

Teste de unidade e JavaDOC

P0029004 – Engenharia de Telecomunicações

Prof. Emerson Ribeiro de Mello

<http://docente.ifsc.edu.br/mello/poo>

15 DE ABRIL DE 2020



**INSTITUTO
FEDERAL**
Santa Catarina

Câmpus
São José

JavaDOC

- Gerador de documentação automática com base em comentários escritos no próprio código fonte
- Usado para documentar a API padrão do JDK
- Gera como resultado um conjunto de páginas HTML
- Pode ser usado para documentar mudanças entre versões de uma API (*doclets* e JDiff)



- Bloco de comentário de múltiplas linhas e que deve iniciar por `/**`
- Cada parágrafo é iniciado por um `*`
- Primeiro parágrafo é descrição abreviada do método ou classe
- Demais parágrafos são usados para descrição detalhada
- Algumas etiquetas (*tags*):
 - `@param` – para descrever o parâmetro de um método
 - `@return` – para descrever o retorno do método
 - `@author` – para indicar o autor do código



Exemplo

```
1 /**
2  * Classe que realiza a soma de inteiros
3  *
4  * @author Emerson Ribeiro de Mello
5  */
6 public class SomaInteiros {
7
8     /**
9      * Faz a soma de dois inteiros
10     * @param a operando 1
11     * @param b operando 2
12     * @return resultado da soma dos operandos
13     */
14     public int soma(int a, int b){
15         return (a+b);
16     }
17 }
```



- Menu Tools → Generate JavaDOC
- No campo "*Output directory*" escolher o diretório onde será armazenado a documentação gerada
- No campo "*Other command line arguments*" colocar:
`-encoding utf8 -charset utf8`



- Settings → Editor → General → Code completion
 - Marcar “Show the documentation popup in”
- Settings → Editor → General
 - Marcar “Show quick documentation on mouse move”



Teste de software

- Permite modificações de requisitos, mesmo no final do desenvolvimento do software
- Entregar software funcionando com frequência (entrega de pequenas versões funcionais)
- **Desenvolvimento guiado por testes**, cria-se primeiros os testes e depois implementa para atendê-los
- **Refatoração contínua do código** para melhorar sua qualidade, evitar redundância, aumentar sua clareza
 - Aplicar pequenas transformações preservando o comportamento



- Processo da Engenharia de Software que busca garantir a qualidade de software desenvolvido
- Escrever primeiro os testes ajuda a entender o projeto
- Ao **testar** um software, **umenta-se a confiabilidade e qualidade** do mesmo, porém **não é possível garantir** que **não existam erros**
- A **execução automática de testes** ajuda a identificar erros introduzidos devido a mudanças no código fonte



- Processo da Engenharia de Software que busca garantir a qualidade de software desenvolvido
- Escrever primeiro os testes ajuda a entender o projeto
- Ao **testar** um software, **umenta-se a confiabilidade e qualidade** do mesmo, porém **não é possível garantir** que **não existam erros**
- A **execução automática de testes** ajuda a identificar erros introduzidos devido a mudanças no código fonte

Myers, G.J. – *The Art of Software Testing*

O processo de testar consiste em executar um programa com o objetivo de encontrar erros.



- Caixa preta, orientada a dados ou orientada a entrada e saída
- **Testar** um componente **sem precisar conhecer a estrutura interna** do mesmo
- **Plano de testes é derivado somente das especificações** que foram usadas para desenvolver o software



Estratégia de teste: Caixa preta

- Caixa preta, orientada a dados ou orientada a entrada e saída
- **Testar** um componente **sem precisar conhecer a estrutura interna** do mesmo
- **Plano de testes é derivado somente das especificações** que foram usadas para desenvolver o software

Método para informar o tipo de um triângulo

- Entrada: 3, 3, 3
- Saída: equilátero



- Processo que visa testar individualmente módulo, componentes ou procedimentos (métodos) de um programa
- Comparar o funcionamento de um módulo com sua especificação
- Possibilita ver se a implementação de um módulo contradiz sua especificação
- Dependências externas (bibliotecas) devem ser removidas do teste, usando por exemplo, componentes de testes que simulam tais bibliotecas



Teste de unidade

- Processo que visa testar individualmente módulo, componentes ou procedimentos (métodos) de um programa
- Comparar o funcionamento de um módulo com sua especificação
- Possibilita ver se a implementação de um módulo contradiz sua especificação
- Dependências externas (bibliotecas) devem ser removidas do teste, usando por exemplo, componentes de testes que simulam tais bibliotecas

Teste de unidade não é adequado para

Testar interfaces de usuário ou interação entre componentes, nesse caso deve-se usar testes de integração



- JUnit é um framework para escrita de testes de unidade em Java
 - Faz uso de anotações Java (@Test)
- Um teste com JUnit consiste de um método em uma classe usada exclusivamente para teste
- Em projetos com Gradle ou Maven tem-se a seguinte organização:
 - `src/main/java` – Classes Java
 - `src/test/java` – Classes de teste



O que deve ser testado? Tudo?

- Testar componentes (métodos) complexos ou críticos é uma boa prática
- Não é necessário fazer testes para métodos triviais
 - Métodos *get* e *set* de cada atributo



Exemplo de um teste de unidade com JUnit4

```
18 import org.junit.Test;
19 import static org.junit.Assert.*;
20 public class TesteMultiplicador{
21
22     @Test
23     public void multiplicarPorZero(){
24         // classe a ser testada
25         MinhaClasse mc = new MinhaClasse();
26
27         // testes
28         // (string, valor esperado, valor obtido)
29         assertEquals("1 x 0 deve ser 0", 0, mc.multiplicar(1,0));
30         assertEquals("0 x 1 deve ser 0", 0, mc.multiplicar(0,1));
31         assertEquals("0 x 0 deve ser 0", 0, mc.multiplicar(0,0));
32     }
33 }
34 }
```



```
35 byte[] expected = "trial".getBytes();
36 byte[] actual = "trial".getBytes();
37 assertEquals("failure - byte arrays not same", expected, actual);
38
39 assertFalse("failure - should be false", false);
40
41 assertTrue("failure - should be true", true);
42
43 assertNotNull("should not be null", new Object());
44
45 assertNull("should be null", null);
```

- Mais informações na documentação oficial do projeto:
<https://github.com/junit-team/junit4/wiki>



Leia três lados de um triângulo e informe se os valores realmente podem formar um triângulo ou não. Se formar, então indique se este é um equilátero (três lados iguais), isósceles (quaisquer dois lados iguais) ou escaleno (três lados diferentes).

- Crie um projeto Java + gradle com IntelliJ
- Crie uma classe `Principal`
- Crie uma classe `Triangulo` com o método para indicar o tipo do triângulo
- Crie uma classe de testes `TestaTriangulo`
- Crie um método de testes para garantir que a classe `Triangulo` está correta



- Escrever os testes antes de implementar as funcionalidades
- Sistema é validado incrementalmente
- Pode evitar que a inclusão de novas funcionalidades introduzam *bugs* em códigos que funcionavam
 - Se um novo código não passar pelo novo teste ou fazer com que outros testes falhem, deve-se reverter as alterações introduzidas



Desenvolvimento guiado por testes

- Escrever os testes antes de implementar as funcionalidades
- Sistema é validado incrementalmente
- Pode evitar que a inclusão de novas funcionalidades introduzam *bugs* em códigos que funcionavam
 - Se um novo código não passar pelo novo teste ou fazer com que outros testes falhem, deve-se reverter as alterações introduzidas

Ciclo do *Test Drive Development* – TDD

- 1 **Escreva caso de teste** que define uma melhoria desejada ou uma nova funcionalidade
- 2 **Desenvolva a funcionalidade** de forma que **passe pelo caso de teste**
- 3 **Refatore o código** para se adequar a padrões de projeto ou para torná-lo mais elegante



Código bem refatorado

- Curto, compacto, claro e sem duplicação
- Parece trabalho de um programador experiente



Código bem refatorado

- Curto, compacto, claro e sem duplicação
- Parece trabalho de um programador experiente

Código malcheiroso

- **Código duplicado** – idêntico ou semelhante em mais de uma parte
- **Alto acomplamento** – uma simples alteração gera impacto em diversas classes ao mesmo tempo
- **Classe grande** – Classe com excesso de código
- **Uso excessivo de constantes literais** – `if (totalSoldado < 10)`



- **Extrair método** – transformar método longo em um mais curto, refatorando uma parte para um outro método auxiliar privado
- **Extrair constantes** – Substituir constantes literais por membros constantes da classe
- **Nomes explicativos de atributos e métodos** – Uso de nome que explica o propósito do método ou atributo, também válido para variáveis temporárias
- **Encapsular atributo** – convenção já usada desde a 1a. aula
- **Subir/descer atributo & Subir/descer método** – quando se faz uso de hierarquia de classes



- Crie uma classe para representar um robô de exploração espacial
- Crie **método para fazer a exploração**. Quando o **robô terminar a exploração**, esse deverá **indicar sua posição atual e para onde está sua frente**
- Plano composto por comandos: E, D, M
 - E - girar no próprio eixo para esquerda
 - D - girar no próprio eixo para direita
 - M - mover 1 coordenada para frente



Exemplo de refatoração: Robô explorador

```
46 public class Robo{
47     private int x, y, f; // nomes não sugestivos
48     public void movimentar(String plano){
49         for (int i = 0 ; i < plano.length(); i++){
50             char p = plano.charAt(i); // nome não sugestivo
51
52             if (p == 'D'){
53                 f = (f + 1 < 4) ? f + 1 : 0; // semelhante ao trecho abaixo
54             }
55             if (p == 'E'){
56                 f = (f - 1 >= 0) ? f - 1 : 3; // semelhante ao trecho acima
57             }
58             if (p == 'M'){
59                 switch (f){ // 0 que significa esses inteiros em f?
60                     case 0: y++; break;
61                     case 1: x++; break;
62                     case 2: y--; break;
63                     case 3: x--;
64                 }
65             } // Imprimir coordenadas não deve ser uma atribuição do movimentar
66             System.out.println("x: " + x + ", y: " + y + ", f: " + f);
67         }
68     }
```



Exemplo de refatoração: Robô explorador

```
69 public class Robo{
70     private int posicaoX;
71     private int posicaoY;
72     /**
73      * Possíveis valores: Norte = 0; Leste = 1, Sul = 2 e Oeste = 3
74      */
75     private int frente;
76
77     public void movimentar(String planoExploracao){
78         ...
79         ...
80     }
81
82     // separação de responsabilidades por métodos
83     public String toString(){
84         return "(" + posicaoX + ", " + posicaoY + ") " + frente;
85     }
86 }
```



Exemplo de refatoração: Robô explorador

```
87 public void movimentar(String planoExploracao){
88     for(String comando: planoExploracao.split("")){
89         int incremento = 1;
90
91         if (comando.equals("E")){
92             incremento = 3;
93         }
94         switch (comando){
95             case "D":
96             case "E":
97                 frente = (frente + incremento) % 4;
98                 break;
99             case "M":
100                 switch (frente){
101                     case 0: posicaoY++; break;
102                     case 1: posicaoX++; break;
103                     case 2: posicaoY--; break;
104                     case 3: posicaoX--;
105                 }
106             }
107         }
108     }
```



Exercícios

- Faça uma classe de teste para testar cada um dos exercícios desenvolvidos nas listas anteriores
- Também é necessário criar documentação com JavaDOC



- Ler a documentação oficial sobre testes parametrizados
<https://github.com/junit-team/junit4/wiki/parameterized-tests>
- Executar e entender os testes de exemplo apresentados em
<https://github.com/poo29004/teste-unidade-exemplo>

