INSTITUTO FEDERAL
Santa Catarina

Curso de Ciência da Computação

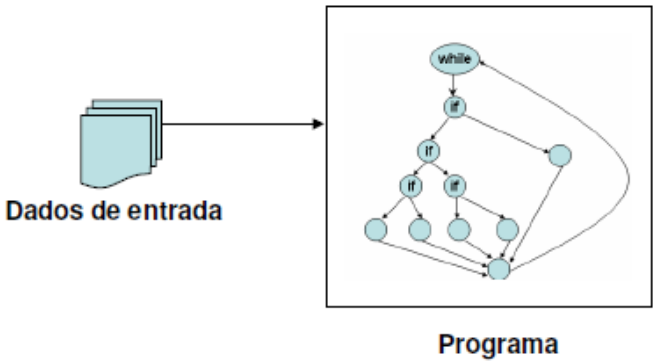
Engenharia de Software
Testes estruturais

Prof. Alexandre Perin de Souza
alexandre.perin@ifsc.edu.br

Lages (SC)

Teste estrutural

- Estruturais ou caixa-branca
  - Os primeiros testes aos quais um sistema é submetido são os testes unitários, que podem ser testes caixa-branca
  - Testes caixa-branca permitem detectar possíveis erros pela garantia de executar todos os comandos e condições do programa ao menos uma vez



Teste estrutural

- Teste unitário, abordagem caixa-branca
  - Passos:
    1. Identificar quais comandos exercitar ou testar
       - Determinar a complexidade ciclomática
    2. Definir os caminhos independentes
       - Quais comandos serão testados
    3. Criar os casos de teste
       - Entrada e saída esperada
    4. Realizar os testes

Teste estrutural

- Determinação da Complexidade Ciclomática (CC)
  - É uma medida de complexidade de programa, baseada no número de estruturas de controle (seleção e repetição)
  - Indica o número máximo de execuções necessárias para exercitar (testar) todos os comandos do programa
  - Formas de calcular:
    a) Simplificada
    b) Grafo de fluxo de programa

Teste estrutural

a) Complexidade ciclomática – cálculo simplificado

- Contam-se o número de estruturas de controle e soma-se 1
- Como estruturas de controle contam-se:
  - IF-THEN: 1 ponto
  - IF-THEN-ELSE: 1 ponto
  - CASE: 1 ponto para cada opção, exceto OTHERWISE
  - FOR: 1 ponto
  - WHILE: 1 ponto
  - REPEAT (DO WHILE): 1 ponto
  - OR ou AND na condição de qualquer das estruturas acima, acrescenta-se 1 ponto para cada OR ou AND
- Ex.:
  - 2 IFs + 1 WHILE = 3 + 1 = 4

Teste estrutural

- Exemplo
  - Determinar a complexidade ciclomática (cc) para o código abaixo:

```
01.     IF a == b THEN
02.        Comando1;
03.     ELSE
04.        Comando2;
05.     ENDIF;
```

    - Complexidade ciclomática (cc) = 1(IF) + 1 = 2

Teste estrutural

b) Complexidade ciclomática – grafo de fluxo

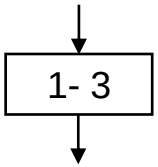
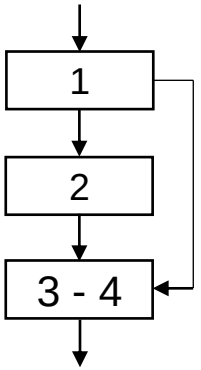
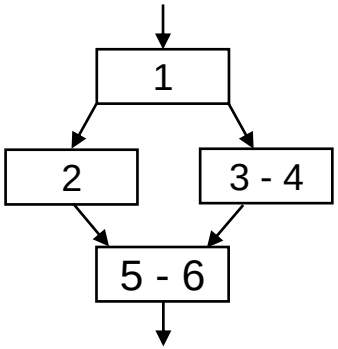
- Construa um grafo, colocando os comandos como nós e nas arestas os fluxos do programa
- Comandos em sequência podem ser colocados em um único nó
- Estruturas de controle devem ser representadas em nós distintos que indiquem a decisão e a repetição

- Para determinar a complexidade ciclomática (cc) faça:

```
cc = núm. arestas – núm. nós + 2
```

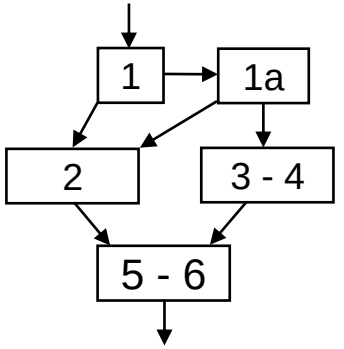
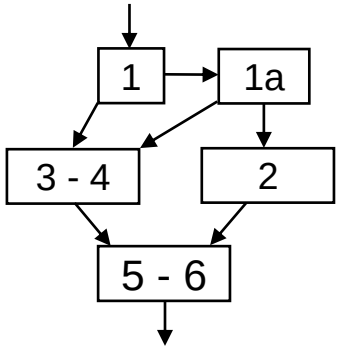
Teste estrutural

- Regras

#	Regra	Código	Grafo
1	Comandos em sequência (dois ou mais)	01. <...>; 02. <...>; 03. <...>;	
2	*If-Then*	01. IF <condição> THEN 02. <...>; 03. ENDIF; 04. <...>;	
3	*If-Then-Else*	01. IF <condição> THEN 02. <...>; 03. ELSE 04. <...>; 05. ENDIF; 06. <...>;	

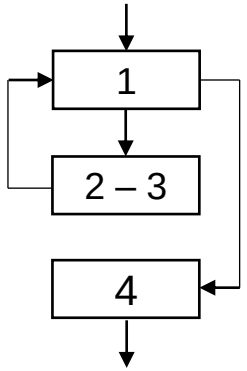
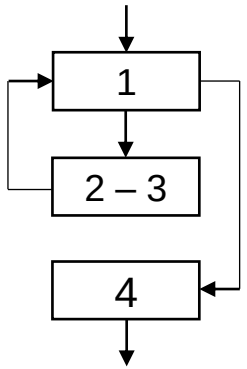
Teste estrutural

- Regras

#	Regra	Código	Grafo
4	*If-Then com condição OR*	01. IF cond1 OR cond2 THEN 02. <...>; 03. ELSE 04. <...>; 05. ENDIF; 06. <...>;	
5	*If-Then com condição AND*	01. IF cond1 AND cond2 THEN 02. <...>; 03. ELSE 04. <...>; 05. ENDIF; 06. <...>;	

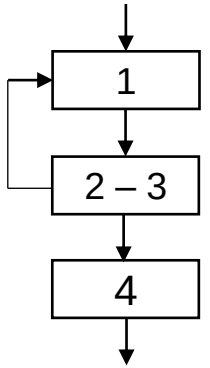
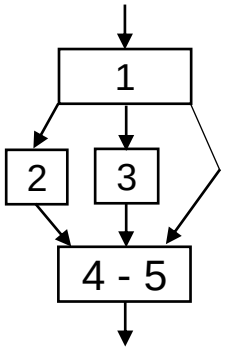
Teste estrutural

- Regras ...

#	Regra	Código	Grafo
6	*For*	01. FOR <condição> DO 02. <...>; 03. ENDFOR; 04. <...>;	
7	*While*	01. WHILE <condição> DO 02. <...>; 03. ENDWHILE; 04. <...>;	

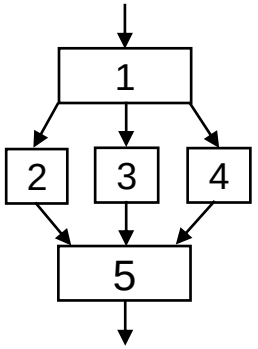
Teste estrutural

- Regras ...

#	Regra	Código	Grafo
8	*Repeat*	01. REPEAT 02. <...>; 03. UNITL <condição>; 04. <...>;	
9	*Case sem otherwise* (para qualquer quantidade de casos)	01. CASE X OF 02. a: <...>; 03. b: <...>; 04. END; 05. <...>;	

Teste estrutural

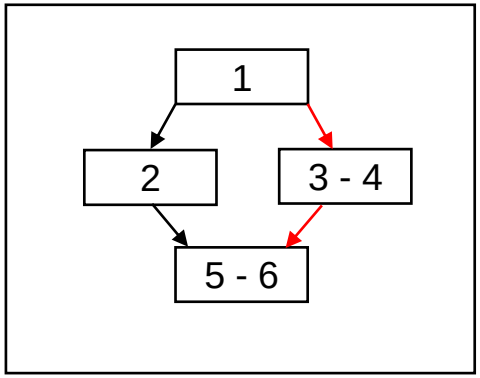
- Regras ...

#	Regra	Código	Grafo
10	*Case com otherwise* (para qualquer quantidade de casos)	01. CASE X OF 02. a: <...>; 03. b: <...>; 04. OTHERWISE <...>; 05. <...>;	

Teste estrutural

- Exemplo 1

```
01.     IF a == b THEN
02.         Comando1;
03.     ELSE
04.         Comando2;
05.     ENDIF;
06.     Comando3;
```

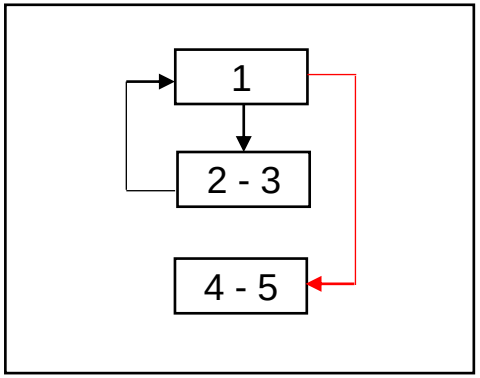


  - Complexidade ciclomática (cc)
    - Simplificada: 1 (IF) + 1 = 2
    - Grafo de fluxo: arestas (4) – nós (4) + 2 = 2

Teste estrutural

- Exemplo 2

```
01.     FOR  i=1 até n
02.                 Comando1;
03.                 Comando2;
04.     ENDFOR;
05.     Comando3;
```

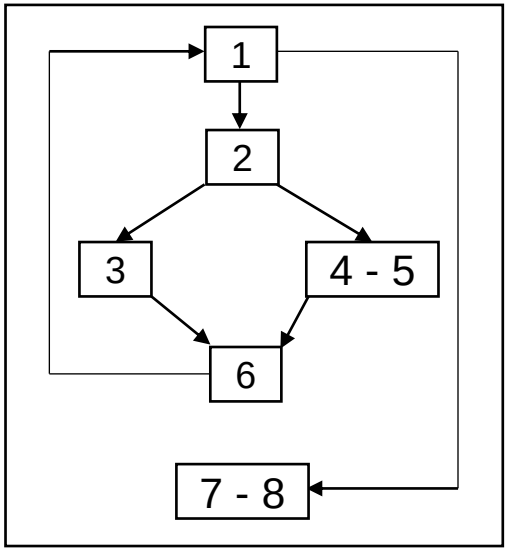


- Complexidade ciclomática (cc)
  - Simplificada: 1 (FOR) + 1 = 2
  - Grafo de fluxo: arestas (3) – nós (3) + 2 = 2

Teste estrutural

- Exemplo 3

```
01.     FOR  i=1 até n
02.             IF cond1 THEN
03.                     Comando1;
04.             ELSE
05.                     Comando2;
06.             ENDIF;
07.     ENDFOR;
08.     Comando3;
```

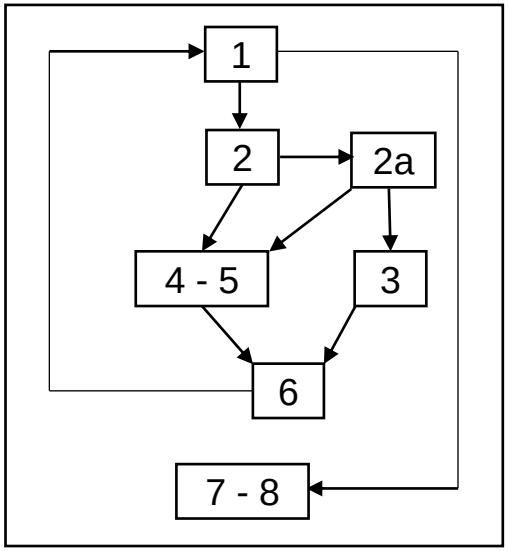


  - Complexidade ciclomática (cc)
    - Simplificada: 1 (FOR) + 1 (IF) + 1 = 3
    - Grafo de fluxo: arestas (7) – nós (6) + 2 = 3

Teste estrutural

- Exemplo 4

```
01.     FOR  i=1 até n
02.             IF cond1 AND cond2 THEN
03.                     Comando1;
04.             ELSE
05.                     Comando2;
06.             ENDIF;
07.     ENDFOR;
08.     Comando3;
```



  - Complexidade ciclomática (cc)
    - Simplificada: 1 (FOR) + 1 (IF) + (1) AND + 1 = 4
    - Grafo de fluxo: arestas (9) – nós (7) + 2 = 4

Teste estrutural

- Teste unitário, abordagem caixa-branca
  - Passos:
    1. Identificar quais comandos exercitar ou testar
       - Determinar a complexidade ciclomática
    2. Definir os caminhos independentes
       - Quais comandos serão testados
    3. Criar os casos de teste
       - Entrada e saída esperada
    4. Realizar os testes

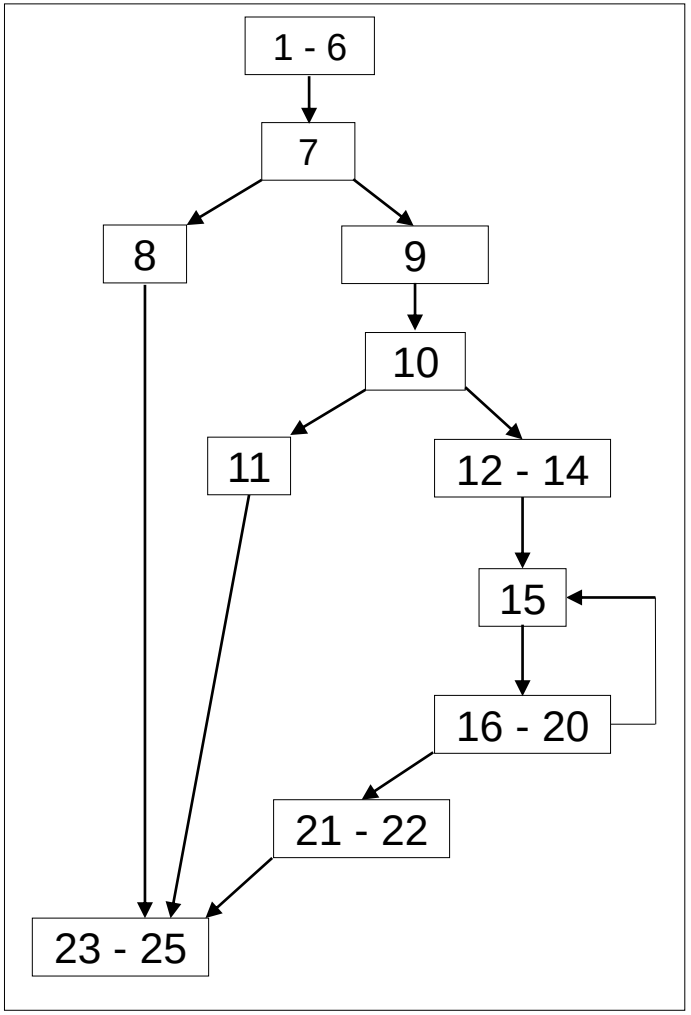
Teste estrutural

- Caminho independente
  - É definido como sendo o conjunto de comandos e condições que se executam ao menos uma vez
  - Mas como identificar ou saber quais e quantos são os caminhos independentes?
    - Analisar o valor da complexidade ciclomática
    - Grafo de fluxo

Teste estrutural

- Exemplo – complexidade ciclomática = 4

```
1  public class Fibonacci {
2
3      public static void main(String args[]) {
4          int num = 3; // informar qual termo deseja gerar
5          int fib = 0;
6
7          if (num == 0) {
8              fib = num;
9          } else {
10             if (num == 1) {
11                 fib = num;
12             } else {
13                 int ultimoFib = 1;
14                 int cont = 1;
15                 do {
16                     int penultimoFib = ultimoFib;
17                     ultimoFib = fib;
18                     fib = penultimoFib + ultimoFib;
19                     cont++;
20                 } while (cont <= num);
21             }
22         }
23         System.out.println(fib);
24     }
25 }
```



Teste estrutural

- Caminhos independentes para o exemplo:
  - c1 = <1-6, 7, 8, 23-25>
  - c2 = <1-6, 7, 9 10, 11, 23-25>
  - c3 = <1-6, 7, 9 10, 12-14, 15, 16-20, 21-22, 23-25>
  - c4 = <1-6, 7, 9 10, 12-14, 15, 16-20, 15, 16-20, 21-22, 23-25>

- Casos de teste para o exemplo:
  - Com base nos caminhos, geram-se os casos de teste

Caminho	Entrada	Saída esperada
c1	num=0	fib=0
c2	num=1	fib=1
c3	num=2	fib=1
c4	num=3	fib=2

Teste estrutural

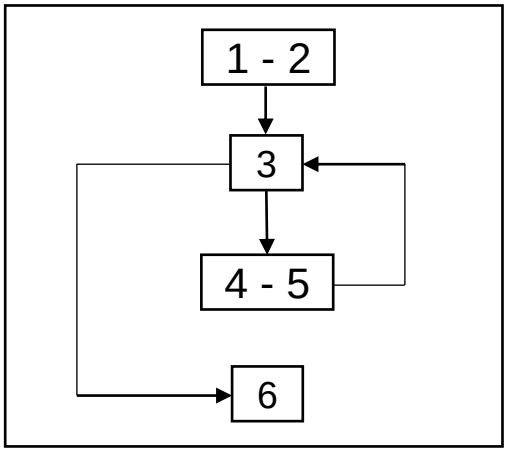
- Exercício 1
  - Utilize a técnica caixa-branca, determine o número máximo de testes para o programa abaixo e elabore um conjunto de casos de testes.

```
01.     Leia(n)
02.     fat = 1;
03.     FOR  i=1 até n
04.                 Fat = fat * i;
05.     ENDFOR;
06.     Escreva(fat);
```

Teste estrutural

- Exercício 1 - resolução

```
01.     Leia(n)
02.     fat = 1;
03.     FOR  i=1 to n
04.                 Fat = fat * i;
05.     ENDFOR;
06.     Escreva(fat);
```



- Complexidade ciclomática (cc)
  - Simplificada: 1 (FOR) + 1 = 2
  - Grafo de fluxo: arestas (4) – nós (4) + 2 = 2

Teste estrutural

- Exercício 1 - resolução
  - Caminhos independentes:

    c1 = <1-2, 3, 6>

    c2 = <1-2, 3, 4-5, 3, 6>

  - Casos de teste

Caminho	Entrada	Saída esperada
c1	n = 0	fat = 1
c2	n = 2	fat = 2

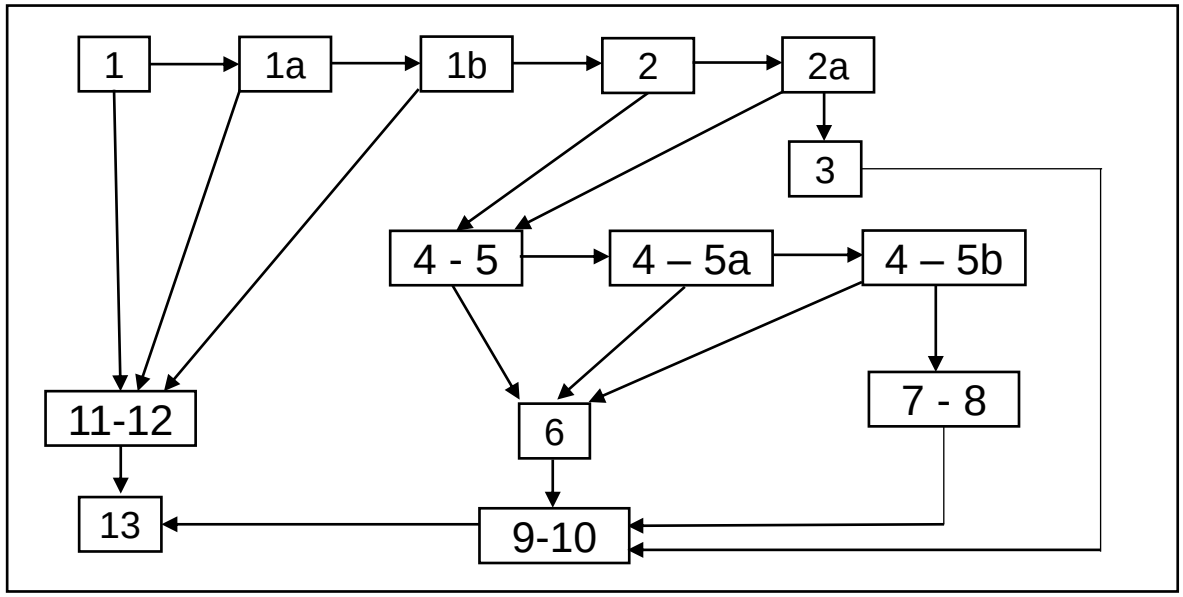
Teste estrutural

- Exercício 2
  - Utilize a técnica caixa-branca, determine o número máximo de testes e elabore um conjunto de casos de testes para o problema abaixo:
    - Dados três valores lado1, lado2 e lado3, verificar se eles podem ser os comprimentos dos lados de um triângulo e, se forem, verificar se compõem um triângulo equilátero, isósceles ou escaleno. Informar se não compuserem nenhum triângulo.

```
01.  IF lado1 <= lado2+lado3 AND lado2 <= lado1+lado3 AND lado3 <= lado1+lado2 THEN
02.    IF lado1 = lado2 AND lado2 = lado3 THEN
03.          Write “equilátero”;
04.    ELSE
05.          IF lado1 = lado2 OR lado1 = lado3 OR lado2 = lado3 THEN
06.             Write “isósceles”
07.          ELSE
08.             Write “escaleno”
09.          ENDIF
10.    ENDIF
11.  ELSE
12.     Write “não formam triângulo”
13.  ENDIF
```

Teste estrutural

- Exercício 2 - resolução



  - Complexidade ciclomática (cc)
    - Simplificada: IF(3) + AND(3) + OR(2) + 1 = 9
    - Grafo de fluxo: arestas (21) – nós (14) + 2 = 9

Teste estrutural

- Exercício 2 - resolução
  - Caminhos independentes:

c1 = <1, 11-12, 13>
c2 = <1, 1a, 11-12, 13>
c3 = <1, 1a, 1b, 11-12, 13>
c4 = <1, 1a, 1b, 2, 4-5, 6, 9-10, 13>
c5 = <1, 1a, 1b, 2, 2a, 4-5, 6, 9-10, 13>
c6 = <1, 1a, 1b, 2, 2a, 4-5, 4-5a, 6, 9-10, 13>
c7 = <1, 1a, 1b, 2, 2a, 3, 9-10, 13>
c8 = <1, 1a, 1b, 2, 4-5, 4-5a, 4-5b, 6, 9-10, 13>
c9 = <1, 1a, 1b, 2, 2a, 4-5, 4-5a, 4-5b, 7-8, 9-10, 13>

Teste estrutural

- Exercício 2 - resolução
  - Casos de teste

Caminho	Entrada	Saída esperada
c1 = <1, 11-12, 13>	10, 4, 4	“não formam triângulo”
c2 = <1, 1a, 11-12, 13>	4, 15, 3	“não formam triângulo”
c3 = <1, 1a, 1b, 11-12, 13>	4, 5, 30	“não formam triângulo”
c4 = <1, 1a, 1b, 2, 4-5, 6, 9-10, 13>	*	*
c5 = <1, 1a, 1b, 2, 2a, 4-5, 6, 9-10, 13>	7, 7, 8	“isósceles”
c6 = <1, 1a, 1b, 2, 2a, 4-5, 4-5a, 6, 9-10, 13>	10, 9, 10	“isósceles”
c7 = <1, 1a, 1b, 2, 2a, 3, 9-10, 13>	1, 1, 1	“equilátero”
c8 = <1, 1a, 1b, 2, 4-5, 4-5a, 4-5b, 6, 9-10, 13>	7, 8, 8	“isósceles”
c9 = <1, 1a, 1b, 2, 2a, 4-5, 4-5a, 4-5b, 7-8, 9-10, 13>	5, 4, 3	“escaleno”

* Caminho impossível de atingir, pois a lógica do programa não permite passar por ele.

Teste estrutural

- Síntese
  - Garante que todos comandos e condições lógicas sejam executadas ao menos uma vez
  - Adequado quando se deseja verificar a estrutura de um programa
  - Técnica utilizada para verificar sistemas, pois atua conforme o código informado (especificação dada)
  - Testar somente aquilo que foi codificado, entradas negativas para o exercício do triângulo não seria percebida pelo teste estrutural
  - Só pode ser produzido depois que o código já esteja escrito (não pode ser usada em TDD)
  - Não trata de situações com uso de polimorfismo e herança (existem técnicas específicas)

INSTITUTO FEDERAL
Santa Catarina

Curso de Ciência da Computação

Engenharia de Software

Testes caixa-preta

Prof. Alexandre Perin de Souza
alexandre.perin@ifsc.edu.br

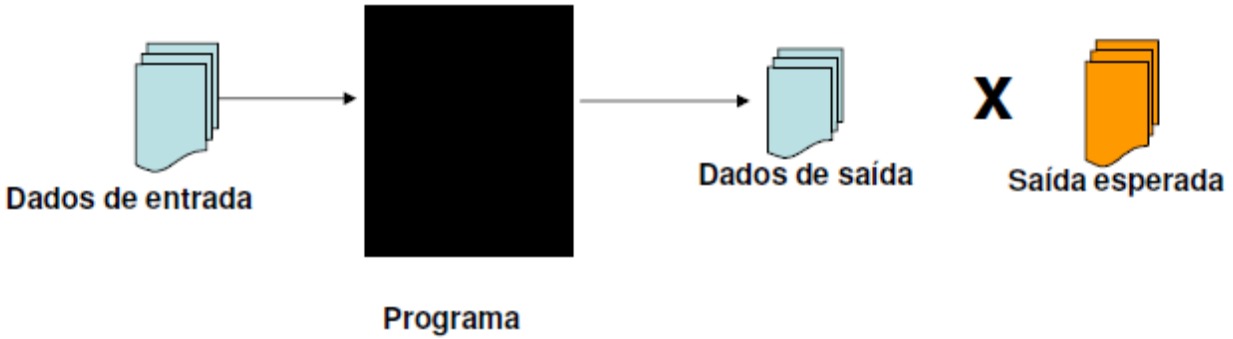
Lages (SC)

Introdução

- Sumário
  - Testes caixa-preta
  - Particionamento em classes
  - Diretrizes para criação de testes (passos)
  - Análise do valor limite
  - Exemplos
  - Exercícios

Teste funcional

- Caixa-preta
  - Em muitas situações é necessário verificar a funcionalidade de um programa, independente do seu código-fonte e da sua estrutura interna
    - Testes funcionais caixa-preta são executados sobre as entradas do programa
    - São baseados na especificação do programa, ou seja, a especificação funcional é usada para derivar os casos de teste



Teste funcional

- Particionamento em classes de equivalência
  - O grande motivo em se usar classes de equivalência reside na impossibilidade de testar todas os elementos de cada conjunto, pois muitos podem ser infinitos
    - Ex.: inteiros
  - Para contornar esta dificuldade, o domínio é dividido em partições ou classes equivalentes (subconjuntos do domínio)
  - Há 4 tipos de classes de equivalências:
    1. Intervalo
    2. Quantidade de valores, números ou dígitos
    3. Conjunto determinado (faixa de valores)
    4. “Deve ser assim ou de tal forma”

Teste funcional

- Tipos de classes de equivalência:

1. Intervalo: quando as entradas são na forma de um intervalo de valores [10..20]
   - Um conjunto válido. Ex.: 15
   - Dois conjuntos inválidos: menor que 10 e outro maior que 20
2. Quantidade de valores: quando se exige um número de valores (um, dois, três... valores para serem digitados). Ex.: CPF exige 11 dígitos ou números
   - Um conjunto válido número de valores exigidos
     - Com 11 dígitos
   - Dois inválidos
     - Um com menos elementos ou nenhum
     - Com mais de 11 dígitos

Teste funcional

- Tipos de classes de equivalência:

3. Conjunto determinado de valores. Ex.: tabela do imposto de renda
   - Um conjunto válido para cada uma das formas (faixas) e
   - Um conjunto inválido para o geral

4. Entradas como uma condição do tipo “deve ser de tal forma”. Ex.: a data da compra deve ser igual ou maior que a data de chegada ao estoque
   - Um conjunto válido (condição verdadeira) e
   - Um conjunto inválido (condição falsa)

Teste funcional

- Diretrizes ou passos para execução de testes:

1. Estudar a especificação do software (requisitos)

   1.1 Elaborar exemplos

2. Identificar classes de equivalência

   2.1 Classes válidas e inválidas

3. Definir os casos de teste

   3.1 Enumerar as classes de equivalência

   3.2 Casos de teste para as classes válidas

   3.3 Casos de teste para as classes inválidas

Teste funcional

- Exemplo 1
  - Considere a especificação abaixo, usada para elaborar testes funcionais caixa-preta. Defina classes e a quantidade de testes.

> *Seja uma cadeia de caracteres "computação" designada como X e uma outra definida como entrada Y. Se Y estiver contida em X retornar true, false caso contrário.*

- Especificação – exemplo:

a) Entrada: "a" – resposta: true

"a" está contido em "computação", portanto true

b) Entrada: "w" – resposta: false

"w" não está contido em "computação", portanto false

Teste funcional

- Exemplo 1...

Partições ou classes

Condições de entrada	Classe válida	Classe inválida
String informada está contida na string “computação”	Pertence (1)	Não pertence (2)

Casos de teste

#	Entrada	Saída esperada	Classe coberta
1	“a“	true	1
2	“compu“	true	1
3	“!“	False	2
4	“x12345678”	false	2

Teste funcional

- Exemplo 2
  - Considere a especificação abaixo, usada para elaborar testes funcionais caixa-preta. Defina classes e a quantidade de testes.

> *Uma busca por um caracter em uma cadeia de caracteres é válida, se a cadeia tiver entre 1 e 10 caracteres. Neste caso, se o caracter estiver presente retorna-se a sua posição e -1 se ele não estiver. Caso a cadeia contiver menos que 1 ou mais que 10 caracteres, independente do caracter buscado, deve-se retornar uma mensagem "entre com um tamanho entre 1 e 10".*

- Especificação - exemplo:
  a) Entrada: "abcd" – "c" resultado: posição 3
  b) Entrada: "computação" –"x" resultado: posição -1
  c) Entrada: "teste_teste" – "t"
     resultado: entre com um tamanho entre 1 e 10

Teste funcional

- Exemplo 2...

Partições ou classes

Condições de entrada	**Classe válida**	**Classe inválida**	
Tamanho (t) da string	1 <= t <= 10 (1)	t < 1 (2)	t > 10 (3)
String informada foi encontrada: sim – retornar posição – não retornar -1	Sim (4)	Não (5)	

Casos de teste

#	**Entrada**	**Saída esperada**	**Classe coberta**
1	“abcd” – “c”	3	1, 4
2	“computação” – “x”	-1	5
3	“teste_teste” – “t”	Entre com tamanho entre 1 e 10	3
4	“” – “e”	Entre com tamanho entre 1 e 10	2

Teste funcional

- Análise do valor limite
  - Participação de equivalência normalmente é usada em conjunto com o critério de análise do valor limite
  - Consiste em considerar valores fronteiriços das classes de equivalência
    - Por exemplo, se um programa exige uma entrada que, para ser válida, deve estar no intervalo [n..m], então existem três classes de equivalência:
      a) Inválida para qualquer x < n
      b) Válida para qualquer x >= n e x <= m
      c) Inválida para qualquer x > m

Teste funcional

- Análise do valor limite...
  - A técnica sugere que possíveis erros vão estar em pontos em que um intervalo se encontra com outro, então:
    a) Para a 1ª classe inválida, testa-se o valor n – 1
    b) Para a classe válida, testam-se os valores de n e m
    c) Para a 2ª inválida, testa-se o valor de m + 1

Teste funcional

- Exemplo 3
  - Considere a especificação abaixo, usada para elaborar testes funcionais caixa-preta. Defina classes e a quantidade de testes.

> *Um identificador válido deve começar com uma letra e conter apenas letras, dígitos e "_". Além disso, deve ter no mínimo um caractere e no máximo seis caracteres de comprimento*.

- Especificação - exemplo:
  a) Entrada: a_1 – resultado: válido
  b) Entrada: cont+1 – resultado: inválido
  c) Entrada: 5sub – resultado: inválido
  d) Entrada: a123456 – resultado: inválido

Teste funcional

- Exemplo 3

Partições ou classes

Condições de entrada	**Classe válida**	**Classe inválida**	
Tamanho t do identificador	1 <= t <= 6 (1)	t > 6 (2)	t < 1 (3)
Primeiro caractere é uma letra	Sim (4)	Não (5)	
Só contém caracteres válidos (alfanuméricos ou '_')	Sim (6)	Não (7)	

Casos de teste

#	**Entrada**	**Saída esperada**	**Classe coberta**
1	a_1	Válido	1, 4, 6
2	2B3	Inválido	5
3	Z-12	Inválido	7
4	A123456	Inválido	2
5	""	Inválido	3

Teste funcional

- Exercício 1
  - Considere a especificação abaixo, usada para elaborar testes funcionais caixa-preta. Defina classes e a quantidade de testes.

> *Uma nota para uma disciplina é válida se ela estiver no intervalo de [0..10] e deve ser inteira*.

- Especificação - exemplo:
  a) Entrada: 8 - resposta: válido
  b) Entrada: 5,5 – resposta: inválido
  c) Entrada: -1 – resposta: inválido
  d) Entrada: 3 – resposta: válido

Teste funcional

- Exercício 1

Partições ou classes

Condições de entrada	**Classe válida**	**Classe inválida**	
Valor da nota	0 <= nota <= 10 (1)	nota < 0 (2)	nota > 10 (3)
Valor da nota inteiro	Sim (4)	Não (5)	

Casos de teste

#	**Entrada**	**Saída esperada**	**Classe coberta**
1	8	Válida	1, 4
2	-1	Inválido	2
3	11	Inválido	3
4	7,76	Inválido	5

Teste funcional

- Exercício 2
  - Considere a especificação abaixo, usada para elaborar testes funcionais caixa-preta. Defina classes e a quantidade de testes.

> *A data de pagamento de uma compra é válida, caso ela seja maior ou igual a data de registro da compra*.

- Especificação - exemplo:
  a) Entrada: data da compra = 10/10/2019
     Data do pagamento = 11/10/2019 (válida)
  b) Entrada: data da compra = 08/11/2019
     Data do pagamento = 25/12/2018 (inválido)

Teste funcional

- Exercício 2

Partições ou classes

Condições de entrada	**Classe válida**	**Classe inválida**
Data do pagamento	data pagamento >= data compra (1)	data pagamento < data compra (2)

Casos de teste

#	**Entrada**	**Saída esperada**	**Classe coberta**
1	10/11/2019	Válida	1
2	08/0/2019	Inválido	2

Introdução

- Sumário
  - Testes funcionais caixa-preta abstraem os comandos internos, pois estão interessados nas saídas
  - Não é possível testar todas as possibilidades, por isso criam-se classes de equivalência.
    - Elas poupam tempo e esforço, pois permitem concentrar nos pontos mais susceptíveis a erros.
  - Diretrizes para criação de testes constituem um passo a passo para execução de testes funcionais caixa-preta.

INSTITUTO FEDERAL
Santa Catarina

Curso de Ciência da Computação

Engenharia de Software
Teste de software

Prof. Alexandre Perin de Souza
alexandre.perin@ifsc.edu.br

Lages (SC)

Introdução

- Motivação
  - No passado
    - Era uma tarefa complementar, com pouca importância, ocorria no final do ciclo de vida e quase sempre era postergada
    - Enfadonha, executada com pouca técnica (maioria *ad hoc*)
  - Hoje
    - Atividade importante e fundamental (qualidade e complexidade)
    - Presente e integrada a diversas etapas do ciclo de vida: ágeis
    - Profissionais e técnicas especializadas surgiram
    - Atividade se tornou mais profissional: empresas dedicadas surgiram

Introdução

- Motivação...
  - Máxima da área de testes
    - Por melhor que seja um desenvolvedor, por mais que ele use técnicas e ferramentas avançadas, é um mito pensar que ele gere software sem erros (BEIZER, 1990).

Introdução

- Fundamentos
  - Testes - definição:
    - Corresponde a um conjunto de atividades usadas para mostrar a presença de erros, mas nunca a ausência deles (DIJKSTRA, 1972).
  - Objetivo:
    - Definir conjuntos finitos e exequíveis de teste que, mesmo não garantindo que o software esteja livre de defeitos, consiga localizar os mais prováveis (WAZLAWICK, 2013).

Introdução

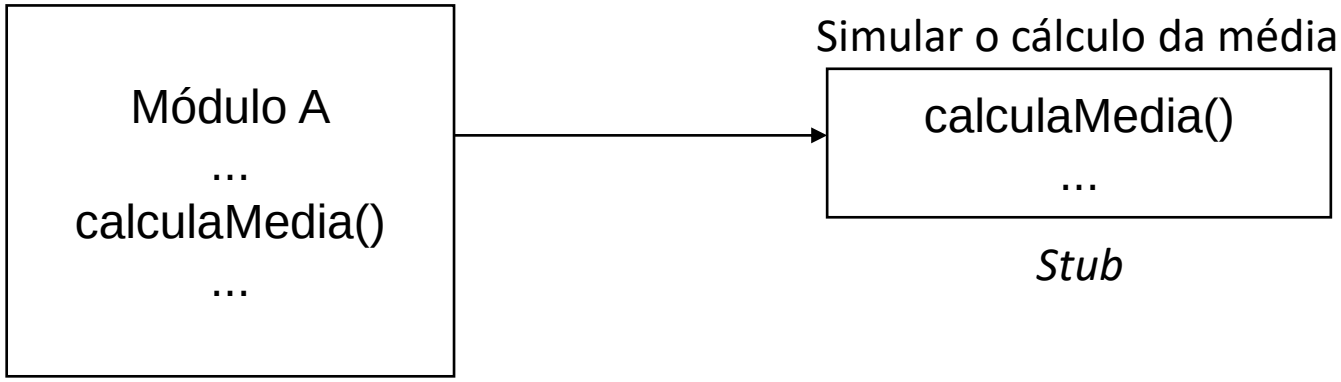
- Fundamentos
  - Termos importantes:
    - Erro (*error*) é uma diferença detectada entre o resultado gerado e esperado de uma computação
    - Defeito (*fault*) é uma linha de código que provoca um erro observado
    - Falha (*failure*) é um não funcionamento do software, possivelmente provocado por um defeito.

Introdução

- Fundamentos...
  - Termos importantes ...
    - Verificação: consiste em analisar o software para ver se ele está sendo construído de acordo com o que foi especificado
    - Validação: consiste em analisar o software para ver se ele atende às necessidades dos usuários (interessados)
    - Depuração: é a atividade que visa buscar a causa do erro, ou seja, o defeito que está causando aquele mal funcionamento. Geralmente é apoiada por ferramentas de depuração.

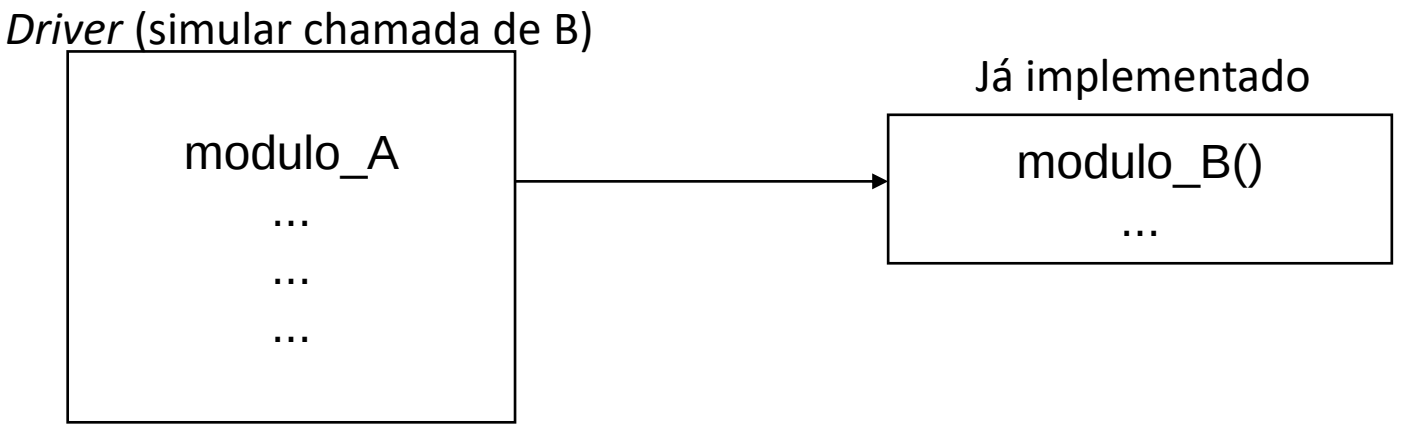
Introdução

- Fundamentos...
  - Termos importantes ...
    - *Stubs* e *drivers:* implementações simuladas ou simplificadas de componentes de sistema construídas para atividades de teste.
      - *Stubs*: quando se precisa de um componente que ainda não foi implementado, cria-se uma implementação simplificada (simulada) chamada de *stub,* que é utilizado como se fosse o componente real



Introdução

- Fundamentos...
  - Termos importantes ...
    - *Stubs* e *drivers*
      - *Drivers*: quando um módulo, que já está implementado, necessita ser acionado por outro módulo ainda não implementado. Por exemplo, o módulo A (ainda não implementado) faz chamadas a um módulo B já implementado (*driver*)



Níveis de teste

- Níveis de teste de funcionalidade:
  - Teste de unidade (unitário)
  - Teste de integração
  - Teste de sistema
  - Teste de aceitação
  - Teste de regressão

Teste de unidade

- São os mais básicos e consistem em verificar se um componente individual (unidade) foi desenvolvido corretamente.
  - Componente pode ser: método, procedimento, classe etc.
  - Costumam ser realizados pelo próprio programador
  - Há ferramentas (Ex.: JUnit - para diversas linguagens)
  - Exemplo:
    - Verificar se um método foi corretamente implementado em uma classe
    - Suponha a classe Livro e o método setIsbn(String umIsbn)
    - A especificação requer que seja trocado o valor do isbn pelo informado no parâmetro, mas, antes disso, o método deve verificar se isbn passado já existe (dois livros não podem ter o mesmo isbn)

Teste de integração

- Servem testar a integração de componentes de software já desenvolvidos.
- Estratégias:
  - *Bing-bang*: consiste em construir os diferentes componentes (classes) e depois integrar tudo junto no final
    - Vantagens x desvantagens
      - Paralelismo na atividade
      - Não precisa de *drivers* e *stubs*
      - Não é incremental (inadequado a métodos ágeis)
      - Integração de muitos componentes pode dificultar achar erros

Teste de integração

- Estratégias...
  - *Bottom-up*: integram-se os módulos de mais baixo nível, aqueles que não dependem de outros para funcionar, e depois integram-se os de mais alto nível
    - Vantagens x desvantagens
      - Não precisa de *stubs*
      - Funcionalidades de mais alto nível são testadas mais tardiamente
  - *Top-down*: integram-se os módulos de nível mais alto, deixando os mais básicos para o final.
    - Vantagens x desvantagens
      - Verificar os componentes mais importantes cedo
      - Muitos *stubs* deverão ser utilizados

Teste de integração

- Estratégias...
  - *Sanduíche*: integram-se os módulos de nível mais alto de forma *top-down* e os módulos de nível mais baixo de forma *bottom-up*.
    - Vantagens x desvantagens
      - Planejamento mais complexo
      - Reduz problemas das técnicas *top-down* e *bottom-up*

Teste de sistema

- São os testes realizados sob ponto de vista do usuário final, varrendo as funcionalidades em busca de falhas em relação aos objetivos originais.
- São executados em condições similares – de ambiente, interfaces e massas de dados – àquelas que um usuário utilizaria no seu dia a dia de manipulação do software
- Os casos de usos expandidos podem apoiar a execução destes tipos de teste

Teste de aceitação

- São realizados por um grupo restrito de usuários finais do sistema, que simulam operações de rotina do sistema de modo a verificar se seu comportamento está de acordo com o solicitado
- Ao final, o cliente poderá aprovar a versão do sistema ou solicitar modificações

Teste de aceitação

- Tipos
  - *Alfa*
    - Teste feito pelo usuário, geralmente nas instalações do desenvolvedor;
    - Desenvolvedor observa e registra falhas ou inconsistências.
  - *Beta*
    - Teste feito pelo usuário, geralmente em suas próprias instalações;
    - Sem a supervisão do desenvolvedor.

Teste de regressão

- É executado sempre que um sistema em operação sofre manutenção. O problema é que a correção de um defeito no software, ou a modificação de alguma de suas funções, pode ter gerado novos defeitos (aí diz-se que o software regrediu!)

Testes suplementares

- São tipos de teste que verificam características associadas aos requisitos de qualidade
  - *Performance*
  - Interface
  - Tolerância a falhas
  - ...
  - Segurança
  - Recuperação de falha
  - Instalação

Testes suplementares

- *Performance* (carga, estresse e resistência)
  - Carga: normalmente usa-se uma quantidade de dados ou transações para verificar se o sistema atende aos requisitos de desempenho
  - Estresse: é um caso extremo de teste de carga. Procura-se levar o sistema ao limite máximo de funcionamento esperado para verificar como ele se comporta
    - Utiliza-se uma carga de trabalho anormal para ver quais seriam os problemas encontrados, caso a carga ficasse acima do limite máximo estabelecido

Testes suplementares

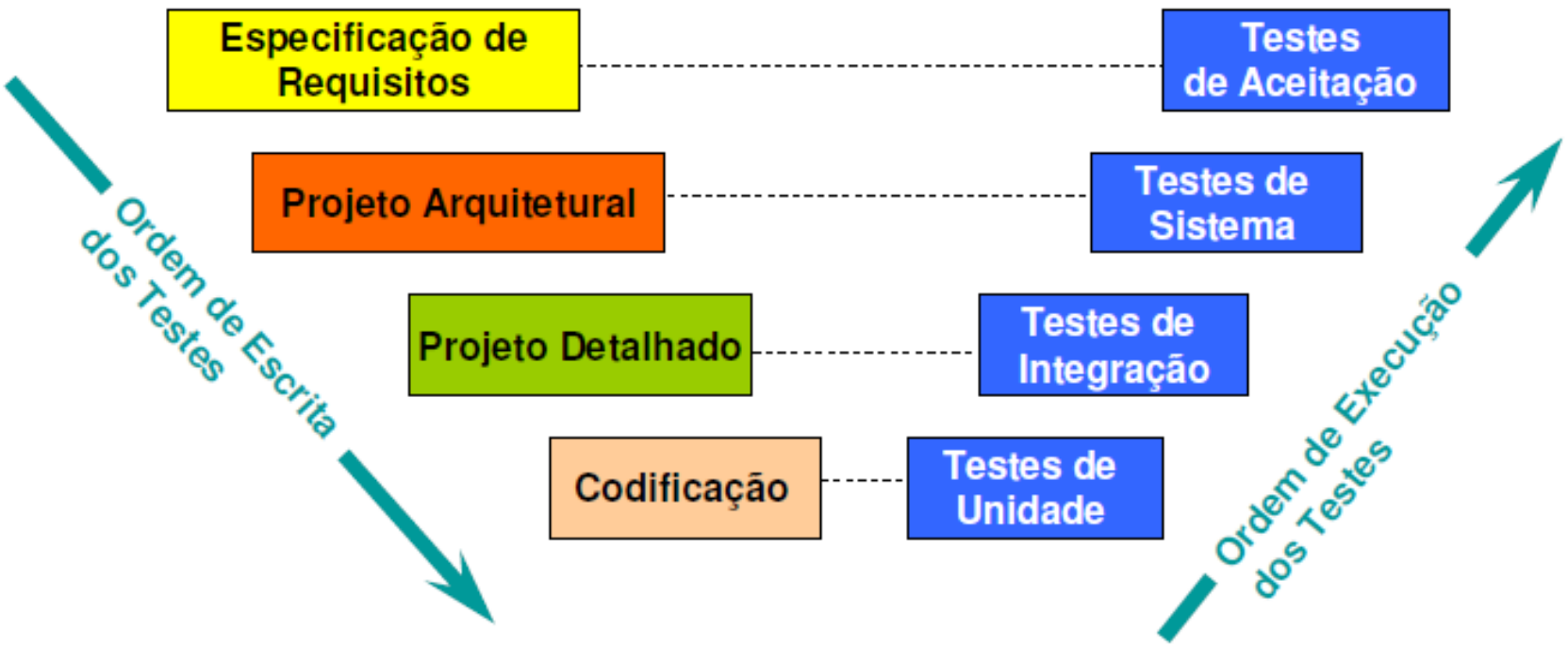
- *Performance* (carga, estresse e resistência)...
  - Resistência: é feito para verificar se um software consegue manter suas características de performance durante um longo período de tempo com uma carga de trabalho
    - Verifica-se o uso da memória ao longo do tempo para garantir que não existam perdas acumulativas de memória em função do lixo não recolhido
    - Verifica-se se não existe degradação de performance após um substancial período de tempo e que o sistema opera com carga de trabalho acima do nominal

Testes suplementares

- Instalação
  - Procura-se verificar se o software não entra em conflito com outros sistemas instalados em uma máquina
  - Neste tipo de teste procura-se também verificar se o sistema é compatível com diferentes sistemas operacionais, fabricantes de máquinas, *browsers* etc.

Estágios

- Modelo em “V” representa os estágios de testes, ou seja, em qual momento eles são realizados

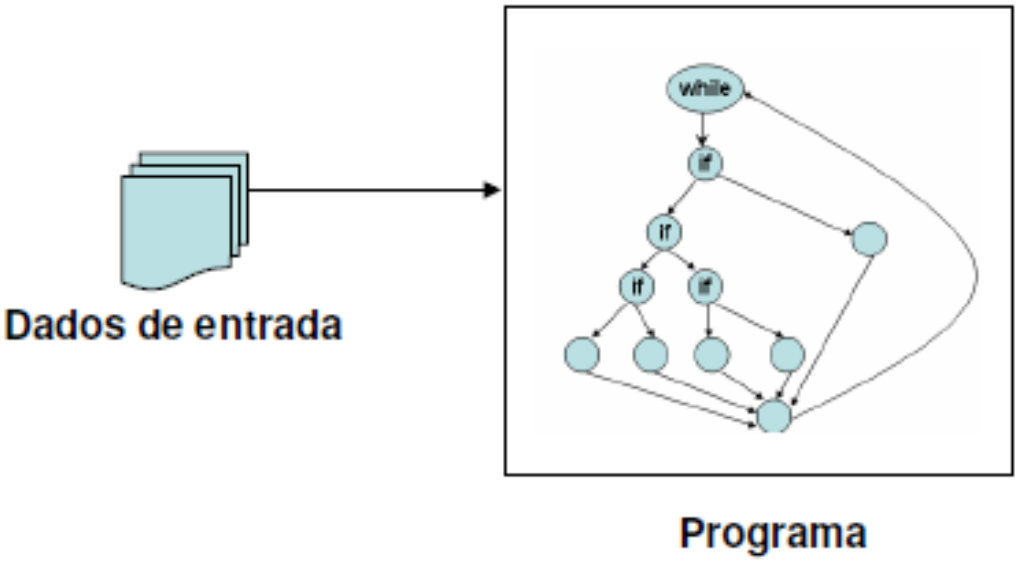


Técnicas ou abordagens

- Há duas grandes famílias:
  - Estruturais ou caixa-branca
  - Funcionais ou caixa-preta

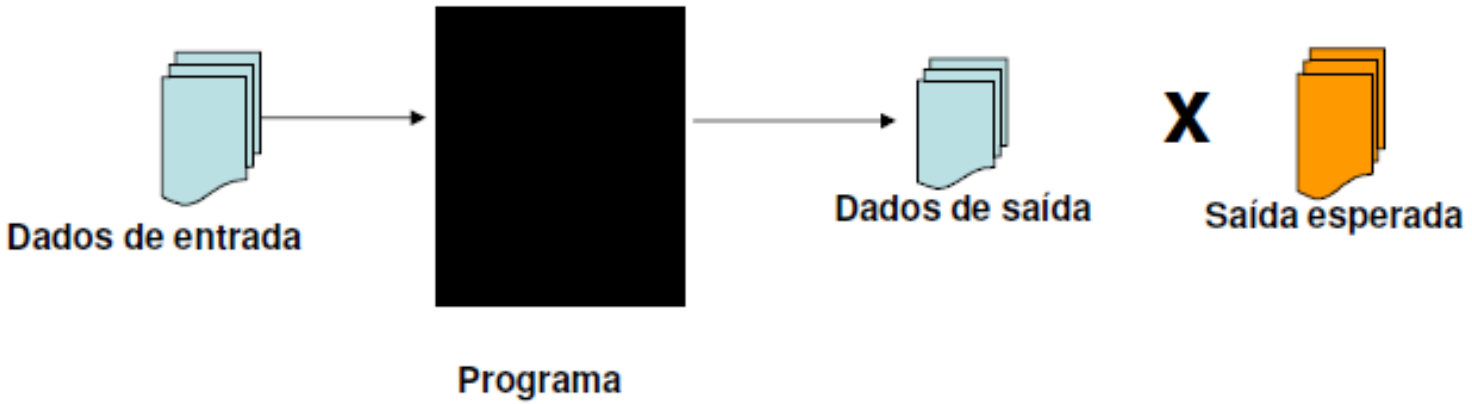
Teste estrutural

- Estruturais ou caixa-branca
  - É capaz de detectar possíveis erros pela garantia de executar todos os comandos e condições do programa ao menos uma vez.



Teste estrutural

- Funcionais ou caixa-preta
  - Testes executados sobre as entradas e saídas do programa, sem que se tenha necessariamente conhecimento do seu código-fonte.



INSTITUTO FEDERAL
Santa Catarina

Exercícios

- Use a internet e pesquisa sobre:
  1. Qual é a demanda por profissionais com conhecimento em teste de software (baixa, média ou alta)?
  2. Existem cursos voltados ao teste? É possível obter certificações?
  3. É possível e viável testar todos os comandos presentes em um programa?
  4. Os testes iniciam quando?

Introdução

- Síntese:
  - Atividade atual, especializada e importante: complexidade e qualidade
  - Apontar erros
  - Não é possível testar todos os comandos, é impraticável
  - Tipos de teste para garantir uma maior cobertura
  - Abordagens ou técnicas de teste: estrutural ou caixa preta

INSTITUTO FEDERAL
Santa Catarina

Curso de Ciência da Computação

Engenharia de Software

Testes automatizados com JUnit

alexandre.perin@ifsc.edu.br

Lages (SC).

INSTITUTO FEDERAL
Santa Catarina

Introdução

- Sumário
  - Teste manual
  - Teste automatizado
  - JUnit
  - Exemplos
  - Exercícios

Introdução

- Teste manual
  - Características principais:
    - É demorado, por isso requer tempo;
    - Dependente do ser humano, propenso a erros;
    - Exige recursos humanos especializados;
    - Tem custo elevado;
    - Útil quando é preciso realizar testes em situações específicas (cliente precisa testar para validar algo).

INSTITUTO FEDERAL
Santa Catarina

Introdução

- Teste automatizado
  - Características principais:
    - É executado por ferramentas específicas;
    - Executa uma grande quantidade de testes em pouco tempo;
    - Mais fácil de gerenciar os testes;
    - É adequado quando há alterações frequentes no código e elas exigem testes frequentes;
    - Testes de carga, *performance* e regressão.

Introdução

- Teste manual e automatizado
  - Características de ambos:
    - Exige planejamento e técnica (o que, quando e como testar);
    - Deve ser realizado sempre que possível;
      - Modelo em V (ver apêndice 2)
    - Precisam de pessoal especializado.

Testes automatizados

- *JUnit*
  - É um *framework open-source* com suporte à criação de testes automatizados na linguagem de programação Java
    - Há versões para outras linguagens (PHPUnit, PyUnit...)
  - Características:
    - Permite testar métodos e classes
    - Está integrado nas IDEs Eclipse e NetBeans
    - Baseado em anotações: @Test, @Before, @ After ...
  - A versão 4 será usada em razão do JUnit 5, a parte de testes suíte, não funcionar adequadamente com eclipse IDE
    - Versão 5, foi lançada em 2017 (https://junit.org/junit5/)
      - Material de apoio: https://junit.org/junit5/docs/current/user-guide/

INSTITUTO FEDERAL
Santa Catarina

Testes automatizados

- Exemplo 1
  - Mostrar como o *JUnit* pode ser utilizado para automatizar testes unitários, utilizando as duas abordagens de teste: caixa-preta e caixa-branca.
  - Passos:
    1. Criar um projeto Java e uma classe Fatorial
    2. Criar uma classe FatorialTeste
    3. Adicionar o JUnit ao projeto
    4. Codificar a classe FatorialTeste
        4.1 Definir a abordagem e os casos de teste
    5. Executar e analisar os resultados

INSTITUTO FEDERAL
Santa Catarina

Testes automatizados

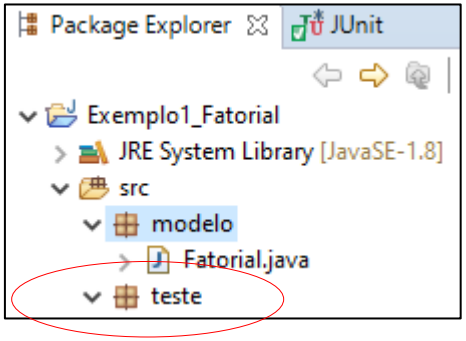
- Passo 1. Classe `Fatorial`

```
1  package modelo;
2
3  public class Fatorial {
4
5      public Fatorial() { }
6
7      public int calcula(int n) {
8
9          if ( n <=0 )
10             return 1;
11         else
12             return calcula(n-1)*n;
13
14     }
15
16 }
```

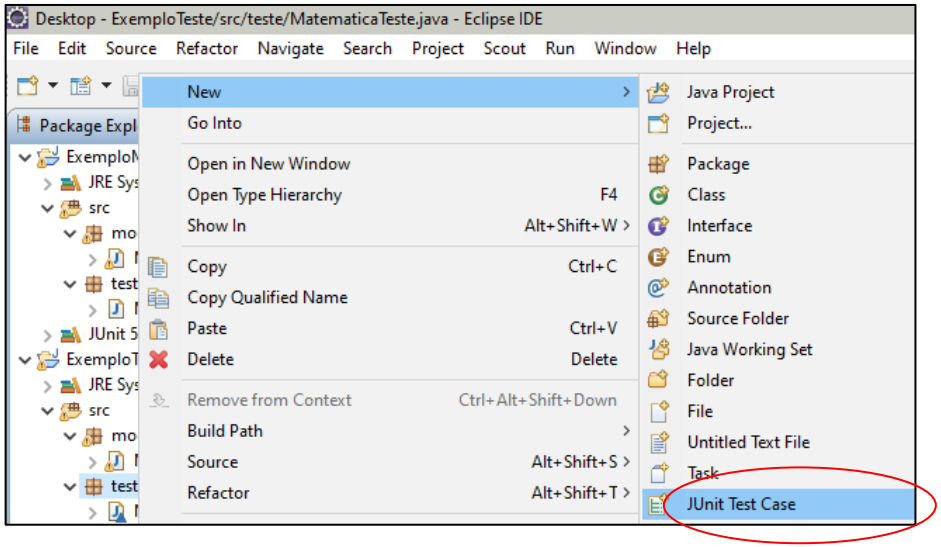
Testes automatizados

- Passo 2: no *package explorer*

2.1 Criar pacote **teste**

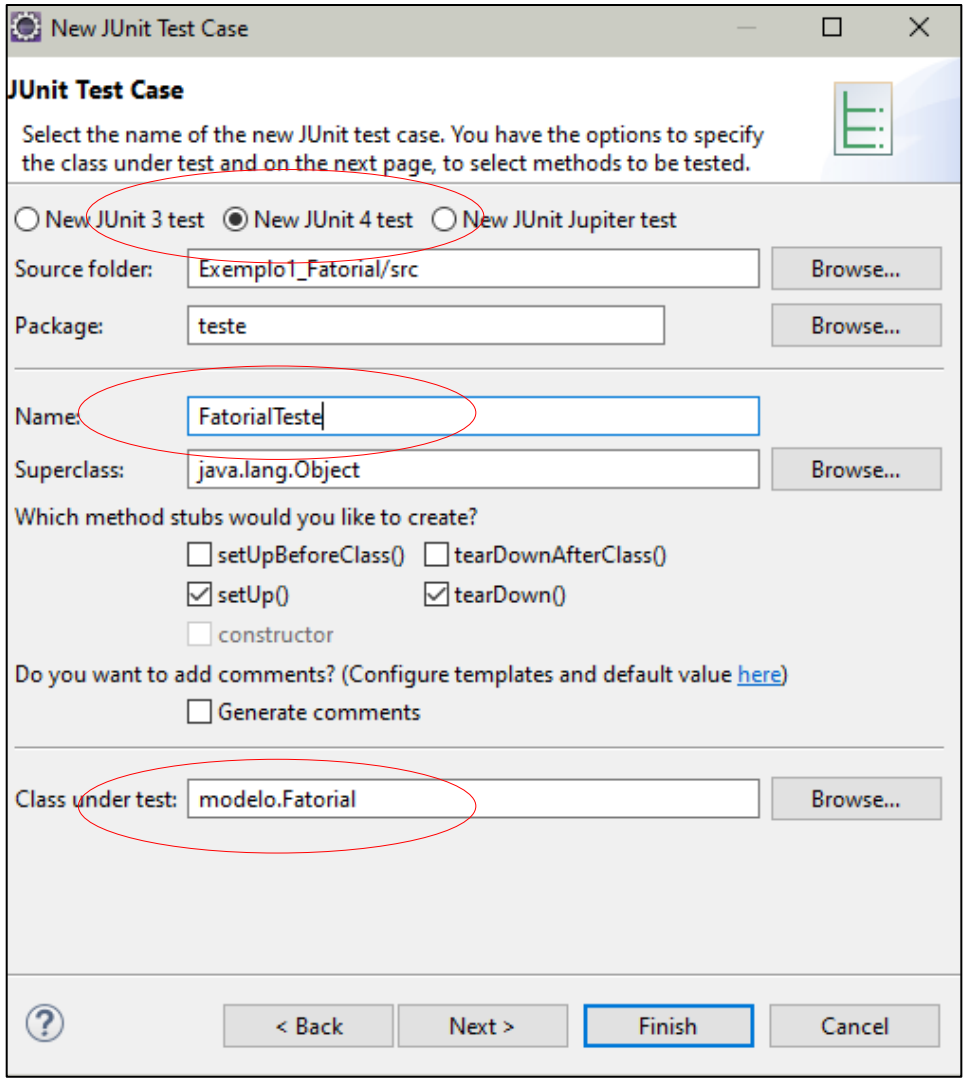


2.2 Criar uma classe e escolher como modelo **JUnit TestCase**



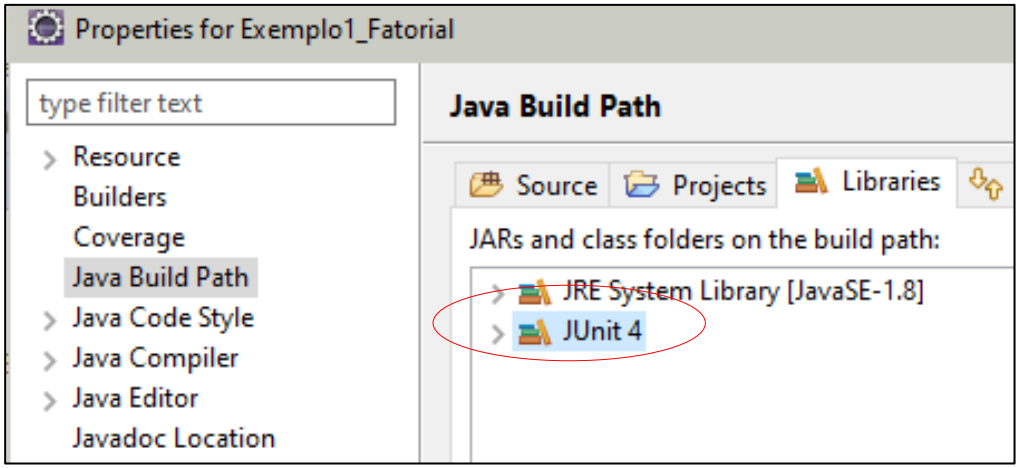
Testes automatizados

- Passo 2.2: Classe FatorialTeste

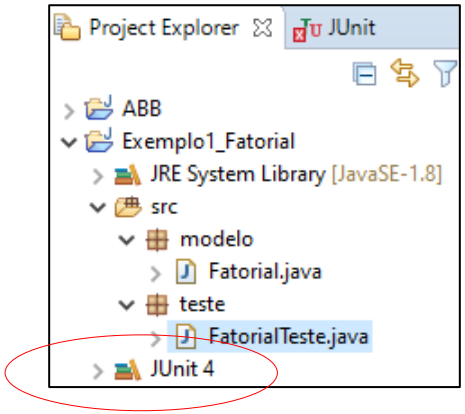


Testes automatizados

- Passo 3: Adicionar o JUnit ao projeto



3.1 Conferindo o *package explorer*



Testes automatizados

- Passo 4: Codificar a classe FatorialTeste

  4.1 Criar um método a ser testado: calculaTest. Antes do método inserir a anotação@**Test** (informa que será um método de teste)

```
1  package teste;
2
3  import static org.junit.jupiter.api.Assertions.*;
6
7  public class FatorialTeste {
8
9      @Test
10     void calculatest() {
11
12     }
13
14 }
```

Testes automatizados

- Passo 4: Codificar a classe FatorialTeste
  - 4.1 Há duas abordagens: caixa-preta e caixa-branca
  - 4.1.1 Abordagem caixa-branca

    Calcular a complexidade ciclomática (cc) = 1 (IF) + 1 = 2

    Caminho 1 (c1) = <9, 10, 13-14>
    Caminho 2 (c2) = <9, 11-12, 13-14>
    } ver slide 8, código da classe fatorial

```java
@Test
public void calculatestCaixaBranca() { // caixa-branca

    Fatorial fatorial = new Fatorial();
    // c1 = <9,10,13-14>
    int resultado = fatorial.calcula(0);
    assertEquals(1, resultado);

    // c2 = <9,11-12,13-14>
    resultado = fatorial.calcula(4);
    assertEquals(24, resultado);
}
```

INSTITUTO FEDERAL
Santa Catarina

Testes automatizados

- Passo 4: Codificar a classe FatorialTeste
  - 4.1 Há duas abordagens: caixa-preta e caixa-branca
  - 4.1.1 Abordagem caixa-preta
    - Entrada como sendo “deve ser assim”
      - Um conjunto válido e um inválido

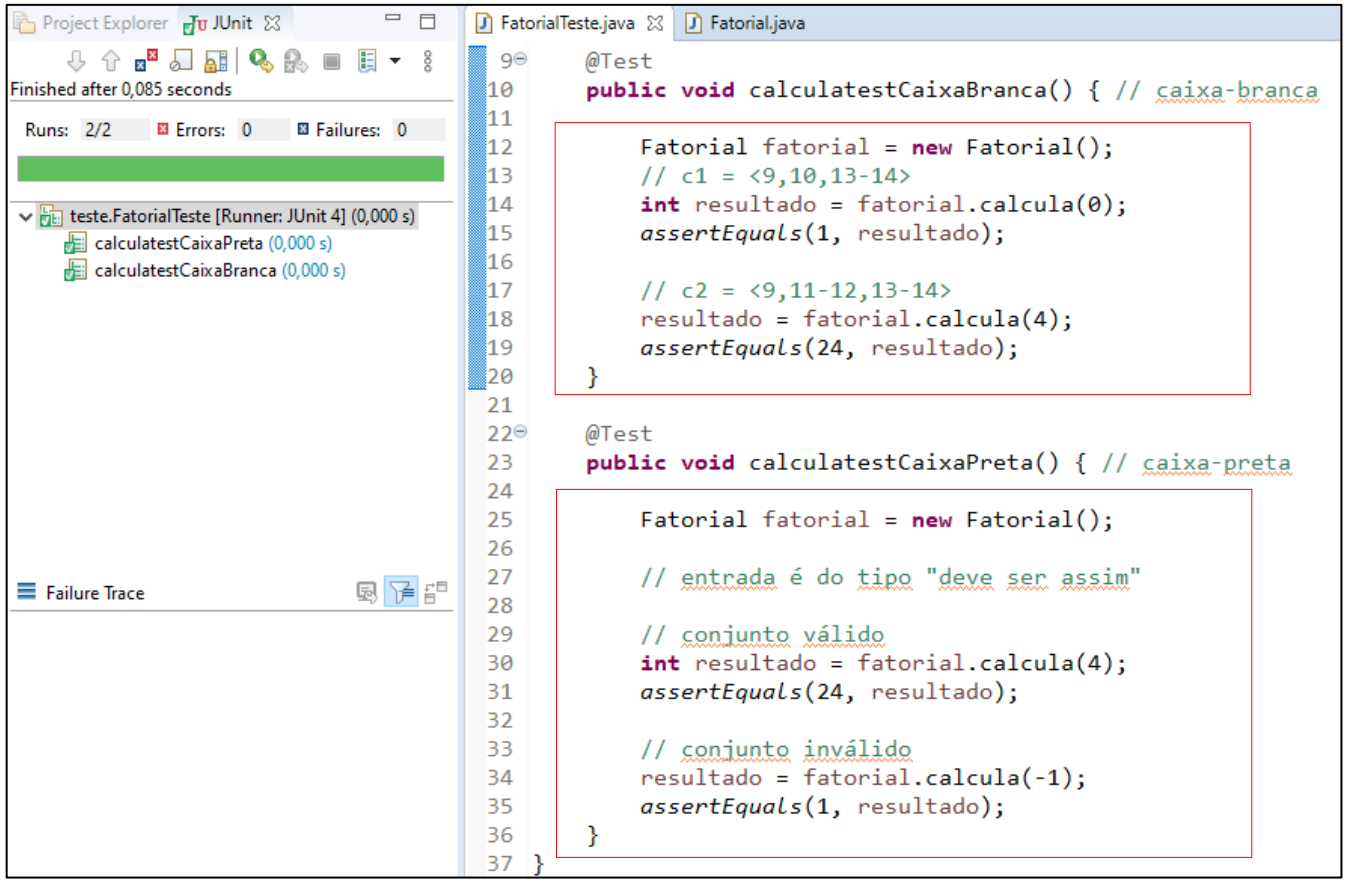
```java
@Test
public void calculatestCaixaPreta() { // caixa-preta

    Fatorial fatorial = new Fatorial();

    // entrada é do tipo "deve ser assim"

    // conjunto válido
    int resultado = fatorial.calcula(4);
    assertEquals(24, resultado);

    // conjunto inválido
    resultado = fatorial.calcula(-1);
    assertEquals(1, resultado);
}
```

Testes automatizados

- Passo 5: Executar e analisar os resultados

5.1 Usar método assertEquals(), que compara valores

```
assertEquals(valorEsperado, valorResultante);
```

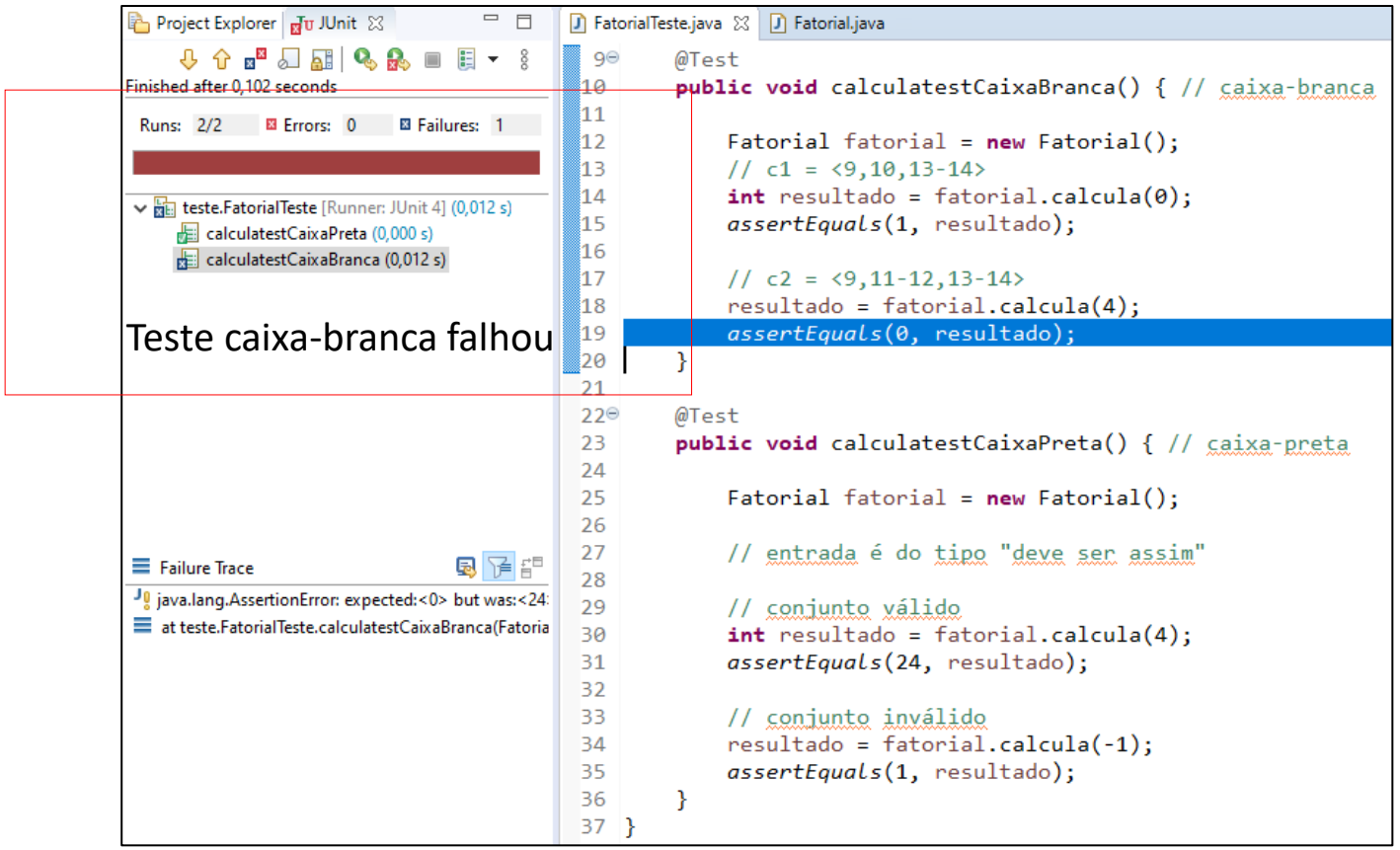


Testes automatizados

- Passo 5: Executar e analisar os resultados

5.1 Usar método assertEquals(), que compara valores

```
assertEquals(valorEsperado, valorResultante);
```



INSTITUTO FEDERAL
Santa Catarina

Testes automatizados

- Exemplo 2
  - Mostrar como o JUnit pode ser utilizado para automatizar testes unitários, utilizando a abordagem de teste caixa-preta.
  - Passos:
    1. Criar um projeto Java e uma classe Matematica
    2. Criar uma classe MatematicaTeste
    3. Adicionar o JUnit ao projeto
    4. Codificar a classe MatematicaTeste
       4.1 Definir a abordagem e os casos de teste
    5. Executar e analisar os resultados

INSTITUTO FEDERAL
Santa Catarina

Testes automatizados

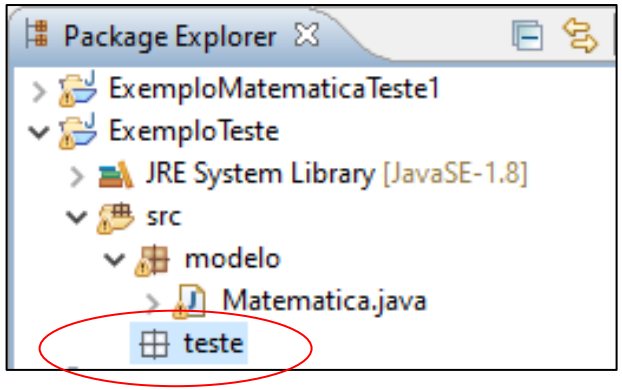
- Passo 1. Classe `Matematica`

```java
package modelo;

public class Matematica {

    public Matematica() {}

    public float soma(float operando1, float operando2) {
        return operando1 + operando2;
    }

    public float subtrai(float operando1, float operando2) {
        return operando1 - operando2;
    }

    public float multiplica(float operando1, float operando2) {
        return operando1 * operando2;
    }

    public float divide(float operando1, float operando2) {
        if (operando2 == 0) {
            return 0;
        }
        return operando1 / operando2;
    }

}
```

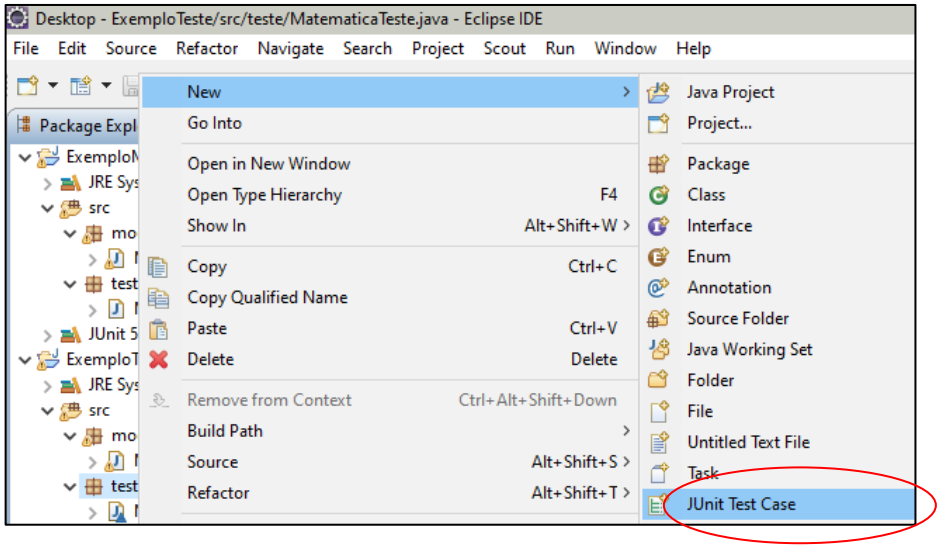
Testes automatizados

- Passo 2: no *package explorer*

2.1 Criar pacote **teste**

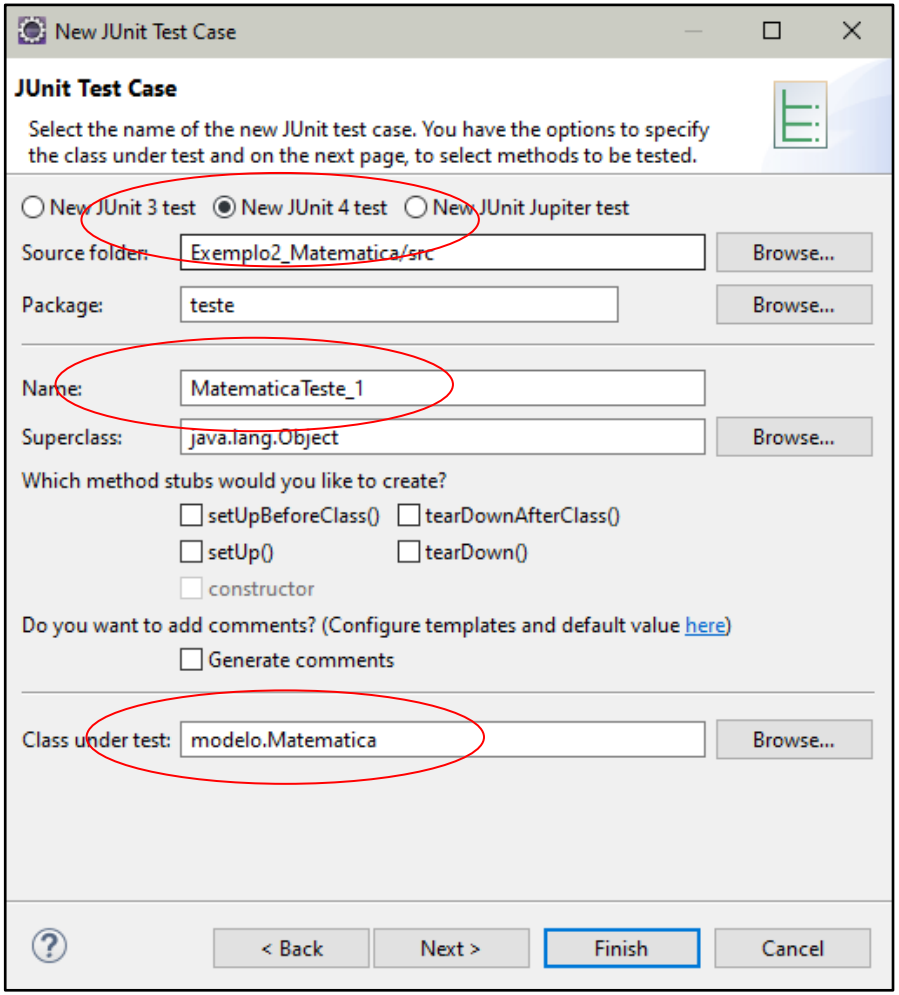


2.2 Criar uma classe e escolher como modelo **JUnit TestCase**



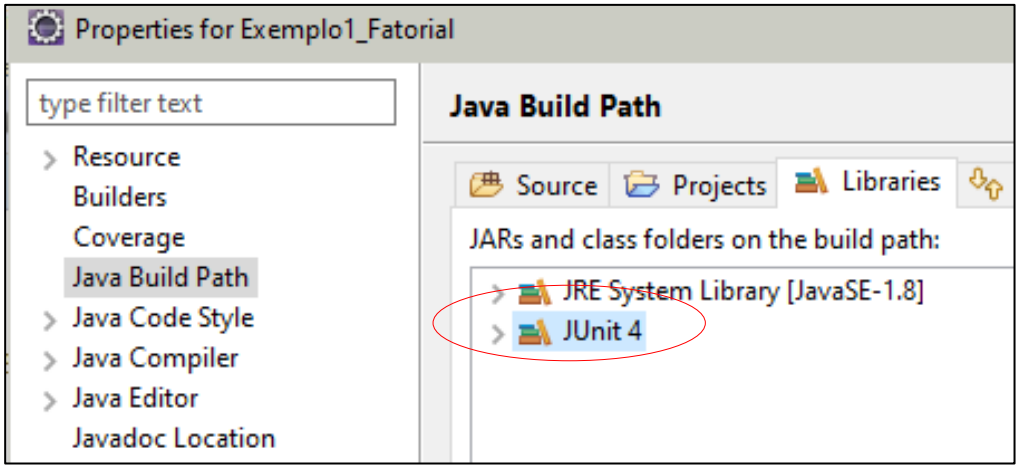
Testes automatizados

- Passo 2.2: Classe MatematicaTeste_1

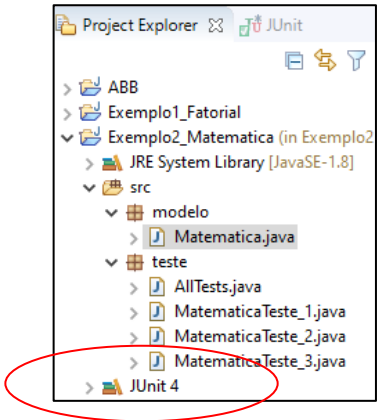


Testes automatizados

- Passo 3: Adicionar o JUnit 4 ao projeto



3.1 Conferindo o *package explorer*



Testes automatizados

- Passo 4: Codificar a classe MatematicaTeste

4.1 Há duas abordagens: caixa-preta e caixa-branca
4.1.1 Abordagem caixa-preta

```
1  package teste;
2
3  import static org.junit.Assert.*;
6
7  public class MatematicaTeste_1 {
8
9      @Test
10     public void somaTest() {
11         Matematica mtm = new Matematica();
12
13         // casos de teste
14         assertEquals(2, mtm.soma(1, 1),0);
15         assertEquals(-10, mtm.soma(-11, 1),0);
16         assertEquals(1.42, mtm.soma(3.14f, -1.72f), 0.001);
17     }
18
19 }
```

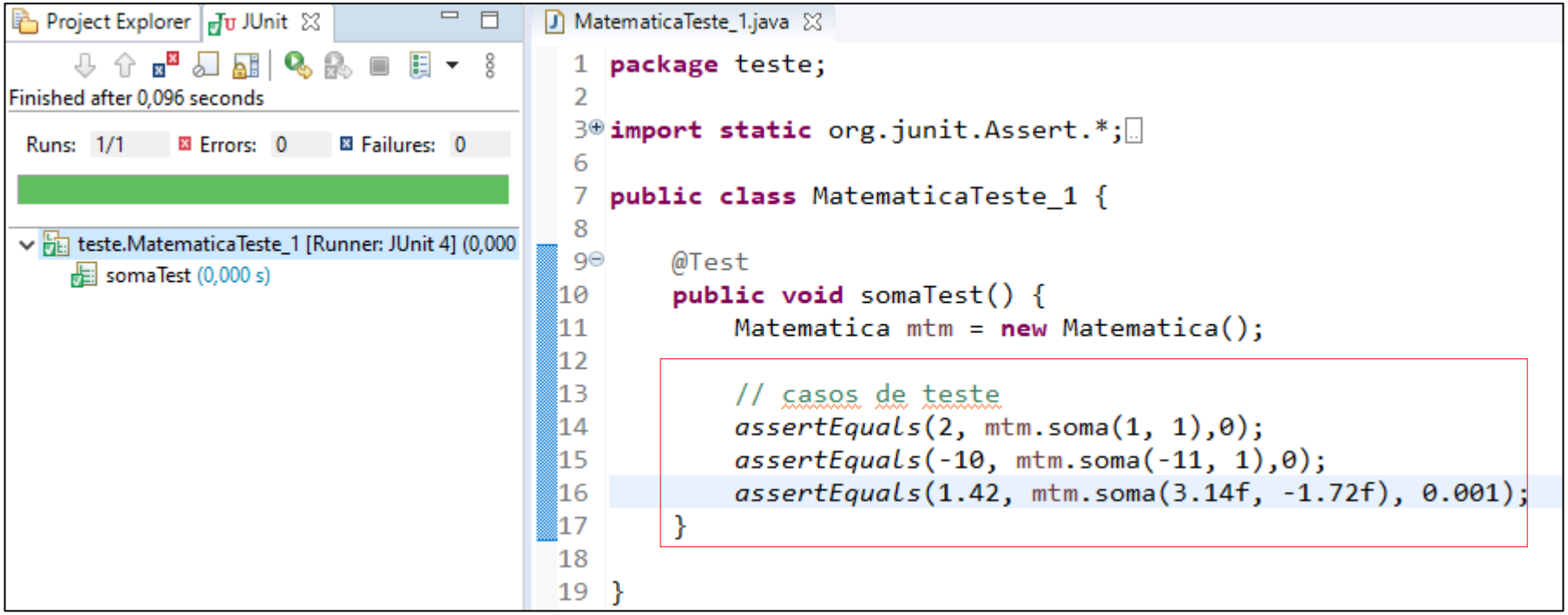
Obs.: em razão da variável ser float, há necessidade de colocar um terceiro parâmetro no método assertEquals. Este parâmetro informa o erro máximo permitido.

Testes automatizados

- Passo 5: Executar e analisar os resultados

5.1 Usar método assertEquals(), que compara valores

```
assertEquals(valorEsperado, valorResultante);
```



Testes automatizados

- Exercício

1. Implementar os demais métodos para a classe MatematicaTeste_1 e para cada um deles, criar vários casos de teste (utilize a abordagem caixa-preta):
   - subtrai
   - multiplica
   - divide

Testes automatizados

- Exercício

2. Considere a especificação abaixo, usada para elaborar testes funcionais caixa-preta. Defina classes (válida/inválida), a quantidade de testes e desenvolva um teste automatizado.

> *Seja uma cadeia de caracteres "computação" designada como X e uma outra definida como entrada Y. Se Y estiver contida em X retornar true, false caso contrário.*

Testes automatizados

- Exercício

3. Considere a especificação abaixo, usada para elaborar testes funcionais caixa-preta. Defina classes (válida/inválida), a quantidade de testes e desenvolva um teste automatizado.

> *Um identificador válido deve começar com uma letra e conter apenas letras, dígitos e "_". Além disso, deve ter no mínimo um caractere e no máximo seis caracteres de comprimento*.

Testes automatizados

- Síntese:
  - Testes precisam ser planejados e executados por pessoal especializado
  - Testes manuais, em situações específicas
  - Para obter os ganhos dos testes automatizados, é necessário ter ferramentas adequadas.

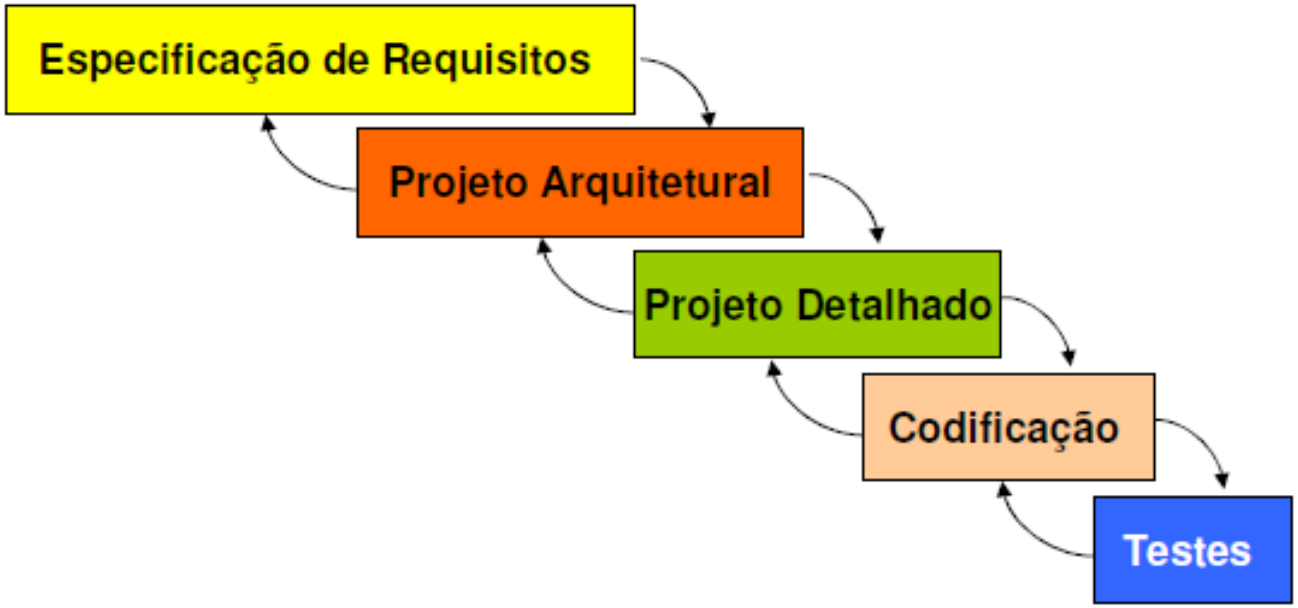
INSTITUTO FEDERAL
Santa Catarina

Testes automatizados

- Apêndice 1 – JUnit versão 5
  - Métodos assert:
    - assertTrue() - verifica se o valor é true
    - assertFalse() – verifica se o valor é false
    - assertNotEquals() - verifica se o valor é não é false
    - assertNotNull() – verifica se o valor é não nulo
    - assertArrayEquals() – verifica se 2 *arrays* contém os mesmos elementos
    - assertNotSame() – verifica se 2 referências não são para o mesmo objeto
    - assertSame() - verifica se 2 referências são para o mesmo objeto
    - asserThat() – verifica se dois valores são iguais

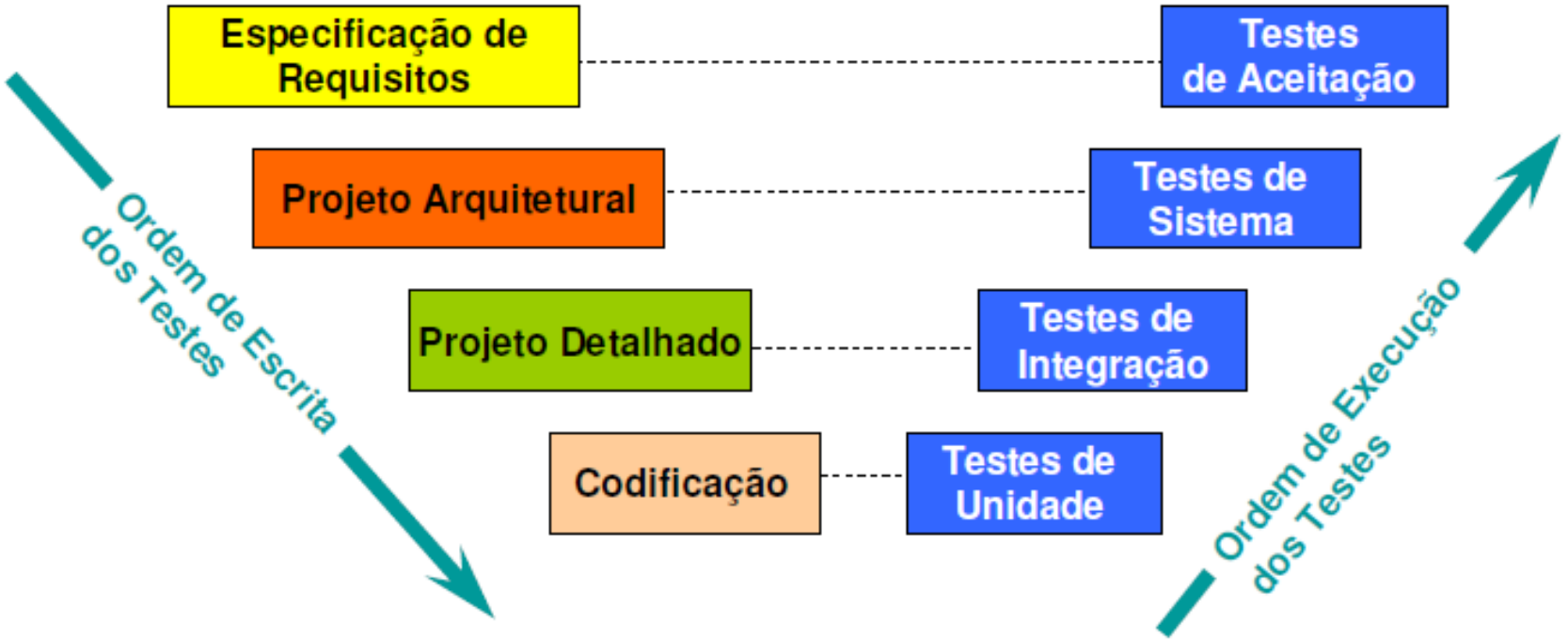
Testes automatizados

- Apêndice 2a
  - Até pouco tempo atrás... os testes eram vistos como uma etapa posterior à implementação (*Big-Bang Testing*)



Testes automatizados

- Apêndice 2b
  - Modelo em “V” representa os estágios de testes, ou seja, em qual momento eles são realizados



INSTITUTO FEDERAL
Santa Catarina

Curso de Ciência da Computação

Engenharia de Software

Testes automatizados com JUnit

alexandre.perin@ifsc.edu.br

Lages (SC).

INSTITUTO FEDERAL
Santa Catarina

Introdução

- Sumário
  - Teste automatizado
    - Independência dos testes
  - Teste *suite*
    - Vários testes

INSTITUTO FEDERAL
Santa Catarina

Introdução

- Testes automatizados – independência de testes
  - Pode ser que na montagem de alguns testes sujam cenários parecidos (mesmo código)
    - Ex.: criar objetos, atribuir valores e inserir objetos em uma lista
  - Por outro lado, para não prejudicar a execução de outros testes, pode ser necessário que, após a execução de um dado teste, seja necessário que objetos e variáveis voltem ao seu estado antes de um teste
    - Ex.: limpar uma lista, fechar uma conexão ou zerar alguma variável

INSTITUTO FEDERAL
Santa Catarina

Introdução

- Testes automatizados – independência de testes
  - É atingida através dos métodos disponíveis no JUnit:
    - *setUp*() – variáveis e objetos que devem ser inicializados
    - *tearDown*() – variáveis e objetos que devem voltar ao estado inicial.
  - A independência de testes traz:
    - Maior produtividade (mesmo código é usado para outros testes)
    - Maior facilidade para gerenciar os testes (somente em métodos específicos é que ocorrem modificações)
    - Maior clareza e organização dos testes

Testes automatizados

- Exemplo 1
  - Considere que seja necessário testar duas classes para um sistema qualquer, conforme descrições a seguir:

    - Classe: Cliente
      - Nome, dataNascimento, CPF e situação
      - Sets/Gets e toString()

    - Classe: GerenciaCliente
      - Lista (ArrayList) utilizada como *stub* para banco de dados
      - limpaLista – método auxiliar para limpar lista

      - insereCliente(Cliente cliente)
      - removeCliente (String cpf)
      - pesquisaCliente(String cpf)

    - Será usada a abordagem de teste será caixa-preta e testes unitários.

Testes automatizados

- Passos

1. Definir e codificar as classes Cliente e GerenciaCliente
2. Na classe GerenciaCliente

   2.1 Criar o método insereCliente(Cliente cliente)

   2.2 Criar o método pesquisaCliente(String cpf), retornando cliente
3. Criar a classe: GerenciaClienteTeste

   3.1 Criar o método testPesquisaCliente()
   - Montagem do cenário de testes
   - Execução do teste
   - Verificação do teste

   3.2 Criar o método testRemoveCliente()
   - Montagem do cenário de testes
   - Execução do teste
   - Verificação do teste

Testes automatizados

- Passo 1: Criação das classes: Cliente e GerenciaCliente

Cliente

```java
package modelo;

import java.time.LocalDate;

public class Cliente {
    private String nome;
    private String cpf;
    private LocalDate dataNascimento;
    private String situacao;


    public Cliente(String nome, String cpf, LocalDate dataNascimento) {
        this.nome = nome;
        this.cpf = cpf;
        this.dataNascimento = dataNascimento;
    }

```

GerenciaCliente

```java
package modelo;

import java.util.ArrayList;

public class GerenciaCliente {

    private List<Cliente> listaClientes;

    public GerenciaCliente() {

    public void insere(Cliente cliente) {

    public boolean remove(String cpf) {

    public Cliente pesquisa(String cpf) {

    public List<Cliente> getListaClientes() {

    public void setListaClientes(List<Cliente> listaClientes) {

    public void limpaLista() {

    public void mostra() {

}
```

Testes automatizados

- Passo 2: GerenciaCliente: insereCliente(Cliente cliente)

```
1  package modelo;
2
3  import java.util.ArrayList;
5
6  public class GerenciaCliente {
7
8      private List<Cliente> listaClientes;
9
10     public GerenciaCliente() {
13
14     public void insere(Cliente cliente) {
15         this.listaClientes.add(cliente);
16     }
17
18     public boolean remove(String cpf) {
37
38     public Cliente pesquisa(String cpf) {
46
47     public List<Cliente> getListaClientes() {
50
51     public void setListaClientes(List<Cliente> listaClientes) {
54
55     public void limpaLista() {
58
59     public void mostra() {
64
65 }
```

Testes automatizados

- Passo 2: GerenciaCliente: pesquisaCliente(String cpf)

```
1  package modelo;
2
3  import java.util.ArrayList;
5
6  public class GerenciaCliente {
7
8      private List<Cliente> listaClientes;
9
10     public GerenciaCliente() {
13
14     public void insere(Cliente cliente) {
17
18     public boolean remove(String cpf) {
37
38     public Cliente pesquisa(String cpf) {
39         for (Cliente cliente : this.listaClientes) {
40             if ( cliente.getCpf().compareTo(cpf) == 0 ) {
41                 return cliente;
42             }
43         }
44         return null;
45     }
```

Testes automatizados

- Passo 2: GerenciaCliente: removeCliente(String cpf)

```
public boolean remove(String cpf) {
    int index = -1;
    boolean encontrou = false;

    for (Cliente cliente : this.listaClientes) {
        index++;
        if ( cliente.getCpf().compareTo(cpf) == 0 ) {
            encontrou = true;
            break;
        }
    }

    if ( encontrou ) {
        this.listaClientes.remove(index);
        return true;
    } else {
        return false;
    }
}
```

Testes automatizados

- Passo 3: GerenciaClienteTeste

  3.1 Criar um método: testPesquisaCliente()

```
1  package teste;
2  import static org.junit.Assert.assertEquals;
3  import static org.junit.Assert.assertNull;
4  import static org.junit.Assert.assertTrue;
5  import java.time.LocalDate;
6  import org.junit.Test;
7  import modelo.*;
8
9  public class GerenciaClienteTeste {
10
11     @Test
12     public void testPesquisaCliente() {
13         // Montagem do cenário
14         GerenciaCliente gerenciaCliente = new GerenciaCliente();
15
16         Cliente cliente = new Cliente("Cliente1", "12341", LocalDate.of(1970, 1, 11));
17         gerenciaCliente.insere(cliente);
18
19         cliente = new Cliente("Cliente2", "12342", LocalDate.of(1970, 2, 12));
20         gerenciaCliente.insere(cliente);
21
22         cliente = new Cliente("Cliente3", "12343", LocalDate.of(1970, 3, 13));
23         gerenciaCliente.insere(cliente);
24
25         // Execução
26         Cliente clienteRetornado = gerenciaCliente.pesquisa("12342");
27
28         // Verificação
29         assertEquals("12342", clienteRetornado.getCpf());
30     }
```

Testes automatizados

- Passo 3: GerenciaClienteTeste

  3.2 Criar um método: testRemoveCliente()

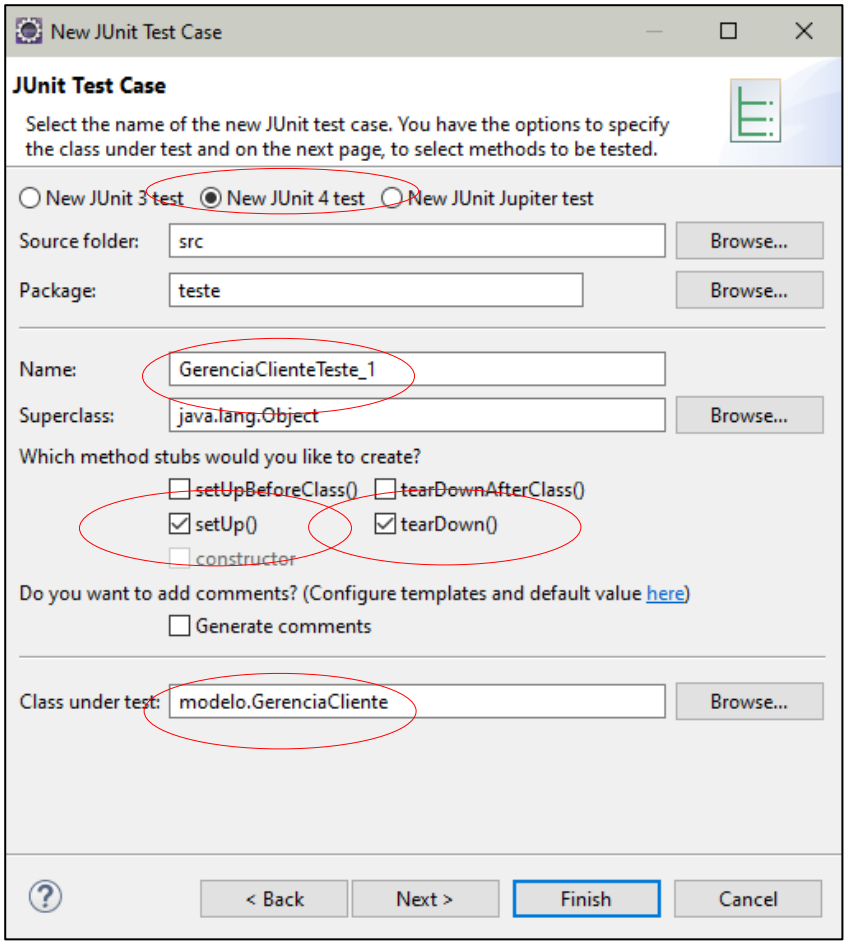
```
34    @Test
35    public void testRemoveCliente() {
36
37        // Montagem do Cenário
38        GerenciaCliente gerenciaCliente = new GerenciaCliente();
39
40        Cliente cliente = new Cliente("Cliente1", "12341", LocalDate.of(1970, 1, 11));
41        gerenciaCliente.insere(cliente);
42
43        cliente = new Cliente("Cliente2", "12342", LocalDate.of(1970, 2, 12));
44        gerenciaCliente.insere(cliente);
45
46        cliente = new Cliente("Cliente3", "12343", LocalDate.of(1970, 3, 13));
47        gerenciaCliente.insere(cliente);
48
49        // Execução
50        boolean clienteRemovido = gerenciaCliente.remove("12342");
51
52        // Verificação
53        assertTrue(clienteRemovido);
54        assertNull(gerenciaCliente.pesquisa("12342"));
55
56    }
57
```

Testes automatizados

- Na classe GerenciaClienteTeste
  - Observar que, nos métodos testPesquisaCliente() e testRemoveCliente(), a parte da montagem do cenário é igual (em ambos os métodos os códigos são iguais)
  - Método *setUp*()
    - Os códigos replicados referentes à montagem dos testes ficam em um método responsável por montar o cenário para cada teste.
    - Todo código que estiver no *setUp*() será executado antes de um teste
  - Método *tearDown*()
    - Variáveis e estruturas são “zeradas” ou “limpas” neste método
    - Por exemplo, pode-se limpar uma lista ou uma base dados, evitando que valores antigos “lixos” atrapalhem a execução de novos testes.

Testes automatizados

- Passo 3: Cenários parecidos
  - Criar a classe: GerenciaClienteTeste_1



Testes automatizados

- Passo 3: Cenários parecidos
  - Criar a classe: GerenciaClienteTeste_1

```java
package teste;

import static org.junit.Assert.*;
import java.time.LocalDate;
import org.junit.After;
import org.junit.Before;
import org.junit.Test;

import modelo.Cliente;
import modelo.GerenciaCliente;

public class GerenciaClienteTeste_1 {

    private GerenciaCliente gerenciaTeste;

    @Before
    public void setUp() throws Exception {
        GerenciaCliente gerenciaCliente = new GerenciaCliente();

        Cliente cliente = new Cliente("Cliente1", "12341", LocalDate.of(1970, 1, 11));
        gerenciaCliente.insere(cliente);

        cliente = new Cliente("Cliente2", "12342", LocalDate.of(1970, 2, 12));
        gerenciaCliente.insere(cliente);

        cliente = new Cliente("Cliente3", "12343", LocalDate.of(1970, 3, 13));
        gerenciaCliente.insere(cliente);

        this.gerenciaTeste = gerenciaCliente;
    }

    @After
    public void tearDown() throws Exception {
        this.gerenciaTeste.limpaLista();
    }
```

setUp() contém o código necessário para permitir que testes se realizem

tearDown() contém código para que os objetos possam voltar ao estado inicial (antes do teste).

Testes automatizados

- Passo 3: Execução

```java
@Before
public void setUp() throws Exception {
    GerenciaCliente gerenciaCliente = new GerenciaCliente();

    Cliente cliente = new Cliente("Cliente1", "12341", LocalDate.of(1970, 1, 11));
    gerenciaCliente.insere(cliente);

    cliente = new Cliente("Cliente2", "12342", LocalDate.of(1970, 2, 12));
    gerenciaCliente.insere(cliente);

    cliente = new Cliente("Cliente3", "12343", LocalDate.of(1970, 3, 13));
    gerenciaCliente.insere(cliente);

    this.gerenciaTeste = gerenciaCliente;
}

@After
public void tearDown() throws Exception {
    this.gerenciaTeste.limpaLista();
}
```

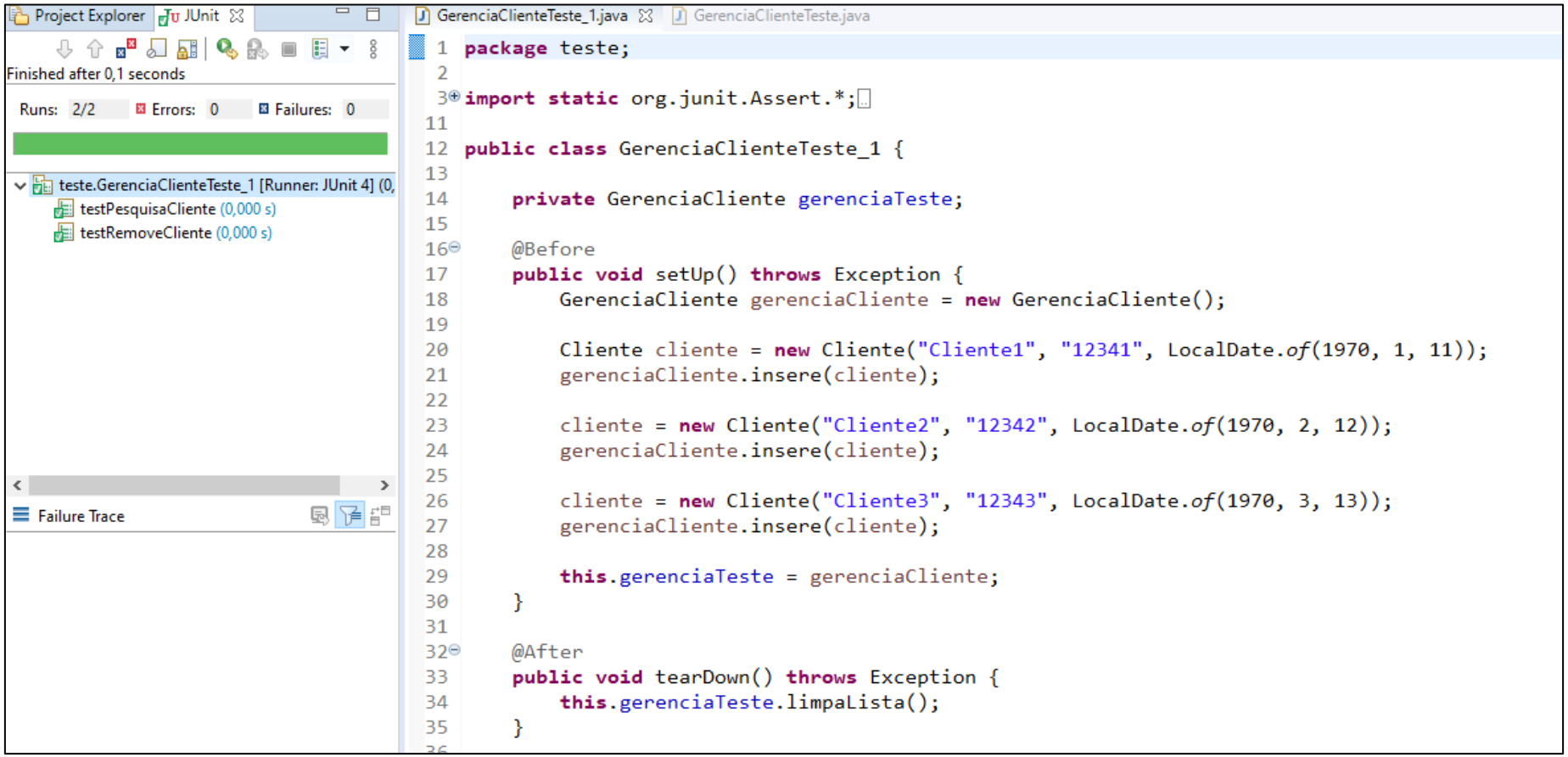
Inicialização e limpeza do teste

Execução dos métodos de teste

```java
@Test
public void testPesquisaCliente() {
    // Execução
    Cliente clienteRetornado = this.gerenciaTeste.pesquisa("12342");

    // Verificação
    assertEquals("12342", clienteRetornado.getCpf());
}

@Test
public void testRemoveCliente() {
    // Execução
    boolean clienteRemovido = this.gerenciaTeste.remove("12342");

    // Verificação
    assertTrue(clienteRemovido);
    assertNull(this.gerenciaTeste.pesquisa("12342"));
}
```

Testes automatizados

- Passo 3: Execução

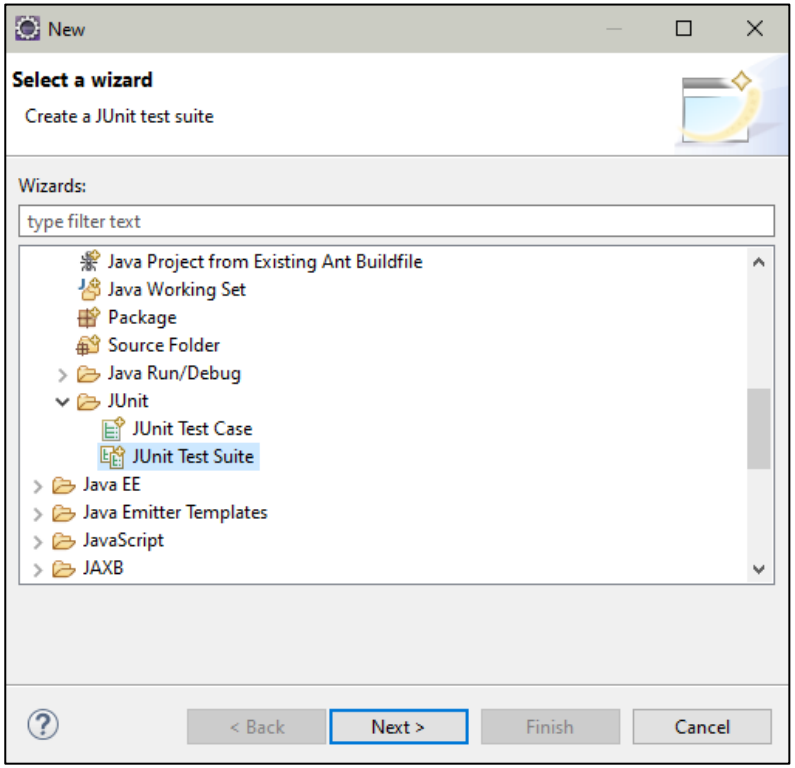


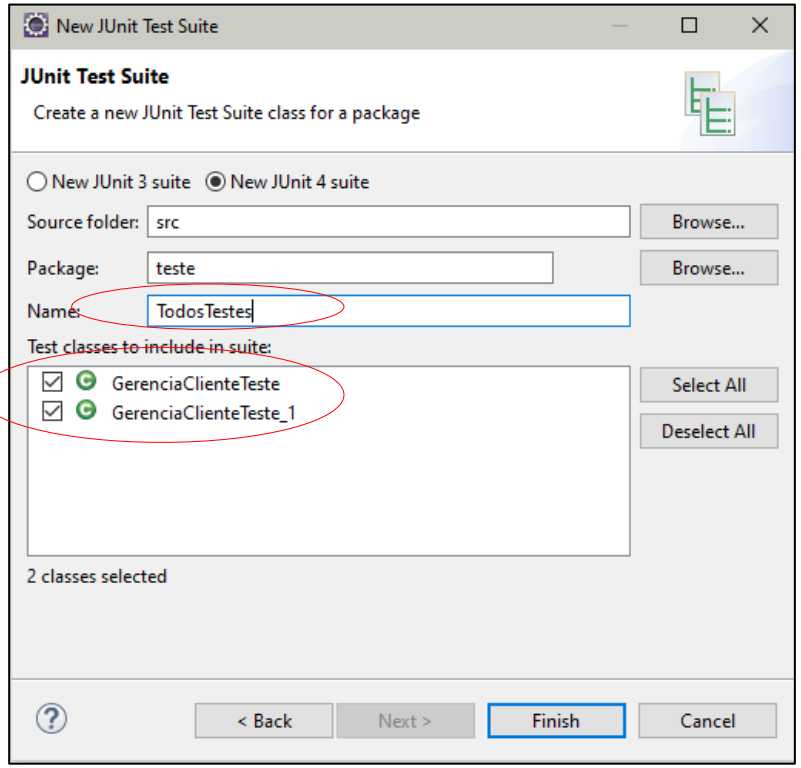
Testes automatizados

- *Suite* de testes
  - Até o momento, os testes podem ser executados:
    - Método em particular: escolhe-se o método de teste e executa o mesmo
    - Classe inteira: escolhe-se uma classe e executam-se todos métodos com @Test presentes na classe de teste
  - Através de uma *suite* de teste é possível executar várias classes de testes ao mesmo tempo
    - A vantagem está em ganhar tempo, pois muitos testes poderão ser realizados em conjunto.

Testes automatizados

- Construção de uma *suite* de testes
  - Cria-se uma classe JUnit Test Suite





INSTITUTO FEDERAL
Santa Catarina

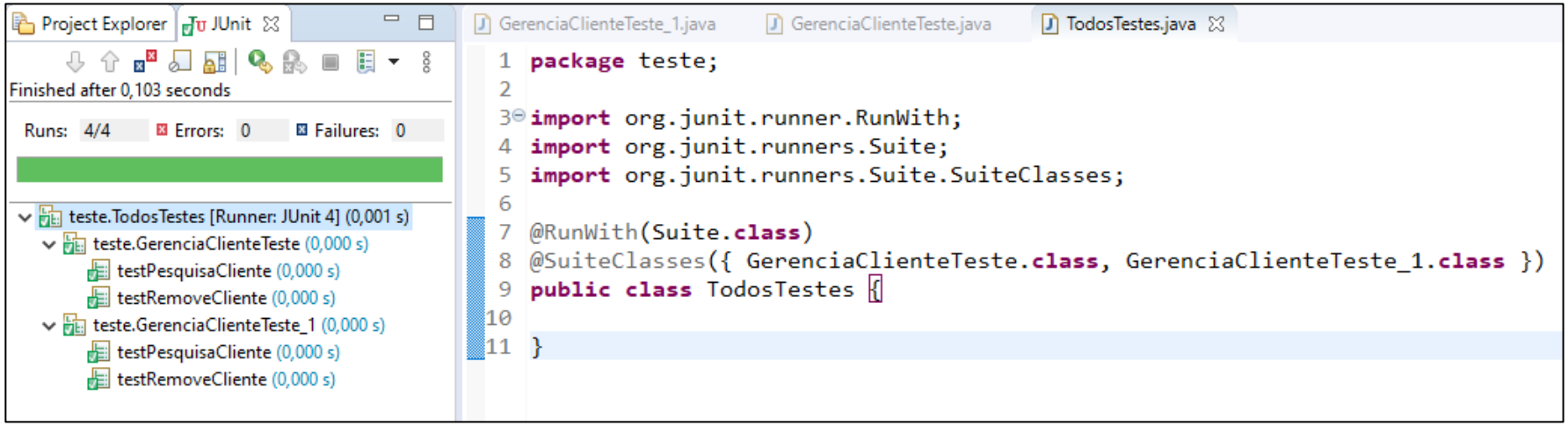
Testes automatizados

- Construção de uma *suite* de testes
  - Cria-se uma classe JUnit Test Suite

```
1  package teste;
2
3  import org.junit.runner.RunWith;
4  import org.junit.runners.Suite;
5  import org.junit.runners.Suite.SuiteClasses;
6
7  @RunWith(Suite.class)
8  @SuiteClasses({ GerenciaClienteTeste.class, GerenciaClienteTeste_1.class })
9  public class TodosTestes {
10
11 }
```

Testes automatizados

- Construção de uma *suite* de testes
  - Cria-se uma classe JUnit Test Suite



Testes automatizados

- Síntese
  - Testes independentes evitam que dados armazenados de outros testes influenciem novos testes
    - Aumento de produtividade
    - Mais claro e organizado o procedimento de teste
    - Mais fácil gerenciamento dos testes
  - *Suite* de testes pode disparar um conjunto de diversos testes automáticos
    - Aumento de produtividade
    - Mais fácil gerenciamento dos testes

INSTITUTO FEDERAL
Santa Catarina

Curso de Ciência da Computação

Engenharia de Software
Testes estruturais

Prof. Alexandre Perin de Souza
alexandre.perin@ifsc.edu.br

Lages (SC)

Teste estrutural

- Atividade
  - Leitura e estudo do material sobre teste estrutural caixa branca
  - Principalmente, analisar os exemplos e exercícios resolvidos
  - Dia 29/04/2020, as 09h, haverá reunião para esclarecer alguma dúvida.