

# FIA/P GRADUAÇÃO

**DISCIPLINA: PROJETO DE SISTEMAS APLICADO AS MELHORES PRÁTICAS EM QUALIDADE DE SOFTWARE E GOVERNANÇA DE TI**

**AULA:  
22 – TESTE UNITÁRIO DE SOFTWARE**

**PROFESSOR:  
RENATO JARDIM PARDUCCI**

PROFRENATO.PARDUCCI@FIAP.COM.BR

[Renato Parducci - YouTube](#)

## AGENDA DA AULA

- ✓ CMMi nível 3 - VER/VAL
- ✓ MPS.br nível D - VER/VAL
- ✓ Técnicas para planejar e aplicar testes de Caixa branca e preta
- ✓ Modelos de definição de testes (Testes do nível Unitário)

## **PRÁTICAS E NÍVEL 3 –TS, VER/VAL**

**Elaboração e aplicação de Teste  
Unitário de Software**

## ESTUDO DE CASO SIMULADO



A GD (Gerência de Desenvolvimento) da empresa de Dilan quer que a sua equipe desenvolva testes para um programa de aplicação que está em construção.

O programa está sendo construído com base no algoritmo publicado em ...

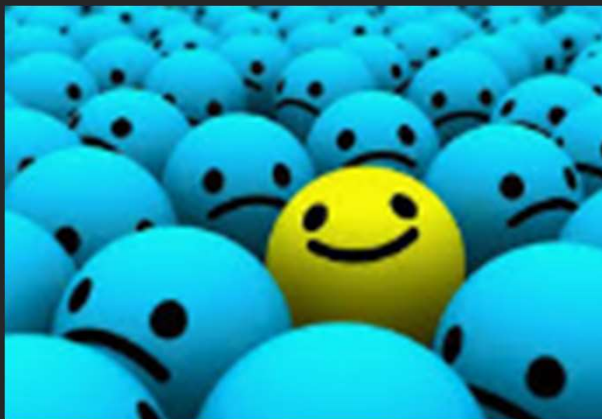
***Algoritmo-Testes-Unitários.docx***

Consuelo vai realizar um treinamento e você praticará imediatamente as técnicas e ferramentas que aprender para poder avaliar a qualidade do programa elaborado com base no algoritmo apresentado com as regras de negócio e lógica de aplicação desejada.

A ideia é que a equipe saiba exatamente quais testes fazer e quantos são, otimizando o ciclo de testes!

## COMO PLANEJAR UM CASO DE TESTE

### ELABORAÇÃO DE TESTES UNITÁRIOS



**Consistem em isolar as comunicações de um componente de software e testar o comportamento funcional desse item, sem interferências de outros componentes.**

-

Em programação Orientada a Objetos, consiste na elaboração de teste sobre um Método de Classe de Objetos escolhido, mascarando as chamadas de outros métodos e simulando retornos desses métodos externos, possibilitando isolar a lógica do Método que queremos testar.

## COMO PLANEJAR UM CASO DE TESTE

**Vamos iniciar a criação e aplicação de testes unitários e validar nosso código!**

Imagine que a sua empresa tem pressa na liberação do software que é aguardado para um lançamento comercial urgente.

Sua chefia pede celeridade nos testes – quer que façam o mínimo de testes possível que garantam uma boa segurança.

Quantos testes você faria?

Quais seriam?

Vamos descobrir a seguir...

## COMO PLANEJAR UM CASO DE TESTE

Serão apresentadas a seguir, os **seguintes métodos de definição** de casos de **testes unitários, funcionais de caixa branca**:

- Iniciação de variáveis;
- Caminhos mínimos (complexidade ciclomática);
- Limites;
- Enlace;
- Condição e Equivalência.

Usando esses métodos , **criamos fichas de teste dirigido, com dados controlados de Input e Output**. A ficha de descrição de testes manuais permanece a mesma estudada anteriormente.

Caso esteja aplicando **TDD**, vai poder saber exatamente quais testes devem ser construídos na JUNIT para orientar a programação!



## TESTE DE SOFTWARE

### ATIVIDADE PRÁTICA



Vamos iniciar uma maratona de aprendizado de técnicas de teste Unitário!

*Vocês irão trabalhar em duplas, programando em pares.*

*Utilizem o Algoritmo-Testes-Unitários.docx, disponibilizado pelo professor e façam a programação em JAVA com Eclipse.*

*Incluam erros propositalis como sentenças de validação lógica com parâmetros errados, valores de variáveis diferentes dos previstos, cálculos errados, nomes de métodos de classe incorretos.*

*Vocês vão passar o seu programa para a equipe ao seu lado para que testem, aplicando as técnicas que estudaremos em sequência e vocês testarão o programa de outro grupo.*

*Assim que terminarem a programação, PAREM!*

## TESTE DE SOFTWARE

## ATIVIDADE PRÁTICA



## PROVA DE FOGO NÚMERO 01

*Os programadores devem anotar em uma planilha resumo, os erros propositalis que “plantaram” defeitos no código, sem deixar que a dupla ao lado (que vai testar o software), conheça quais são esses defeitos!*

*Os testes serão feitos mascarando chamadas externas do código que queremos testar e forçando valores nas variáveis de input, dando display das variáveis de output (como alternativa, podemos usar o depurador e manipular os valores de input e observar as variáveis de output).*

***Nos testes Unitários funcionais, assim como nos de Integração, precisamos trabalhar com Casos de Teste de Caixa Branca e Dados dirigidos.***

## COMO PLANEJAR UM CASO DE TESTE

**INICIALIZAÇÃO DE VARIÁVEIS:** uma das maiores causas de falhas intermitentes em software é a falta de inicialização de variáveis.

Antes de proceder com qualquer teste funcional, é necessário:

- 1º) Mapear quais as variáveis usadas no programa a testar;
- 2º) Identificar qual a primeira linha do código onde a variável aparece após a declaração do seu tipo;
- 3º) Avaliar se essa linha é de inicialização – se não for, existe um problema de falta de inicialização a ser corrigido.

```

1. Boolean x;
2. Int y;
3. Ler y;

4. SE x = False ENTÃO
5.   y= y +1
6. SENÃO
7.   y=y -1
8. FIM-SE
9. FIM-SE
    
```



No algoritmo ao lado, X é declarado na linha 1 e usado pela 1ª vez na linha 4, a qual não é de inicialização. A variável x pode ficar com “lixo” em uma nova rodada do programa e causar uma falha – deve-se corrigir o algoritmo, criando a inicialização da variável antes do uso.

A primeira linha que usa y, após a declaração, é a de leitura do seu valor (inicialização) – isso está correto!

## COMO PLANEJAR UM CASO DE TESTE



**CAMINHOS MÍNIMOS:** aplicável aos testes **Funcionais** de nível **Unitário** e requer conhecimento da lógica interna (**Caixa Branca**) - técnica criada por Mc Cabe.

Avalia os pontos de decisão do software, os quais representam interpretações lógicas que desviam o curso do programa para trechos diferentes do código.

Usa a métrica de Complexidade Ciclomática que se baseia no princípio de que a maior parte dos problemas de um software ocorre em função dos desvios lógicos que levam a caminhos diferentes das linhas de programação.

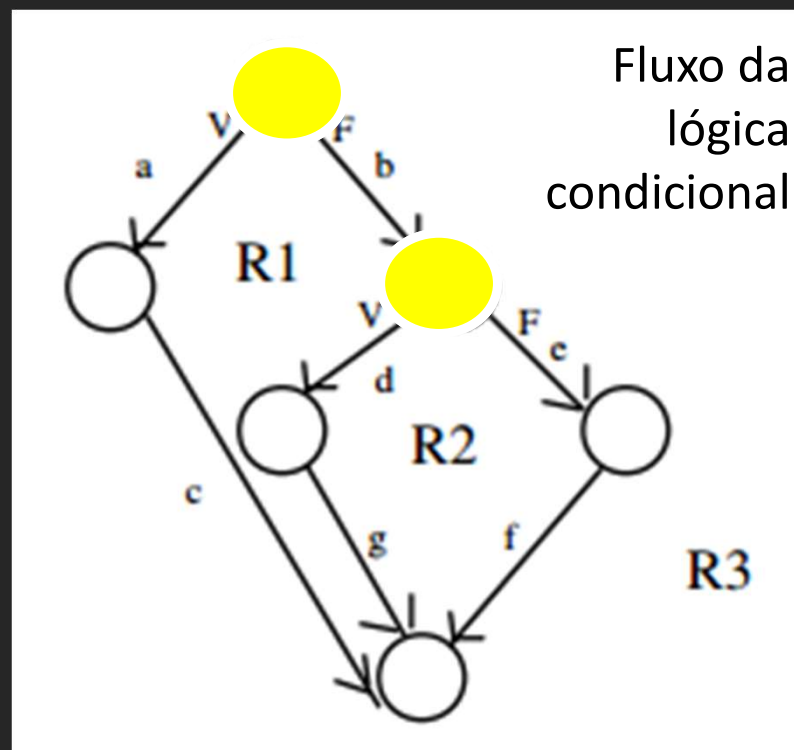
Pela técnica de McCabe, vamos garantir que cada trecho do código criado seja percorrido pelo menos 1 vez em testes.

## COMO PLANEJAR UM CASO DE TESTE

**COMPLEXIDADE CICLOMÁTICA:** conhecido como Teste dos Caminhos Mínimos.

Avalia os pontos de decisão do software, os quais representam interpretações lógicas que desviam o curso do programa para trechos diferentes do código.

Método criado pelo matemático norte-americano Thomas McCabe baseado na teoria de grafos - consiste essencialmente em fazer com que os casos de teste sejam gerados de forma a fazer com que o fluxo do programa passe por um número mínimo de caminhos entre a entrada e a saída do programa, sem o risco de ocorrerem redundâncias.

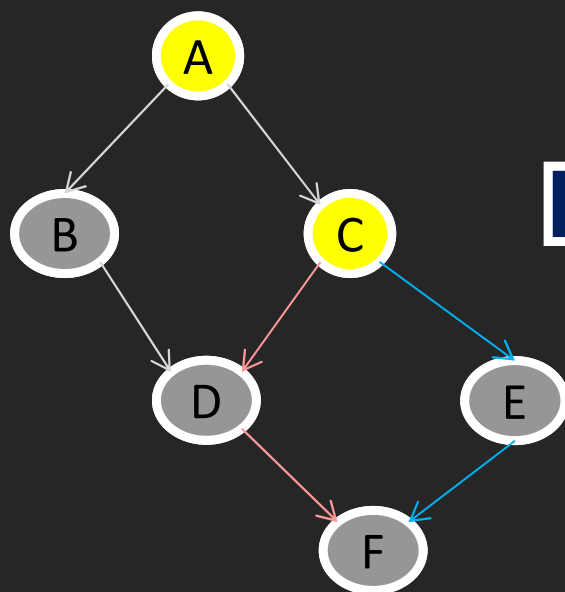


## COMO PLANEJAR UM CASO DE TESTE

### COMPLEXIDADE CICLOMÁTICA

Para descobrir o número de testes a realizar:

- No grafo de controle identificamos os chamados caminhos independentes.
- **Contar o número de estruturas de decisão no programa e somar 1. No exemplo da figura abaixo, temos 2 "if". Logo, o número de caminhos independentes é 3.**
- Cada caminho obtido dará origem a um caso de teste e quanto maior o número de pontos de decisão lógica, mais complexos os testes e a própria aplicação.



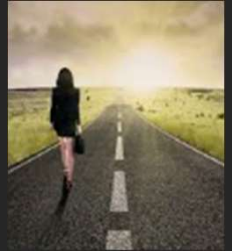
Neste caso, cada nó representa uma linha do Algoritmo avaliado, onde estão em destaque amarelo, as linhas que envolvem decisão lógica (como ifs, DoWhile...)

Existem 3 caminhos para a lógica da aplicação, partindo da primeira linha de código:

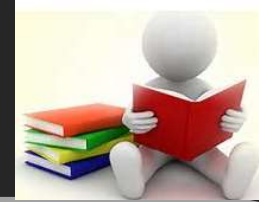
C1-Caminho 1 = A, B, D, F

C2-Caminho 2 = A, C, D, F

C3-Caminho 3 = A, C, E, F



## ESTUDO DE CASO SIMULADO



Analise se tem algum problema nesse algoritmo!

1. SE  $x = \text{False}$  ENTÃO
2.    $y = y + 1$
3. SENÃO
4.    $y = y - 1$
5. SE  $w + z \geq 72$  ENTÃO
6.    $y = y + 2$
7. SENÃO
8.    $y = y - 2$
9. FIM-SE
10. FIM-SE
11. Retorna  $y$

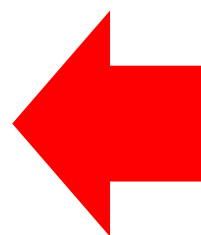
## ESTUDO DE CASO SIMULADO



Analise se tem algum problema nesse algoritmo!

```

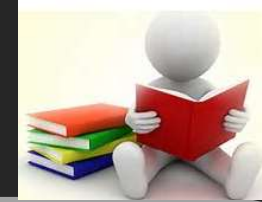
2. SE x = False ENTÃO
3.   y = y + 1
4. SENÃO
5.   y = y - 1
6. SE w + z >= 72 ENTÃO
7.   y = y + 2
8. SENÃO
9.   y = y - 2
10. FIM-SE
11. FIM-SE
12. Retorna y
    
```



Falta  
inicializar x,  
y, w e z



## ESTUDO DE CASO SIMULADO



Desenhe o grafo de caminhos de teste e calcule a Complexidade Ciclomática do trecho de algoritmo que foi passado para você avaliar.  
Calcule a Complexidade Ciclomática da transação.

1.ProgX (x: booleano, y:inteiro, z: inteiro, w:inteiro)

2.SE x = False ENTÃO

3. y= y +1

4.SENÃO

5. y=y -1

6. SE w+z >= 72 ENTÃO

7. y=y+2

8. SENÃO

9. y=y-2

10. FIM-SE

11.FIM-SE

12. Retorna y

Resolvido o  
problema de  
inicialização!

## ESTUDO DE CASO SIMULADO



Desenhe o grafo de caminhos de teste e calcule a Complexidade Ciclomática do trecho de algoritmo que foi passado para você avaliar.  
Calcule a Complexidade Ciclomática da transação.

1. ProgX (x: booleano, y: inteiro, z: inteiro, w: inteiro)

2. SE x = False ENTÃO

3. y = y + 1

4. SENÃO

5. y = y - 1

6. SE w + z >= 72 ENTÃO

7. y = y + 2

8. SENÃO

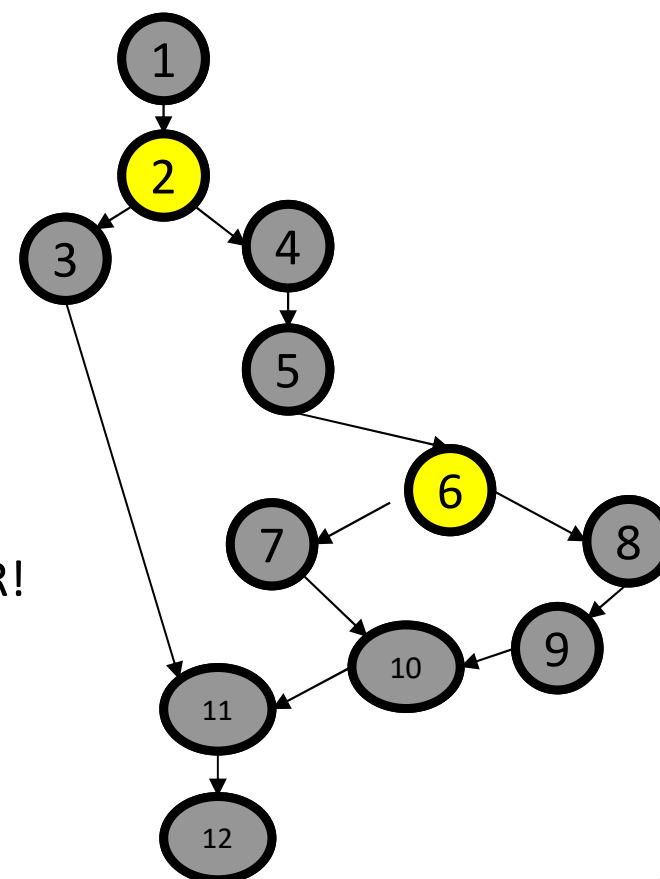
9. y = y - 2

10. FIM-SE

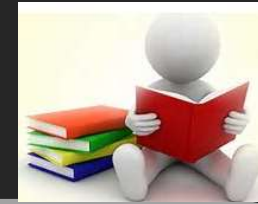
11. FIM-SE

12. Retorna y

3 TESTES A CONSTRUIR!



## ESTUDO DE CASO SIMULADO



Crie agora, um conjunto de Casos de Testes com valores de Input e Output previsto para todas as variáveis envolvidas na transação, utilizando a regra de Caminhos Mínimos.

1.ProgX (x: booleano, y:inteiro, z: inteiro, w:inteiro)

2.SE x = False ENTÃO

3. y= y +1

4.SENÃO

5. y=y -1

6. SE w+z >= 72 ENTÃO

7. y=y+2

8. SENÃO

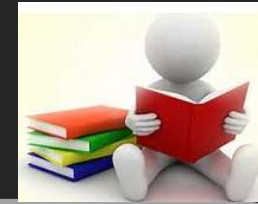
9. y=y-2

10. FIM-SE

11.FIM-SE

12. Retorna y

## ESTUDO DE CASO SIMULADO



Crie agora, um conjunto de Casos de Testes com valores de Input e Output previsto para todas as variáveis envolvidas na transação, utilizando a regra de Caminhos Mínimos.

1.ProgX (x: booleano, y:inteiro, z: inteiro, w:inteiro)

2.SE x = False ENTÃO

3. y= y +1

4.SENÃO

5. y=y -1

6. SE w+z >= 72 ENTÃO

7. y=y+2

8. SENÃO

9. y=y-2

10. FIM-SE

11.FIM-SE

12. Retorna y

### TESTE 1

Inputs:

-x=false

-y=0

-z=0

-w=0

Outputs:

-y=1

### TESTE 2

Inputs:

-x=true

-y=0

-z=0

-w=0

Outputs:

-y=-3

### TESTE 3

Inputs:

-x=true

-y=0

-z=100

-w=100

Outputs:

-y= 1

## ESTUDO DE CASO SIMULADO



ProgX (x: booleano, y:inteiro, z: inteiro, w:inteiro)

R=0

**IF x = False THEN**

y= y +1

**ELSE**

y=y -1

**IF w+z >= 72 THEN**

y=y+2

**ELSE**

y=y-2

**ENDIF**

**ENDIF**

r = r+y

**IF r = w+z THEN**

r = 0

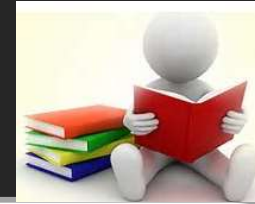
**ENDIF**

Retorna r

Desenhe o grafo de caminhos de teste e calcule a Complexidade Ciclomática do trecho de algoritmo que foi passado para você avaliar.

Calcule a Complexidade Ciclomática da transação.

## ESTUDO DE CASO SIMULADO

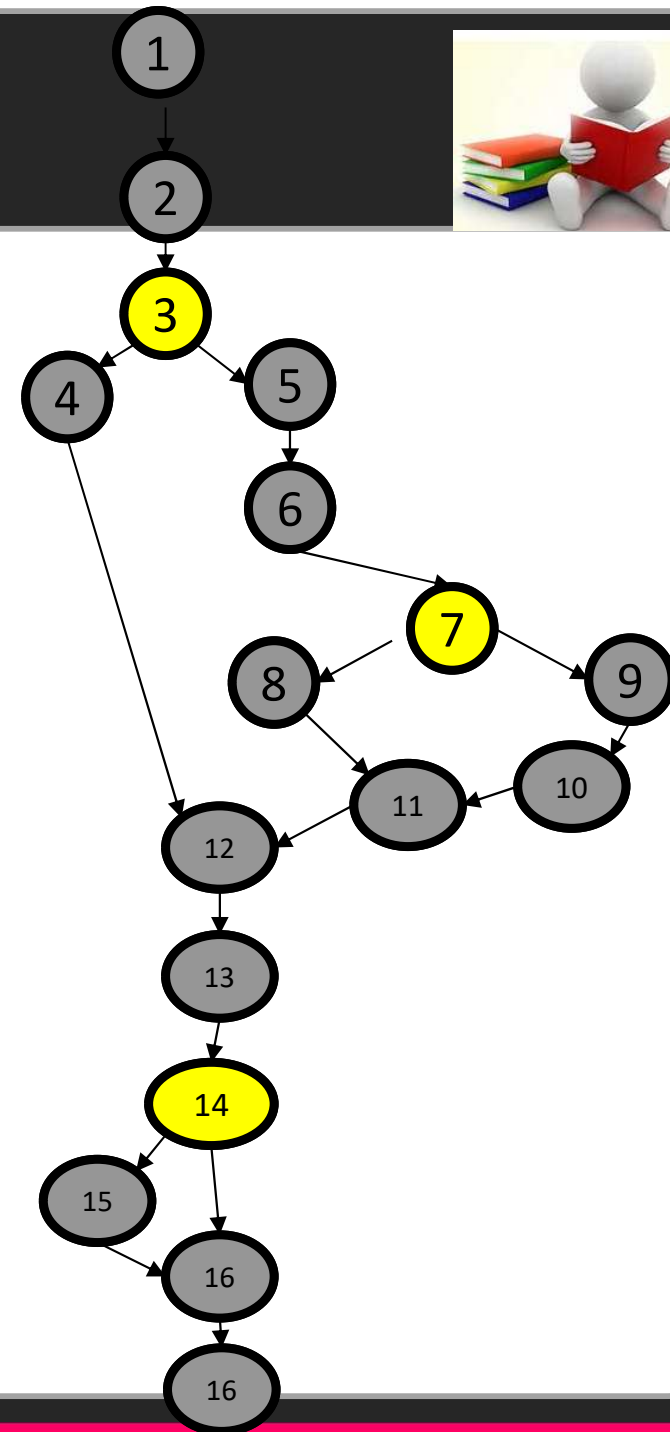


```

1.ProgX (x: booleano, y:inteiro, z: inteiro, w:inteiro)
2.R=0
3.IF x = False THEN
4.  y= y +1
5. ELSE
6.  y=y -1
7.  IF w+z >= 72 THEN
8.    y=y+2
9.  ELSE
10.    y=y-2
11.  ENDIF
12. ENDIF

13.r = r+y

14.IF r = w+z THEN
15.  r = 0
16.ENDIF
17.Retorna r
    
```



## ESTUDO DE CASO SIMULADO



ProgX (x: booleano, y:inteiro, z: inteiro, w:inteiro)

R=0

**IF x = False THEN**

y= y +1

**ELSE**

y=y -1

**IF w+z >= 72 THEN**

y=y+2

**ELSE**

y=y-2

**ENDIF**

**ENDIF**

r = r+y

**IF r = w+z THEN**

r = 0

**ENDIF**

Retorna r

Crie os casos de teste unitários de caixa branca, com base na regra de caminhos mínimos.

## ESTUDO DE CASO SIMULADO



ProgX (x: booleano, y:inteiro, z: inteiro, w:inteiro)

R=0

**IF x = False THEN**

y= y +1

ELSE

y=y -1

**IF w+z >= 72 THEN**

y=y+2

ELSE

y=y-2

ENDIF

ENDIF

r = r+y

**IF r = w+z THEN**

**r = 0**

ENDIF

Retorna r

### TESTE 1

INPUTS:

x=false

y=0

z=0

w=0

**r=0**

OUTPUTS:

**r=1**

### TESTE 2

INPUTS:

x=true

y=0

z=0

w=0

**r=0**

OUTPUTS:

**r= -3**

### TESTE 3

INPUTS:

x=true

y=0

z=50

w=50

**r=0**

OUTPUTS:

**r= 1**

### TESTE 4

INPUTS:

x=false

y=-1

z=0

w=0

**r=0**

OUTPUTS:

**r= 1**



## ESTUDO DE CASO SIMULADO



Crie o grafo de caminhos e os casos de teste unitários de caixa branca, com base na regra de Mc Cabe, para o algoritmo de programa que lhe foi disponibilizado para teste.

Caso X

=1 entao

$X=X+1$ ;

=2 entao

$X=X+2$ ;

=3 então

$X=X+3$ ;

=4 então

$X=X+5$ ;

Fim-Caso

$X=X-1$

## TESTE DE SOFTWARE

### ATIVIDADE PRÁTICA

#### PROVA DE FOGO NÚMERO 01



*Criem testes com base na Complexidade Ciclomática de McCabe para o programa que receberam da dupla ao lado , considerando o método de controle!*

*Como o teste é de Caixa Branca, vocês vão se basear no Algoritmo que foi passado pelo professor, e que contém a lógica correta (esperada) para a aplicação.*

*Vocês não podem modificar o programa dos colegas!*

*Criem os Cartões de teste ou Linhas em uma planilha, com o conteúdo dos cartões: Identificação; Objetivo do teste; Preparação necessária de dados; Dados de entrada; Dados de saída esperados; Passos para executar o teste; Resultado alcançado na aplicação do teste (passou ou falhou).*

## TESTE DE SOFTWARE

## ATIVIDADE PRÁTICA

## PROVA DE FOGO NÚMERO 01



*Mostrem a planilha/quadro de resultados para a dupla que criou o código e pergunte a eles se vocês encontraram todos os defeitos “plantados”.*

*A dupla que programou o código deve dizer apenas se todas as falhas foram pegas ou não, sem detalhar qual defeito de software não foi identificado!*

## TESTE DE SOFTWARE

## ATIVIDADE PRÁTICA

## PROVA DE FOGO NÚMERO 01

*Transformem agora os Casos de Teste em programas de teste em JUNIT de forma a tornar a reexecução automatizada!*



# JUnit

## ESTUDO DE CASO SIMULADO



A equipe da GD está radiante com a perspectiva de tornar seus testes mais objetivos.

Além disso, quando Dilan perguntou quantos testes precisariam ser feitos para testar um programa minimamente e recebeu a resposta quase imediata do desenvolvedor “5”, ele achou que estivesse chutando e depois se espantou ao saber que por trás do número, tinha um raciocínio lógico inteligente, proporcionado pelo método de Caminhos Mínimos, desenvolvido pelo estudioso Mc Cabe.

Diante desse entusiasmo inicial, Consuelo apontou que, a avaliação de inicialização de variáveis e o modelo de Caminhos Mínimos, baseado na Complexidade Ciclomática da lógica da aplicação não cobrem 100% das possibilidades de falha de programas de computador.

Ela continuou sua doutrinação pela qualidade apresentando novos métodos para definir Casos de Teste com Técnica de Caixa Branca, do Tipo Funcionais e do Nível Unitário para construir a Estratégia de Testes.

## COMO PLANEJAR UM CASO DE TESTE



**Os testes de McCabe não esgotam todos os problemas, embora permita testar todas as linhas de código pelo menos uma vez!**

**Vamos conhecer outras técnicas que vão ajudar a detectar outras falhas que McCabe não conseguiu explorar!**



## COMO PLANEJAR UM CASO DE TESTE



**TESTE DE ENLACE:** aplicável aos testes **Funcionais** de nível **Unitário** e requer conhecimento da lógica interna (**Caixa Branca**).

Avalia as execução de trechos do código que estão em Loop.

Ele **complementa o Teste de Complexidade Ciclomática** fazendo com que um mesmo ponto de decisão seja avaliado em três condições mínimas:

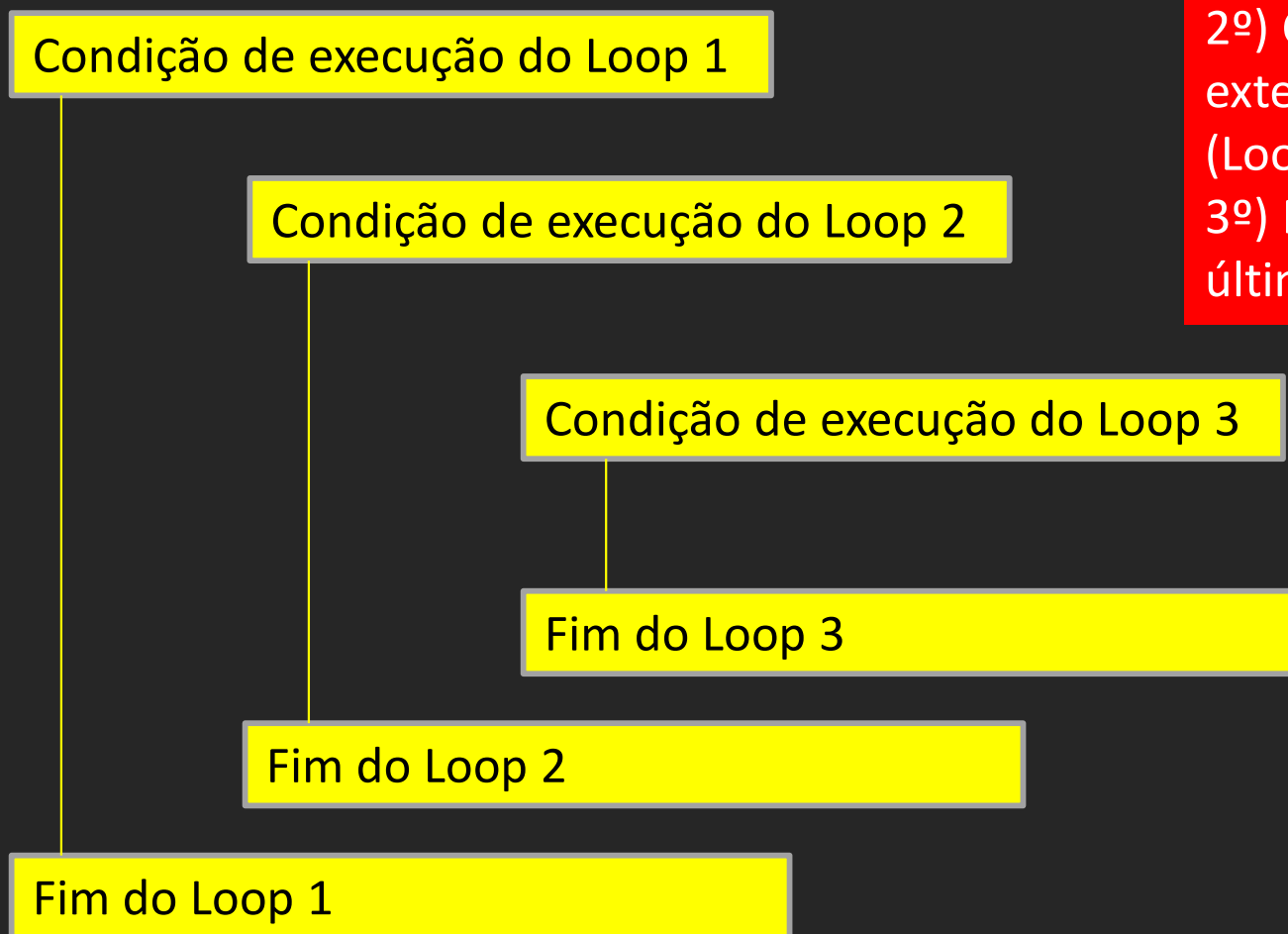
- **Não entrar no Loop;**
- **Ficar no Loop ao menos uma vez;**
- **Realizar ao menos um retorno no ciclo, com reexecução do mesmo trecho de código.**

# COMO PLANEJAR UM CASO DE TESTE

## TESTE DE ENLACE



- 1º) Cria-se testes para o Loop mais interno (Loop 3)
- 2º) Cria-se testes para o Loop externo imediatamente superior (Loop 2)
- 3º) Procede-se com o passo 2º até o último nível de loop (mais externo)





## COMO PLANEJAR UM CASO DE TESTE



### TESTE DE ENLACE – Aprendendo na prática

Quantos e quais casos de testes você precisa desenvolver para o código a seguir, baseando-se no conceito de Teste de Laços?

```
LER (x, y, z, w);
```

```
REPETIR
```

```
  x = x + 1
```

```
REPETIR
```

```
  y = y + 1
```

```
  z = z + 100
```

```
ENQUANTO w < 50
```

```
  w = w + 1
```

```
FIM-ENQUANTO
```

```
ATÉ y > 20
```

```
ATÉ x >= 100
```

## COMO PLANEJAR UM CASO DE TESTE

### TESTE DE ENLACE – Aprendendo na prática



**Os Loops podem ser testados individualmente, controlando-se manualmente as variáveis de influência.**  
**Atentar que se a condição vem antes da sentença a ser executada, temos 3 testes mínimos, caso contrário, teremos 2 testes.**

REPETIR

$x = x + 1$

REPETIR

$y = y + 1$

$z = z + 100$

ATÉ  $y > 20$

ENQUANTO  $w < 50$

$w = w + 1$

FIM-ENQUANTO

ATÉ  $x \geq 100$

1º) Caso de teste da condição/loop 1

\*Testar pelo menos 2 casos (faz sem sair e faz e sai)

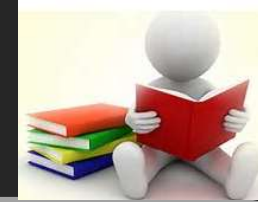
2º) Caso de teste da condição/loop 3

\*Testar pelo menos 2 casos (faz sem sair e faz e sai)

3º) Caso de teste da condição/loop 2

\*Testar pelo menos 3 casos (nem entra, faz e sai ou faz e não sai)

## ESTUDO DE CASO SIMULADO



Defina os Casos de Teste com valores e entrada de dados e saída prevista para cada variável do programa que será desenvolvido com base no algoritmo a seguir.

REPETIR

$x = x + 1$

REPETIR

$y = y + 1$

$z = z + 100$

ATÉ  $y > 20$

ENQUANTO  $w < 50$

$w = w + 1$

FIM-ENQUANTO

ATÉ  $x \geq 100$

**ELABORE OS CASOS DE TESTES UNITÁRIOS DE CAIXA BRANCA E FUNCIONAIS, BASEANDO-SE NA AVALIAÇÃO DE ENLACES!**

**RELACIONE:**

- Identificação/número do Caso de Teste;
- Descrição do objetivo
- Dados de entrada previstos
- Dados de saída previstos
- Preparação, se houver necessidade de manipular previamente dados de bancos de dados para que os testes funcionem



## SIMULAÇÃO DO EFEITO DOS TESTES:

Algoritmo Calc-Numeros  
 Recebe parâmetros int X; int Y  
 Enquanto X < Y  
     Y := Y+X-1  
 Fim-enquanto  
 Fim-algoritmo

Especificação original feita pela equipe de projetistas ( BASE PARA ELABORAÇÃO DE CASOS DE TESTES)

### TESTE CONTROLADO

Caso de teste: 001  
 Objetivo: avaliar Calc-Numeros  
 Preparação: nenhuma  
 Entradas: X = 0; Y = 1  
 Saída esperada: X=0; Y=0

### TESTE CONTROLADO

Caso de teste: 002  
 Objetivo: avaliar Calc-Numeros  
 Preparação: nenhuma  
 Entradas: X = 0; Y = 0  
 Saída esperada: X=0; Y=0



## SIMULAÇÃO DO EFEITO DOS TESTES:

PERCEBA QUE OS TESTES DEVEM SEMPRE SER FEITOS COM BASE NO PROJETO DO SOFTWARE E ALGORITMOS!

CRIAR TESTES LENDO O CÓDIGO FINAL DA APLICAÇÃO LEVA A ERROS POIS, O CÓDIGO PODE NÃO TER SIDO ESCRITO DE FORMA FIEL ÀS REGRAS DE NEGÓCIO QUE SÃO ESPERADAS.



Código fonte que foi produzido e deveria ter seguido o algoritmo (**sobre ele serão aplicados os testes**)

Algoritmo Calc-Numeros

Recebe parâmetros int X; int Y

Enquanto X < Y

Y := Y+X-1

Fim-enquanto

Retorna Y

Fim-algoritmo

```
Int void Calc-Numeros (int X; int Y)
{
    Dowhile X <= Y {
        Y = Y+X+1;
    }
    Return Y;
}
```

## TESTE CONTROLADO

Caso de teste: 001

Objetivo: avaliar Calc-Numeros

Preparação: nenhuma

Entradas: X = 0; Y = 1

Saída esperada: Y=0



**QUANDO APLICADO SOBRE O PROGRAMA, ELE FICA EM UM LOOP INFINITO! APONTA FALHA!**

**VALIDAMOS A LÓGICA QUE AJUDA A SAIR DO LOOP!**



## SIMULAÇÃO DO EFEITO DOS TESTES:

Algoritmo Calc-Numeros

Recebe parâmetros int X; int Y

Enquanto X < Y

Y := Y+X-1

Fim-enquanto

Retorna Y

Fim-algoritmo

Int void Calc-Numeros (int X; int Y)

```
{
  Dowhile X <= Y {
    Y = Y+X+1;
  }
  Return Y;
}
```

### TESTE CONTROLADO

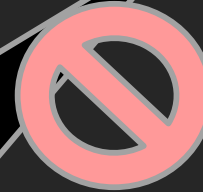
Caso de teste: 002

Objetivo: avaliar Calc-Numeros

Preparação: nenhuma

Entradas: X = 0; Y = 0

Saída esperada: X=0; Y=0



**QUANDO APLICADO SOBRE O PROGRAMA, ELE ENTRA NA SENTENÇA PÓS-CONDIÇÃO QUANDO NÃO DEVERIA! APONTA FALHA!**

**VALIDAMOS A CONDIÇÃO DE ENTRADA DO LOOP!**

## TESTE DE SOFTWARE

### ATIVIDADE PRÁTICA

#### PROVA DE FOGO NÚMERO 02



*Criem testes com base em Avaliação de Enlaces para o programa que receberam da dupla ao lado , considerando o método de controle!*

*Como o teste também é de Caixa Branca, vocês vão se basear no Algoritmo que foi passado pelo professor, e que contém a lógica correta (esperada) para a aplicação.*

*Mais uma vez, vocês não podem modificar o programa dos colegas!*

*Criem os Cartões de teste ou Linhas em uma planilha, com o conteúdo dos cartões: Identificação; Objetivo do teste; Preparação necessária de dados; Dados de entrada; Dados de saída esperados; Passos para executar o teste; Resultado alcançado na aplicação do teste (passou ou falhou).*



## TESTE DE SOFTWARE

## ATIVIDADE PRÁTICA



## PROVA DE FOGO NÚMERO 02



*Mostrem a planilha/quadro de resultados para a dupla que criou o código e pergunte a eles se vocês encontraram todos os defeitos “plantados”.*

*A dupla que programou o código deve dizer apenas se todas as falhas foram pegas ou não, sem detalhar qual defeito de software não foi identificado!*

## COMO PLANEJAR UM CASO DE TESTE



**TESTE DE LIMITES:** aplicável aos testes **Funcionais** de nível **Unitário** e requer conhecimento da lógica interna (**Caixa Branca**).

Criado por Meyers, avalia valores que influenciam as decisões.

Ajuda a definir quais os valores a serem informados para provocar os desvios no programa de aplicação.

Pode fazer com que um caminho lógico do Grafo de Mc Cabe ou do Ciclo de Enlace seja executado mais de uma vez. Por essa razão, concluímos que o Teste de Limites incrementa os Casos de Testes mínimos de Mc Cabe, criando um novo conjunto de testes que ajudam a cercar defeitos no software.

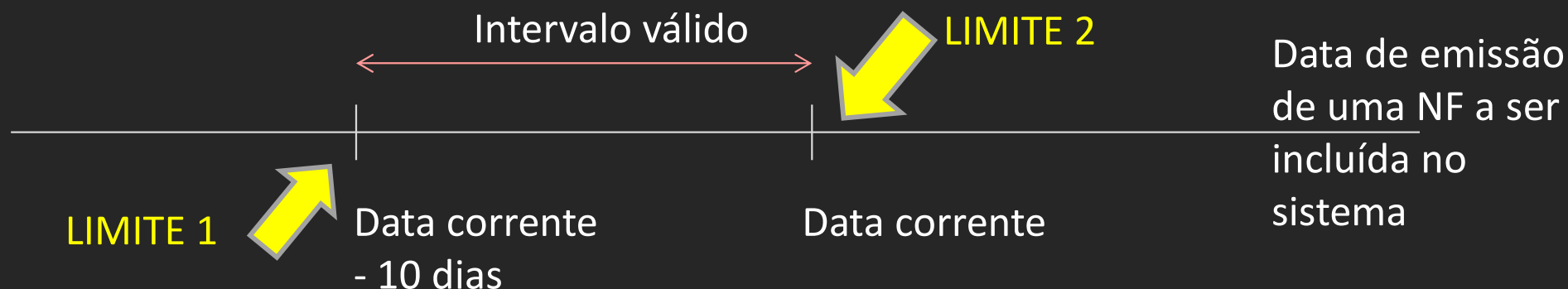
## COMO PLANEJAR UM CASO DE TESTE



### TESTE DE LIMITES

Os testes vão focar somente a entrada de dados próximas ao limite, sendo os demais testes irrelevantes:

- 1 teste com o 1º valor fora do limite inferior;
- 1 teste com o 1º valor dentro do limite inferior;
- 1 teste com o 1º valor fora do limite superior;
- 1 teste com o 1º valor dentro do limite superior.



## COMO PLANEJAR UM CASO DE TESTE



## TESTE DE LIMITES

Por exemplo, considerando o algoritmo (em verde) que foi usado para gerar o programa (em vermelho), vamos criar os casos de testes , levando em conta o algoritmos e aplicar os testes sobre o código para identificar falhas.

```
LER(dataEmissao)
SE Sysdate-10 <= dataEmissao <= Sysdate
ENTAO ESCREVE("NOTA FISCAL PERMITIDA");
```

Algoritmo base para criar os testes

Base  
para

```
Scanner ent = new Scanner(System.in);
Date dataEmissao = ent.next();
if ((dataEmissao > Sysdate-10 ) && (dataEmissao < Sysdate)){
    System.out.println ("NOTA FISCAL PERMITIDA");
}
```

Código sobre o  
qual serão  
aplicados os testes

## COMO PLANEJAR UM CASO DE TESTE



### TESTE DE LIMITES

Se usarmos o método de Caminhos Mínimos, **poderíamos criar os dois casos de testes nos cartões verdes e aplica-los – o resultado não vai apontar falhas (os outputs serão os esperados, mediante os inputs).**

```
LER(dataEmissao)
SE Sysdate-10 <= dataEmissao <= Sysdate
ENTAO ESCREVE("NOTA FISCAL PERMITIDA");
```

Gera

TESTE 1 – CAMNHO MIN.

.INPUT:

...dataEmissao= 15/10/2019

OUTPUT:

...dataEmissao= 15/10/2019

..mensagem APARECE

Aplica

```
Scanner ent = new Scanner(System.in);
Date x = ent.next();
if ((dataEmissao > Sysdate-10 ) && (dataEmissao < Sysdate)){
    System.out.println ("NOTA FISCAL PERMITIDA");
}
```

TESTE 2 – CAMNHO MIN.

.INPUT:

...dataEmissao= 25/10/2019

OUTPUT:

...dataEmissao= 25/10/2019

..mensagem NÃO APARECE

## COMO PLANEJAR UM CASO DE TESTE TESTE DE LIMITES

Os testes com base em Limites serão 4 e dois deles não vão dar a resposta esperada, apontando que o programa tem falhas em relação ao algoritmo!

```
LER(dataEmissao)
SE Sysdate-10 <= dataEmissao <= Sysdate ENTAO ESCRIVE("NOTA FISCAL
PERMITIDA");
```

```
if ((dataEmissao > Sysdate-10 ) && (dataEmissao < Sysdate)){
    sysout ("NOTA FISCAL PERMITIDA");
}
```

### TESTE 1 - passou

```
.INPUT
...dataEmissao = 12/10/2019
.OUTPUT
...dataEmissao = 12/10/2019
...não exibe mensagem
```

### TESTE 2 - falhou

```
.INPUT
...dataEmissao = 13/10/2019
.OUTPUT
...dataEmissao = 13/10/2019
...exibe mensagem
```

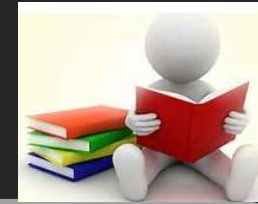
### TESTE 3 - falhou

```
.INPUT
...dataEmissao = 23/10/2019
.OUTPUT
...dataEmissao = 23/10/2019
...exibe mensagem
```

### TESTE 4 - passou

```
.INPUT
...dataEmissao = 24/10/2019
.OUTPUT
...dataEmissao = 24/10/2019
...não exibe mensagem
```

## ESTUDO DE CASO SIMULADO



Defina os Casos de Teste com valores e entrada de dados e saída prevista para cada variável do programa que será desenvolvido com base no algoritmo a seguir, considerando o método de avaliação de Limites

SE  $12 \leq a \leq 19$  ENTÃO

$a = a + 1$

SENÃO

SE  $a < b$  ENTÃO

$a = a + b$

FIM-SE

FIM-SE

Compare o número de testes por esse critério em relação ao critério de complexidade ciclomática

**ELABORE OS CASOS DE TESTES UNITÁRIOS DE CAIXA BRANCA E FUNCIONAIS, BASEANDO-SE NA AVALIAÇÃO DE LIMITES!**

**RELACIONE:**

- Identificação/número do Caso de Teste;
- Descrição do objetivo
- Dados de entrada previstos
- Dados de saída previstos
- Preparação, se houver necessidade de manipular previamente dados de bancos de dados para que os testes funcionem

## EXEMPLOS DOS TESTES:

### Algoritmo Calc-Numeros

Recebe parâmetros int X; int Y

int W

W := X+Y

Se  $0 < W < 10$

W := W -5

Fim-se

Retorna W

Fim-algoritmo

### TESTE CONTROLADO

Caso de teste: 001

Objetivo: avaliar Calc-Numeros

Preparação: nenhuma

Entradas: X = 0; Y = 0

Saída esperada: W=0

### TESTE CONTROLADO

Caso de teste: 004

Objetivo: avaliar Calc-Numeros

Preparação: nenhuma

Entradas: X = 9; Y = 1

Saída esperada: W=10

### TESTE CONTROLADO

Caso de teste: 003

Objetivo: avaliar Calc-Numeros

Preparação: nenhuma

Entradas: X = 9; Y = 0

Saída esperada: W=4

### TESTE CONTROLADO

Caso de teste: 002

Objetivo: avaliar Calc-Numeros

Preparação: nenhuma

Entradas: X = 1; Y = 0

Saída esperada: W = -4





## SIMULAÇÃO DO EFEITO DOS TESTES:

### Algoritmo base para criar testes

```

Algoritmo Calc-Numeros
  Recebe parâmetros int X; int Y
  int W
  W := X+Y
  Se 0 < W < 10
    W := W -5
  Fim-se
  Retorna W
Fim-algoritmo
    
```

### Código fonte a testar

```

Int void Calc-Numeros(int X; int Y)
{
  int W;
  W = X+Y;
  If 0 <= W < 10
    then {W := W -5
  }
  Return W
}
    
```

### TESTE CONTROLADO

Caso de teste: 001

Objetivo: avaliar Calc-Numeros

Preparação: nenhuma

Entradas: X = 0; Y = 0

Saída esperada: W=0



**QUANDO APLICADO SOBRE O PROGRAMA, ELE RETORNA UM VALOR DIFERENTE DO ESPERADO!**

**RETORNA W = -5, APONTANDO FALHA!**



## EFEITO DE NÃO APLICAR A REGRA DE LIMITES:

### Algoritmo Calc-Numeros

```

Recebe parâmetros int X; int Y
int W
W := X+Y
Se 0 < W < 10
    W := W -5
Fim-se
Retorna W
Fim-algoritmo
    
```

```

Int void Calc-Numeros(int X; int Y)
{
    int W;
    W = X+Y;
    If 0 <= W < 10
        then {W := W -5
    }
    Return W
}
    
```

TESTE CONTROLADO USANDO DADOS QUAISQUER QUE GARANTAM ENTRADA NA CONDIÇÃO

Caso de teste: 001

Objetivo: avaliar Calc-Numeros

Preparação: nenhuma

Entradas: X = 5; Y = 2

Saída esperada: W=2

**O TESTE AGORA NÃO MOSTRA A FALHA DE PROGRAMAÇÃO!**

**RETORNA W = 2, DIZENDO QUE O PROGRAMA ESTÁ PERFEITO, QUANDO NÃO ESTÁ!**

**ESSE TESTE É INÚTIL!**

## TESTE DE SOFTWARE

### ATIVIDADE PRÁTICA

#### PROVA DE FOGO NÚMERO 03



*Criem testes com base em Avaliação de Limites para o programa que receberam da dupla ao lado , considerando o método de controle!*

*Como o teste também é de Caixa Branca, vocês vão se basear no Algoritmo que foi passado pelo professor, e que contém a lógica correta (esperada) para a aplicação.*

*Mais uma vez, vocês não podem modificar o programa dos colegas!*

*Criem os Cartões de teste ou Linhas em uma planilha, com o conteúdo dos cartões: Identificação; Objetivo do teste; Preparação necessária de dados; Dados de entrada; Dados de saída esperados; Passos para executar o teste; Resultado alcançado na aplicação do teste (passou ou falhou).*

## TESTE DE SOFTWARE

## ATIVIDADE PRÁTICA

## PROVA DE FOGO NÚMERO 03



*Mostrem a planilha/quadro de resultados para a dupla que criou o código e pergunte a eles se vocês encontraram todos os defeitos “plantados”.*

*A dupla que programou o código deve dizer apenas se todas as falhas foram pegas ou não, sem detalhar qual defeito de software não foi identificado!*

## COMO PLANEJAR UM CASO DE TESTE



### TESTE DE PARTIÇÃO/CONDIÇÃO DE EQUIVALÊNCIA:

aplicável aos testes **Funcionais** de nível **Unitário** e requer conhecimento da lógica interna (**Caixa Branca**).

Avalia grupos de valores que implicam nas mesmas decisões.

Assim como o estudo de Limites, ajuda a definir quais os valores a serem informados para provocar os desvios no programa de aplicação. A diferença para a abordagem de Limites é que na regra de Partição de Equivalência, **podemos trabalhar com valores não numéricos!**

Os critérios de teste definidos pelo modelo de Condição e Equivalência podem fazer com que um caminho lógico do Grafo de McCabe ou do Ciclo de Enlace seja executado mais de uma vez e amplia as possibilidades que o teste de Limites atende. Por essa razão, concluímos que o este modelo teste **incrementa os testes desenvolvidos pelos outros modelos vistos anteriormente.**

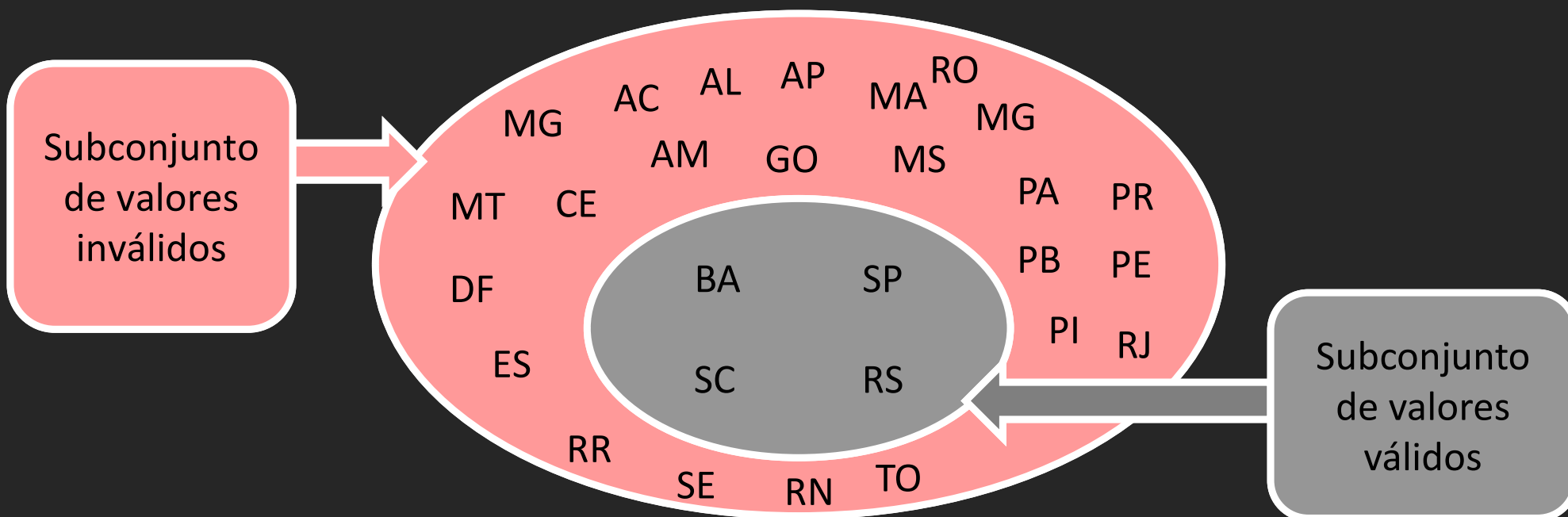
## COMO PLANEJAR UM CASO DE TESTE

### TESTE DE PARTIÇÃO/CONDIÇÃO DE EQUIVALÊNCIA



Devem ser observados quais os conjuntos de dados que levam ao mesmo encaminhamento.

Devemos testar pelo menos um elemento de cada conjunto distinto, sendo necessário testar todos os elementos do conjunto válido.



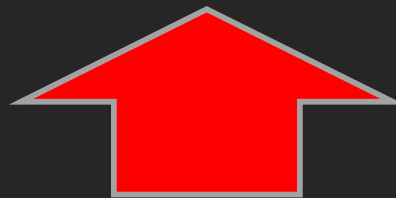
## COMO PLANEJAR UM CASO DE TESTE

### TESTE DE PARTIÇÃO/CONDIÇÃO DE EQUIVALÊNCIA



**EXEMPLO:** Se a especificação do software trata a validade código de estado da federação brasileira, considerando com válidos BA, SP, SC e RS, identificamos:

- Conjunto da **classe válida**: qualquer uma entre BA, SP, SC e RS.
- Conjunto da **classe inválida**: qualquer um que não seja BA, SP, SC ou RS .



Deve ser criado 1 teste PARA CADA VALOR DO CONJUNTO VÁLIDO e APENAS MAIS 1 para qualquer valor do CONJUNTO INVÁLIDO

## ESTUDO DE CASO SIMULADO



Defina os Casos de Teste com valores e entrada de dados e saída prevista para cada variável do programa que será desenvolvido com base no algoritmo a seguir, considerando o método de avaliação de Condição e Equivalência

```
SE Serie_Nota_Fiscal = "U" OU "U1"
  ISS := "Isento"
SENÃO
  ISS := "Tributado"
FIM-SE
```

Compare o número de testes por esse critério em relação ao critério de complexidade ciclomática

**ELABORE OS CASOS DE TESTES UNITÁRIOS DE CAIXA BRANCA E FUNCIONAIS, BASEANDO-SE NA AVALIAÇÃO DE LIMITES!**

**RELACIONE:**

- Identificação/número do Caso de Teste;
- Descrição do objetivo
- Dados de entrada previstos
- Dados de saída previstos
- Preparação, se houver necessidade de manipular previamente dados de bancos de dados para que os testes funcionem




## COMO AUTOMATIZAR OS TESTES UNITÁRIOS

As ferramentas de criação e execução de **testes de SCRIPT** são as recomendadas para automatizar os testes unitários.

Já aplicamos JUNIT com e sem TDD nesse tipo de automação.



Outra ferramenta muito utilizada é a  , para testar Classes JAVA e Java Script. Ela **opera de forma muito semelhante à JUNIT**, permitindo criar casos de teste isolados das Classes de aplicação, que podem ser gravados no mesmo projeto e workspace, para serem reexecutados quantas vezes for necessário. JEST trabalha com o mesmo sistema da JUNIT, de **chamar métodos de Classes em um método de teste, enviando parâmetros de input e validando outputs esperados**.

*\*Não vamos explorar JEST, em função da similaridade com JUNIT.*

## TESTE DE SOFTWARE

### ATIVIDADE PRÁTICA

#### PROVA DE FOGO NÚMERO 04



*Criem testes com base em Avaliação de Condição e Equivalência para o programa que receberam da dupla ao lado, considerando o método de controle!*

*Como o teste também é de Caixa Branca, vocês vão se basear no Algoritmo que foi passado pelo professor, e que contém a lógica correta (esperada) para a aplicação.*

*Mais uma vez, vocês não podem modificar o programa dos colegas!*

*Criem os Cartões de teste ou Linhas em uma planilha, com o conteúdo dos cartões: Identificação; Objetivo do teste; Preparação necessária de dados; Dados de entrada; Dados de saída esperados; Passos para executar o teste; Resultado alcançado na aplicação do teste (passou ou falhou).*

## TESTE DE SOFTWARE

## ATIVIDADE PRÁTICA

## PROVA DE FOGO NÚMERO 04



*Mostrem a planilha/quadro de resultados para a dupla que criou o código e pergunte a eles se vocês encontraram todos os defeitos “plantados”.*

*A dupla que programou o código deve dizer apenas se todas as falhas foram pegas ou não, sem detalhar qual defeito de software não foi identificado!*

## TESTE DE SOFTWARE

### ATIVIDADE PRÁTICA

#### PROVA DE FOGO NÚMERO 05

*Sigam as instruções em **BASE PARA EXERCÍCIOS DE TESTES***

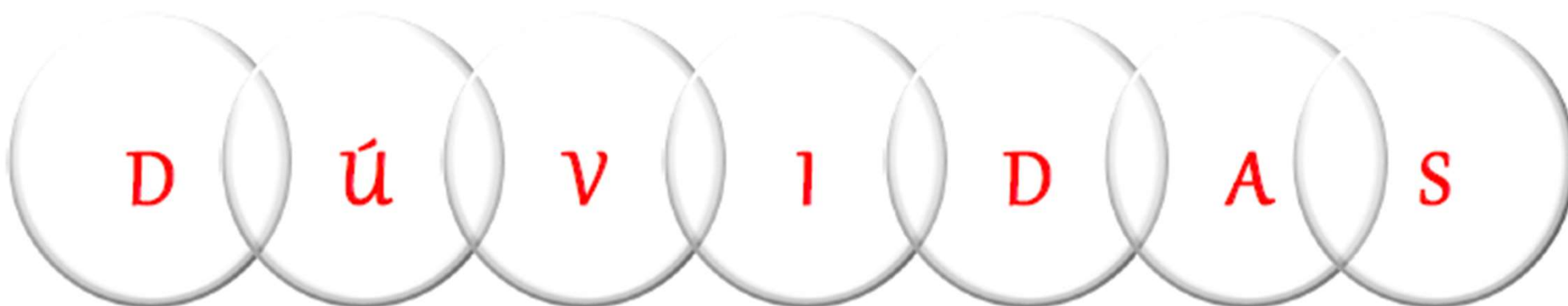


## ORGANIZAÇÃO PARA REALIZAR TESTES

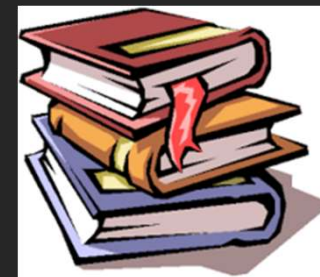


**ESCUTE O PODCAST NO CANAL DO PROFESSOR  
RIGOR NOS TESTES**

<https://youtu.be/mUMJ4VnPLxY>



## Referência bibliográficas



### BIBLIOGRAFIA :

- MOLINARI, Leonardo. Testes de Software – Produzindo Sistemas Melhores e Mais Confiáveis, 4a. Edição. Editora Erica, 2013.
- MOLINARI, Leonardo. Inovação e Automação de Testes de Software, 1ª edição. Érica, 2010.
- CMMi V3. SEI - Software Engineering Institute., USA, 2007. Disponpivel na biblioteca online da Carnegie Melon University.
- Reis, Luís Filipe Souza. ISO 9000/Auditorias de sistemas da qualidade.Editora: Érica, 1995.

## TESTE DE SOFTWARE

**Continua na próxima aula...**

PROFESSOR:  
**RENATO JARDIM PARDUCCI**

PROFRENATO.PARDUCCI@FIAP.COM.BR