

GESTÃO E QUALIDADE DE SOFTWARE

TESTES UNITÁRIOS EM JAVA OPERAÇÕES COM EMPREGADOS

Aluna: Lara Luísa Ayrolla Abreu

1. O QUE SÃO TESTES UNITÁRIOS E PARA QUE SERVEM?

Testes unitários são testes realizados com a intenção de testar individualmente pequenos pedaço de um código, que tendem a ser funções ou métodos.

Por mais que possam ser realizados manualmente, na maioria das vezes são criados dentro do próprio projeto a ser testado e executados automaticamente.

São muito importantes para verificar a funcionalidade de cada parte do código e ajudam a identificar problemas de forma mais rápida, pois, a falha de um teste unitário é um grande indicativo de que o trecho de código referente a esse teste está com algum erro ou *bug*.

2. TESTES UNITÁRIOS NA LINGUAGEM JAVA

Na linguagem Java, que será utilizada para demonstrar os testes unitários deste relatório, testes unitários são comumente feitos utilizando junit, que é uma biblioteca e framework com diversas notações e funcionalidades que facilitam a criação dos testes unitários e a automatização da execução deles.

3. FUNCIONALIDADES DO CÓDIGO: OPERAÇÕES COM EMPREGADOS

O código a ser testado foi desenvolvido em Java, no mesmo projeto em que os testes unitários foram desenvolvidos. Para acessar o repositório do código fonte, basta clicar [neste link](#).

No pacote “entity”, foi criada a classe “Employee”, na qual estão contidas variáveis comuns a todos os tipos de empregados e um construtor para preencher os valores dessas variáveis. Também foram criadas as classes “Manager”, representando gerente, e “Seller”, representando vendedor, que estendem a classe “Employee”, com diversas funções que fazem operações com comissão, horas extras, adicional noturno, prêmio e vendedor do mês.

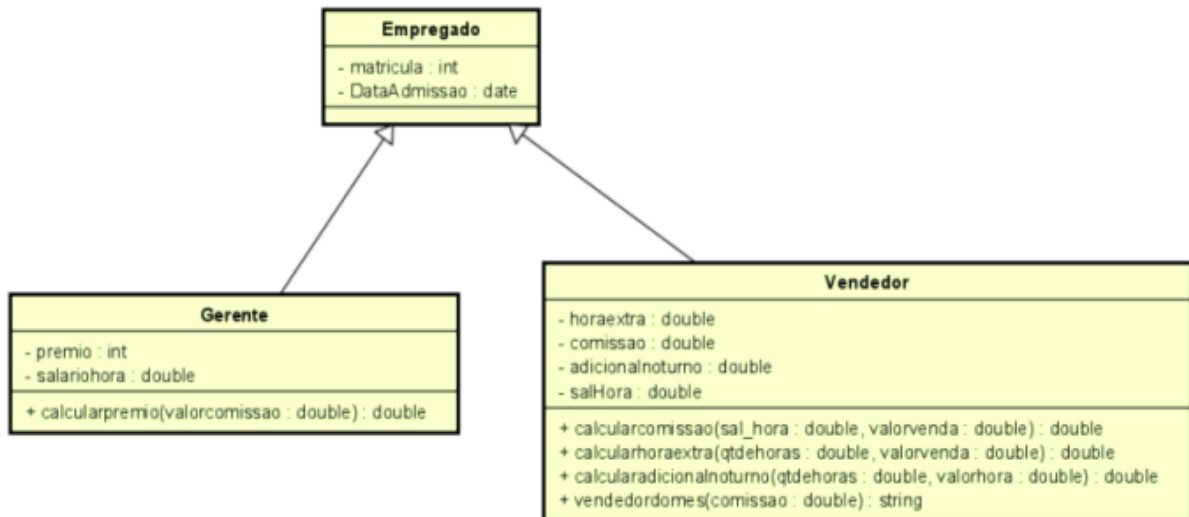


Diagrama em português que representa uma aproximação do código final

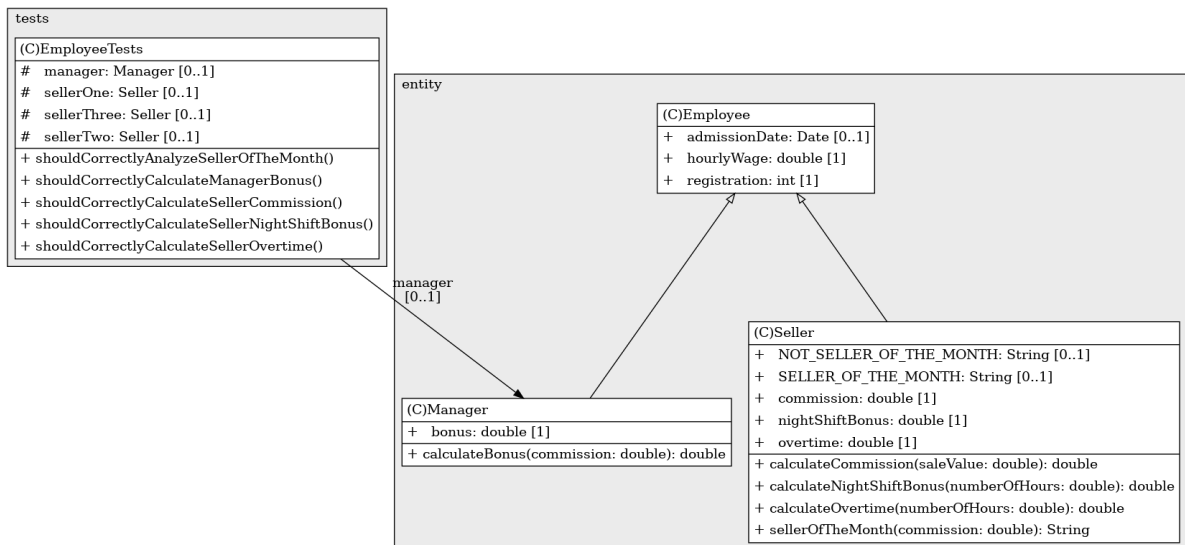


Diagrama em inglês que representa o código final + a classe de testes

a. Classe “Employee”

Ao começo da classe, foram criadas variáveis comuns a todos os tipos de empregados, que serão utilizadas ao longo do código pelos métodos da classe e das classes filhas. Há somente um método, que é um construtor utilizado para inicializar os valores dessas variáveis.

```

1 package entity;
2
3 import java.util.Date;
4
5 public class Employee {
6     public int registration;
7     public Date admissionDate;
8     public double hourlyWage;

```

Declaração da classe “Employee” e de variáveis

```

10 Employee (int registrationNumber, Date admission, double hourlySalary)
11 {
12     this.registration = registrationNumber;
13     this.admissionDate = admission;
14     this.hourlyWage = hourlySalary;
15 }

```

Construtor que inicializa as variáveis da classe

b. Classe “Manager”

A classe “Manager” estende a classe “Employee”. Por conta disso, ela herda todas as variáveis e redefine o construtor, fazendo uma chamada ao super — construtor da classe pai. Além disso, ela define a variável “bônus”.

Quanto à única função além do construtor, ela tem a funcionalidade de calcular o valor do bônus recebido pelo gerente, através da comissão, que é a soma da comissão de três vendedores.

```

1 package entity;
2
3 import java.util.Date;
4
5 3 usages
6 public class Manager extends Employee {
7     3 usages
8     public double bonus;
9
10    1 usage
11    public Manager (int registrationNumber, Date admission, double hourlySalary) {
12        super(registrationNumber, admission, hourlySalary);
13    }

```

Declaração da classe “Manager”, variável e construtor

```

12    2 usages
13    public double calculateBonus (double commission)
14    {
15        if (commission > 1000) {
16            this.bonus = 5000;
17        } else {
18            this.bonus = commission;
19        }
20
21    return this.bonus;
22    }

```

Função que calcula o bônus recebido pelo gerente

c. Classe “Seller”

Assim como a classe “Manager”, a classe “Seller” estende a classe “Employee”. Por conta disso, ela herda todas as variáveis e redefine o construtor, fazendo uma chamada ao super — construtor da classe pai. Além disso, ela define variáveis e constantes — variáveis do tipo final.

Quanto às funções, elas têm as funcionalidades de calcular comissão, horas extras, adicional noturno e dizer se um funcionário é o vendedor do mês ou não, utilizando as variáveis e constantes previamente definidas.

```

Seller.java x
1 package entity;
2
3 import java.util.Date;
4
5 7 usages
6 public class Seller extends Employee {
7     4 usages
8     public double overtime = 0;
9     8 usages
10    public double commission = 0;
11    4 usages
12    public double nightShiftBonus = 0;
13
14    2 usages
15    public final String SELLER_OF_THE_MONTH = "Seller of the month!";
16    3 usages
17    public final String NOT_SELLER_OF_THE_MONTH = "Not seller of the month.";
18
19    3 usages
20    public Seller (int registrationNumber, Date admission, double hourlySalary) {
21        super(registrationNumber, admission, hourlySalary);
22    }

```

Declaração da classe "Seller", variáveis, constantes e construtor

```

12 usages
17 public double calculateCommission (double saleValue)
18 {
19     double partialCommission = hourlyWage * saleValue * 0.01;
20
21     this.commission += partialCommission;
22
23     return partialCommission;
24 }

```

Função que calcula a comissão através do salário por hora e do valor da venda

```

6 usages
26 public double calculateOvertime (double numberOfHours)
27 {
28     double partialOvertime = hourlyWage * numberOfHours;
29
30     this.overtime += partialOvertime;
31
32     return partialOvertime;
33 }

```

Função que calcula o valor das horas extras através do salário por hora e do número de horas

```

6 usages
35 public double calculateNightShiftBonus (double numberOfHours)
36 {
37     double partialNightShiftBonus = hourlyWage * numberOfHours * 0.2;
38
39     this.nightShiftBonus += partialNightShiftBonus;
40
41     return partialNightShiftBonus;
42 }

```

Função que calcula o adicional noturno através do salário por hora e do número de horas

```

3 usages
44 public String sellerOfTheMonth (double commission)
45 {
46     if (this.commission >= commission) {
47         return SELLER_OF_THE_MONTH;
48     }
49
50     return NOT_SELLER_OF_THE_MONTH;
51 }

```

Função que verifica se é o vendedor do mês ou não, utilizando constantes da classe "Seller"

4. TESTES DESENVOLVIDOS

No pacote "tests", foi criada a classe "EmployeeTests", na qual estão contidos os métodos utilizados para testar as funções das classes "Employee", "Manager" e "Seller".

Ao começo da classe, são feitas as importações das classes que serão testadas, da biblioteca que será utilizada para o desenvolvimento dos testes ("JUnit") e são instanciados objetos de nome "manager" do tipo "Manager", e "sellerOne", "sellerTwo" e "sellerThree" do tipo "Seller", que serão utilizados para chamar as funções das classes testadas.

Foram desenvolvidos cinco métodos, com trinta e dois casos de teste, que verificam se o funcionamento das funções das classes "Employee", "Manager" e "Seller" está correto. Isso é feito através de comparações dos retornos das funções testadas com valores previamente definidos pelos testes como corretos ou incorretos.

Um teste passa somente se o resultado retornado pela função testada, quando recebe os valores predefinidos como parâmetro, coincide com o resultado esperado. Todos os casos de teste de um método de teste devem passar para que esse método passe.

```

EmployeeTests.java x
1 package tests;
2
3 import entity.Manager;
4 import entity.Seller;
5 import org.junit.*;
6
7 import java.util.Date;
8
9 public class EmployeeTests {
10     14 usages
11     Seller sellerOne = new Seller( registrationNumber: 1, new Date(), hourlySalary: 20);
12     14 usages
13     Seller sellerTwo = new Seller( registrationNumber: 2, new Date(), hourlySalary: 20);
14     14 usages
15     Seller sellerThree = new Seller( registrationNumber: 3, new Date(), hourlySalary: 20);
16     2 usages
17     Manager manager = new Manager( registrationNumber: 4, new Date(), hourlySalary: 30);

```

Declaração da classe “EmployeeTests”, importações e instâncias de objetos para teste

```

15 @Test
16 public void shouldCorrectlyCalculateManagerBonus ()
17 {
18     sellerOne.calculateCommission( saleValue: 210);
19     sellerTwo.calculateCommission( saleValue: 1100);
20     sellerThree.calculateCommission( saleValue: 2010);
21
22     double commissionSum = sellerOne.commission + sellerTwo.commission + sellerThree.commission;
23
24     Assert.assertEquals( expected: 664, manager.calculateBonus(commissionSum), delta: 0);
25     Assert.assertEquals( expected: 5000, manager.calculateBonus( commission: commissionSum*2), delta: 0);
26 }

```

Método que testa o cálculo do prêmio/bônus de gerentes

```

28 @Test
29 public void shouldCorrectlyCalculateSellerCommission ()
30 {
31     Assert.assertEquals( expected: 40, sellerOne.calculateCommission( saleValue: 200), delta: 0);
32     Assert.assertEquals( expected: 200, sellerTwo.calculateCommission( saleValue: 1000), delta: 0);
33     Assert.assertEquals( expected: 2, sellerThree.calculateCommission( saleValue: 10), delta: 0);
34
35     Assert.assertEquals( expected: 2, sellerOne.calculateCommission( saleValue: 10), delta: 0);
36     Assert.assertEquals( expected: 20, sellerTwo.calculateCommission( saleValue: 100), delta: 0);
37     Assert.assertEquals( expected: 400, sellerThree.calculateCommission( saleValue: 2000), delta: 0);
38
39     Assert.assertEquals( expected: 42, sellerOne.commission, delta: 0);
40     Assert.assertEquals( expected: 220, sellerTwo.commission, delta: 0);
41     Assert.assertEquals( expected: 402, sellerThree.commission, delta: 0);
42 }

```

Método que testa o cálculo da comissão de vendedores

```

44      @Test
45      public void shouldCorrectlyCalculateSellerOvertime ()
46      {
47          Assert.assertEquals( expected: 2000, sellerOne.calculateOvertime( numberOfHours: 100), delta: 0);
48          Assert.assertEquals( expected: 400, sellerTwo.calculateOvertime( numberOfHours: 20), delta: 0);
49          Assert.assertEquals( expected: 200, sellerThree.calculateOvertime( numberOfHours: 10), delta: 0);
50
51          Assert.assertEquals( expected: 2000, sellerOne.calculateOvertime( numberOfHours: 100), delta: 0);
52          Assert.assertEquals( expected: 400, sellerTwo.calculateOvertime( numberOfHours: 20), delta: 0);
53          Assert.assertEquals( expected: 200, sellerThree.calculateOvertime( numberOfHours: 10), delta: 0);
54
55          Assert.assertEquals( expected: 4000, sellerOne.overtime, delta: 0);
56          Assert.assertEquals( expected: 800, sellerTwo.overtime, delta: 0);
57          Assert.assertEquals( expected: 400, sellerThree.overtime, delta: 0);
58      }

```

Método que testa o cálculo do valor das horas extras de vendedores

```

60      @Test
61      public void shouldCorrectlyCalculateSellerNightShiftBonus ()
62      {
63          Assert.assertEquals( expected: 400, sellerOne.calculateNightShiftBonus( numberOfHours: 100), delta: 0);
64          Assert.assertEquals( expected: 80, sellerTwo.calculateNightShiftBonus( numberOfHours: 20), delta: 0);
65          Assert.assertEquals( expected: 40, sellerThree.calculateNightShiftBonus( numberOfHours: 10), delta: 0);
66
67          Assert.assertEquals( expected: 400, sellerOne.calculateNightShiftBonus( numberOfHours: 100), delta: 0);
68          Assert.assertEquals( expected: 80, sellerTwo.calculateNightShiftBonus( numberOfHours: 20), delta: 0);
69          Assert.assertEquals( expected: 40, sellerThree.calculateNightShiftBonus( numberOfHours: 10), delta: 0);
70
71          Assert.assertEquals( expected: 800, sellerOne.nightShiftBonus, delta: 0);
72          Assert.assertEquals( expected: 160, sellerTwo.nightShiftBonus, delta: 0);
73          Assert.assertEquals( expected: 80, sellerThree.nightShiftBonus, delta: 0);
74      }

```

Método que testa o cálculo do adicional noturno de vendedores

```

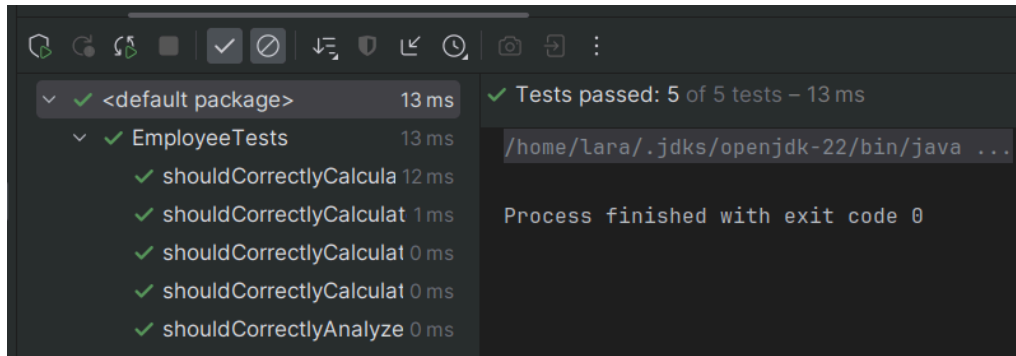
76      @Test
77      public void shouldCorrectlyAnalyzeSellerOfTheMonth ()
78      {
79          double biggestCommission = 402;
80
81          sellerOne.calculateCommission( saleValue: 2000);
82          sellerTwo.calculateCommission( saleValue: biggestCommission*5);
83          sellerThree.calculateCommission( saleValue: 1200);
84
85          Assert.assertEquals(
86              sellerOne.NOT_SELLER_OF_THE_MONTH,
87              sellerOne.sellerOfTheMonth(biggestCommission)
88          );
89
90          Assert.assertEquals(
91              sellerTwo.SELLER_OF_THE_MONTH,
92              sellerTwo.sellerOfTheMonth(biggestCommission)
93          );
94
95          Assert.assertEquals(
96              sellerThree.NOT_SELLER_OF_THE_MONTH,
97              sellerThree.sellerOfTheMonth(biggestCommission)
98          );
99      }

```

Método que testa a verificação de vendedor do mês, utilizando constantes da classe “Seller”

5. RESULTADOS E CONCLUSÃO

Após a execução dos testes da classe “EmployeeTests”, todos os trinta e dois casos de teste e, conseqüentemente, todos os cinco métodos, passaram.



Resultado da execução dos testes unitários

Além disso, 100% das funções e linhas do código foram testadas, o que pôde ser comprovado utilizando uma ferramenta da IDE IntelliJ do Jet Brains.

A screenshot of the IntelliJ IDEA Coverage tool window. The title bar reads 'Coverage All in EmployeeOperations (1)'. Below the title bar is a toolbar with icons for coverage analysis. The main area is a table showing coverage percentages for different elements in the project. All elements show 100% coverage for classes, methods, and lines.

Element ^	Class, %	Method, %	Line, %
all	100% (4/4)	100% (13/13)	100% (73/73)
entity	100% (3/3)	100% (8/8)	100% (27/27)
Employee	100% (1/1)	100% (1/1)	100% (4/4)
Manager	100% (1/1)	100% (2/2)	100% (5/5)
Seller	100% (1/1)	100% (5/5)	100% (18/18)
tests	100% (1/1)	100% (5/5)	100% (46/46)
EmployeeTests	100% (1/1)	100% (5/5)	100% (46/46)

Porcentagem de classes, métodos e linhas testadas em todo o projeto

O resultado foi satisfatório e atingiu as expectativas. Isso ocorreu, porque os testes foram feitos de maneira cautelosa, utilizando valores adequados para a comparação com os retornos das funções, e as classes testadas cumpriram o objetivo de criar códigos de qualidade que produzem as soluções esperadas.

É importante ressaltar que, mesmo que todas as funções das classes tenham sido testadas e todos os testes tenham passado, não é possível garantir a ausência de *bugs*. Algum caso específico pode não ter sido explorado.

Em adição, é possível que mudanças futuras no código criem *bugs* ou façam com que os testes deixem de passar. Nesse caso, o *bug* deve ser corrigido ou o teste ajustado para funcionar com a nova versão do código.

Por conta dessas incertezas, é importante criar uma base grande de testes, para identificar erros em todos os trechos do código o mais rápido possível, executar os testes com frequência e dar manutenção no código e nos testes.

6. REFERÊNCIAS

DevMedia. **E aí? Como você testa seus códigos?** Disponível em: <https://www.devmedia.com.br/e-ai-como-voce-testa-seus-codigos/39478>. Acesso em: 2 abr. 2024.

jUnit 5. Disponível em: <https://junit.org/junit5/>. Acesso em: 2 abr. 2024.

Testing Company. **Testes Automatizados e Unitários: Entenda as suas características e diferenças.** Disponível em: <https://testingcompany.com.br/blog/testes-automatizados-e-unitarios-entenda-as-suas-caracteristicas-e-diferencas#:~:text=Comumente%2C%20testes%20unit%C3%A1rios%20s%C3%A3o%20desenvolvidos,nas%20fases%20iniciais%20do%20projeto>. Acesso em: 2 abr. 2024.