	on					
	off					
typical	in					
> b	on					
	off					
typical	in					

Exemplo

- **Requisito:** Pizza ou massa
- O programa decide quando uma pessoa deve comer pizza ou massa. Dados dois números aleatórios, x e y, se x estiver entre [5,20] e y for menor ou igual a 89 o programa retorna pizza, senão massa.

```
public String pizzaOrPasta(int x, int y) {  
    return (x >= 5 && x < 20 && y <= 89) ?  
        "pizza" :  
        "pasta";  
}
```

Aplicando apenas teste baseado em especificação

- **Pizza:** o programa retorna pizza.
 - T1={x=15, y=50}.
- **Pasta:** o programa retorna massa.
 - T2={x=15, y=100}.

```
public class PizzaPastaTest {  
    private final PizzaPasta pp = new PizzaPasta();  
  
    @Test  
    void pizza() {  
        assertEquals("pizza", pp.pizzaOrPasta(15, 50));  
    }  
  
    @Test  
    void pasta() {  
        assertEquals("pasta", pp.pizzaOrPasta(15, 100));  
    }  
}
```

Montando a matriz para valor limite

Boundary conditions for $x \geq 5 \ \&\& \ x < 20 \ \&\& \ y \leq 89$							
			test cases (x, y)				
	Condition	type	t1	t2	t3	t4	t5
	≥ 5	on					
		off					
	< 20	on					
		off					
	typical	in					
	≤ 89	on					
		off					
	typical	in					

Escolhendo valores

- 3 condições (2 x 3 = 6 casos)
- T1={x=5, y=24}, saída = pizza
- T2={x=4, y=13}, saída = massa
- T3={x=20, y=-75}, saída = massa
- T4={x=19, y=48}, saída = pizza
- T5={x=15, y=89}, saída = pizza
- T6={x=8, y=90}, saída = massa

Boundary conditions for $x \geq 5 \ \&\& \ x < 20 \ \&\& \ y \leq 89$								
			test cases (x, y)					
Variable	Condition	type	t1	t2	t3	t4	t5	t6
x	≥ 5	on	5					
		off		4				
	< 20	on			20			
		off				19		
	typical	in					15	8
y	≤ 89	on					89	
		off						90
	typical	in	24	13	-75	48		

Implementando os casos de teste

```
@Test
void boundary_x1() {
    assertEquals("pizza", pp.pizzaOrPasta(5, 24));
    assertEquals("pasta", pp.pizzaOrPasta(4, 13));
}

@Test
void boundary_x2() {
    assertEquals("pasta", pp.pizzaOrPasta(20, -75));
    assertEquals("pizza", pp.pizzaOrPasta(19, 48));
}

@Test
void boundary_y() {
    assertEquals("pizza", pp.pizzaOrPasta(15, 89));
    assertEquals("pasta", pp.pizzaOrPasta(8, 90));
}
```

Limites não tão explícitos

- Uma indústria de chocolate tem de despachar para seus clientes pacotes com diferentes pesos. As barras são produzidas com dois pesos: 1Kg e 5Kg. Precisa-se saber quantas barras de cada peso deve-se enviar para cada cliente sabendo que a prioridade é enviar barras de 5Kg. A classe “ChocolateBars” ao lado possui um método chamado “calculate”. Este retorna a quantidade de barras de 1Kg necessárias para completar a encomenda.

```
public class ChocolateBars {  
    public static final int CANNOT_PACK_BAG = -1;  
  
    public int calculate(int small, int big, int total) {  
        int maxBigBoxes = total / 5;  
        int bigBoxesWeCanUse = Math.min(maxBigBoxes, big);  
        total -= (bigBoxesWeCanUse * 5);  
  
        if(small <= total)  
            return CANNOT_PACK_BAG;  
        return total;  
    }  
}
```

Usando teste baseado na especificação

- **Classes de equivalência:**

- **Precisa apenas barras pequenas.** Uma solução que usa apenas barras pequenas.
- **Precisa apenas barras grandes.** Uma solução que usa apenas barras grandes.
- **Precisa barras grandes e pequenas.** Uma solução que usa tanto barras grandes como pequenas.
- **Barras insuficientes.** Um caso que é impossível atender porque não tem barras suficientes.
- **Fora da especificação.** um caso excepcional.

- **Casos de teste:** (p=pequena, g=grande, n=demanda)

- **Precisa apenas barras pequenas.** P=4, g=2, n=3
- **Precisa apenas barras grandes.** p=5, g=3, n=10
- **Precisa grandes e pequenas.** p=5, g=3, n=17
- **Barras insuficientes.** p=1, g=1, n=10
- **Fora da especificação.** p=4, g=2, n=-1

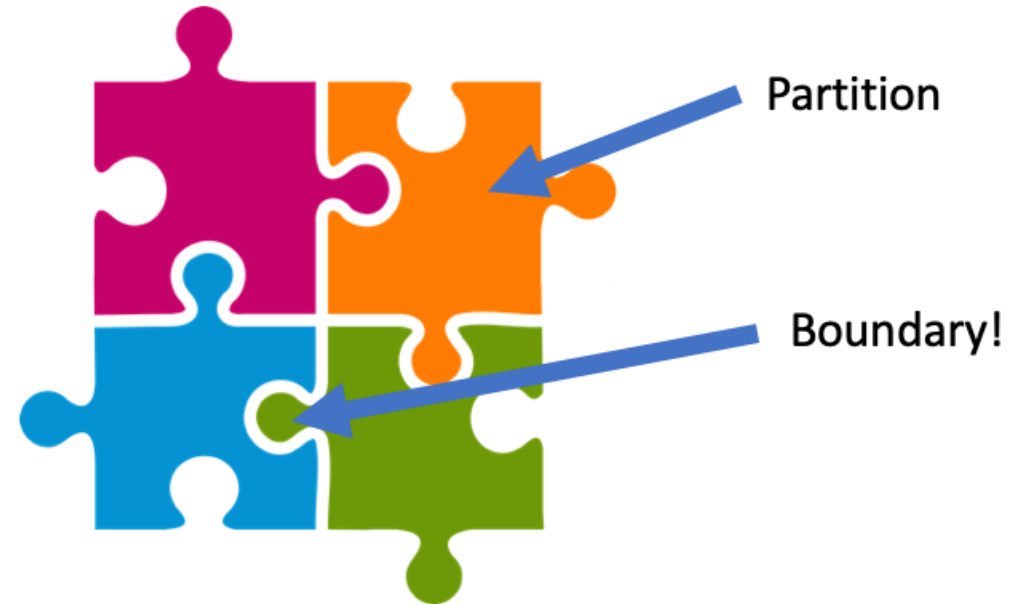
- Entretanto alguém testou com (2,3,17) como entrada e o programa falhou. Depois de alguma depuração percebeu-se que o comando “if(small <= total)” deveria ser “if(small < total)”. Aparentemente este bug é algo que só se consegue detectar usando teste baseado em limites.

- Observe que o teste (2,3,17) pertence a partição “Precisa barras grandes e pequenas”. Neste caso será feito uso de todas as barras grandes e de todas as pequenas. O programa funcionaria se existissem 3 barras pequenas disponíveis (3, 3, 17). Este é um problema de limite que não fica muito explícito nos requisitos.

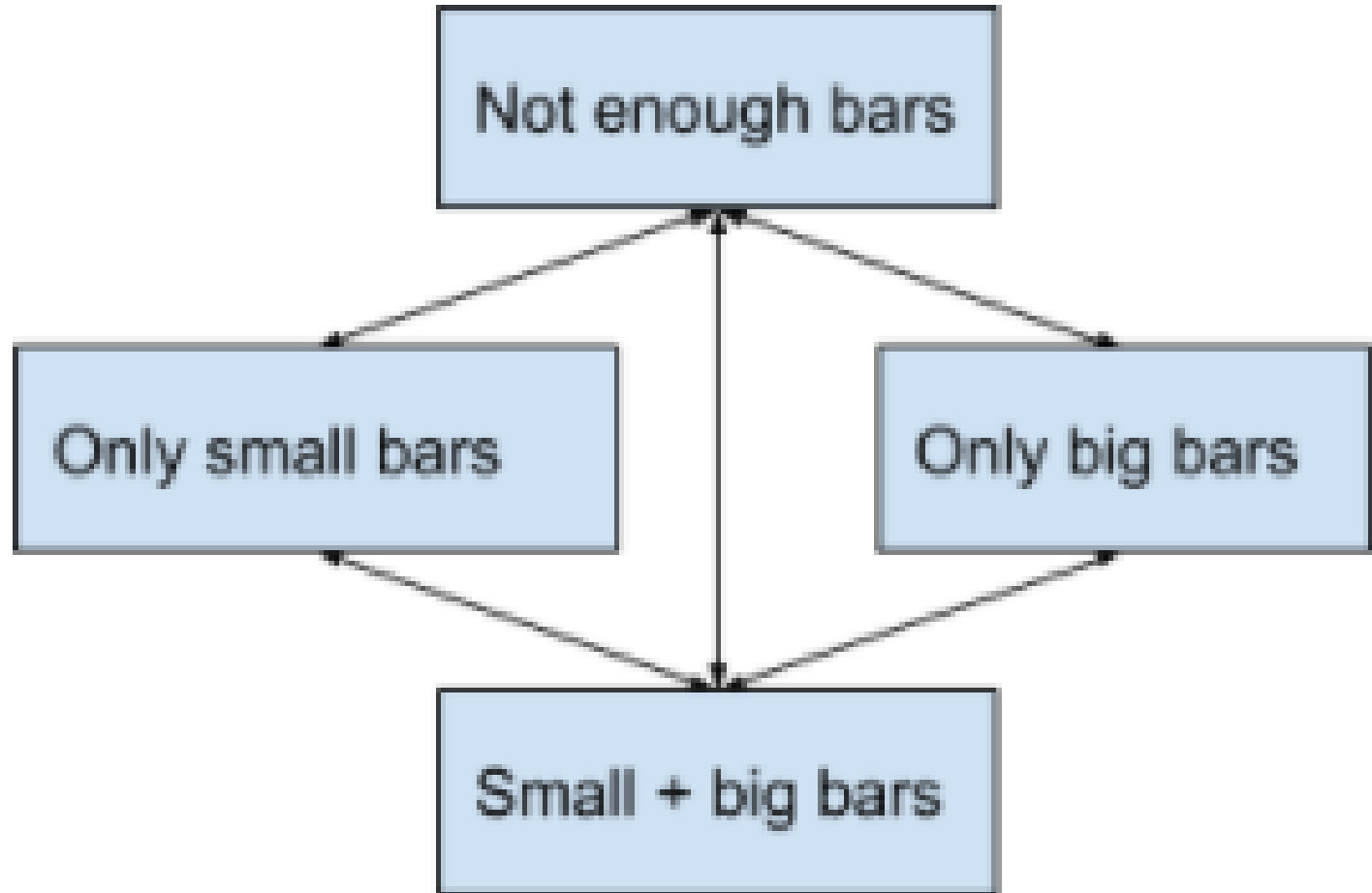
Limites “escondidos”

- O problema esta nos limites das partições
- Exemplo:
 - $(1,3,17) \rightarrow$ partição “barras insuficiente”
 - $(2,3,17) \rightarrow$ partição “precisa barras grandes e pequenas”

Note o limite, neste caso, na passagem de 1 para 2 barras grandes



Outros
limites
escondidos





Lesson learned

- Limites podem não emergir apenas das “condições dos ifs” que visualizamos na implementação. Limites podem ocorrer nas interações mais sutis entre as partições e são potenciais candidatos a defeitos.

JUnit: lidando com grandes volumes de dados

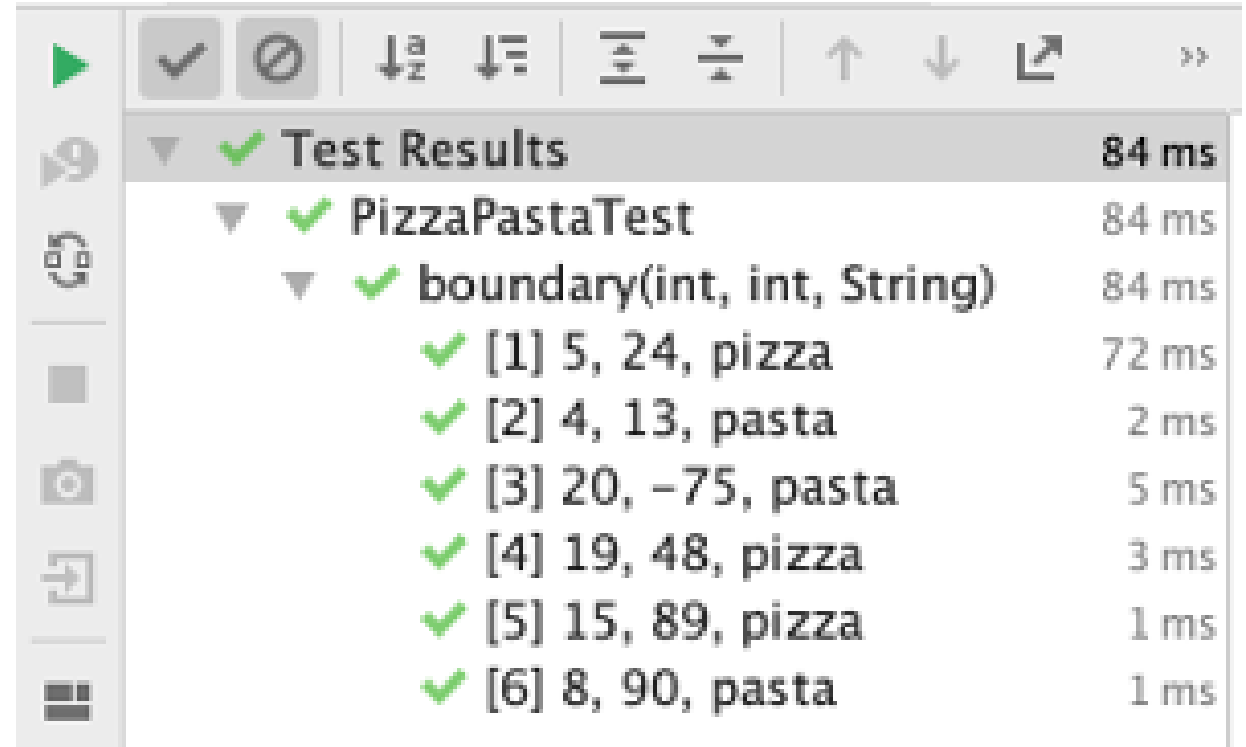
- Na medida que nos especializamos nas técnicas o número de casos de teste cresce
- Isso torna complicado criar um método de teste para cada caso de teste
- Para os casos de teste que tem a mesma estrutura, entretanto, o JUnit oferece o recurso dos “testes parametrizados”
- Incluir a seguinte dependência:

```
<dependency>
  <groupId>org.junit.jupiter</groupId>
  <artifactId>junit-jupiter-params</artifactId>
  <version>5.6.2</version>
  <scope>test</scope>
</dependency>
```

```
@ParameterizedTest
@CsvSource({
    "5, 24, pizza",
    "4, 13, massa",
    "20, -75, massa",
    "19, 48, pizza",
    "15, 89, pizza",
    "8, 90, massa"
})
void boundary(int x, int y, String eRes){
    assertEquals(eRes, pp.pizzaOrPasta(x, y));
}
```

Porque usar testes parametrizados

- Testes parametrizados devem ser usados no lugar de mais de um assert no mesmo método porque são capazes de gerar relatórios de teste como o da figura ao lado
- Outros recursos de teste parametrizados podem ser encontrados na documentação do JUnit



The screenshot shows the JUnit Test Results window. The test suite is 'Test Results' (84 ms). The test class is 'PizzaPastaTest' (84 ms). The test method is 'boundary(int, int, String)' (84 ms). The test method has six parametrized test cases, all of which passed (indicated by green checkmarks). The test cases are: [1] 5, 24, pizza (72 ms), [2] 4, 13, pasta (2 ms), [3] 20, -75, pasta (5 ms), [4] 19, 48, pizza (3 ms), [5] 15, 89, pizza (1 ms), and [6] 8, 90, pasta (1 ms).

Test Results	84 ms
✓ PizzaPastaTest	84 ms
✓ boundary(int, int, String)	84 ms
✓ [1] 5, 24, pizza	72 ms
✓ [2] 4, 13, pasta	2 ms
✓ [3] 20, -75, pasta	5 ms
✓ [4] 19, 48, pizza	3 ms
✓ [5] 15, 89, pizza	1 ms
✓ [6] 8, 90, pasta	1 ms

Aspectos a considerar quando procurando limites



Conformidade com formatos (eMail, RG, CEP etc)



Ordenação (se a ordem influencia)



Intervalos (o mais explícito: limites do intervalo, limites fora de faixa)



Referencias (pode aceitar null?)



Existência (BD responde? servidor responde?)



Cardinalidade (os laços iteram o número correto de vezes)



Tempo (timeout, concorrência, ordem temporal invertida)



Veja a lista
de
exercícios