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 extendem a classe “Employee”, com diversas funções que fazem operações com comissão, horas extras, adicional noturno, prêmio e vendedor do mês.

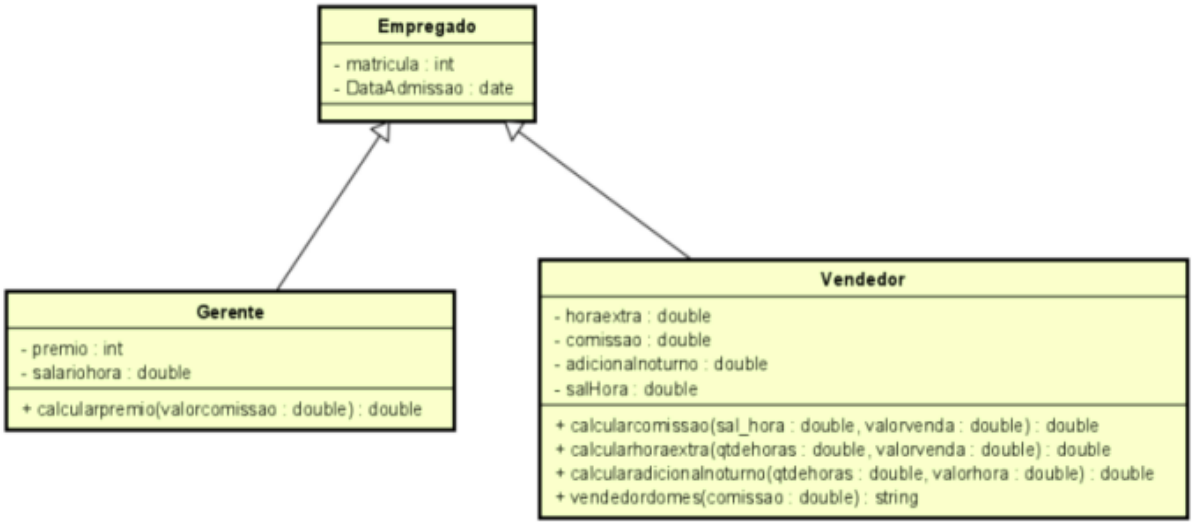


Diagrama em português que representa uma aproximação do código final

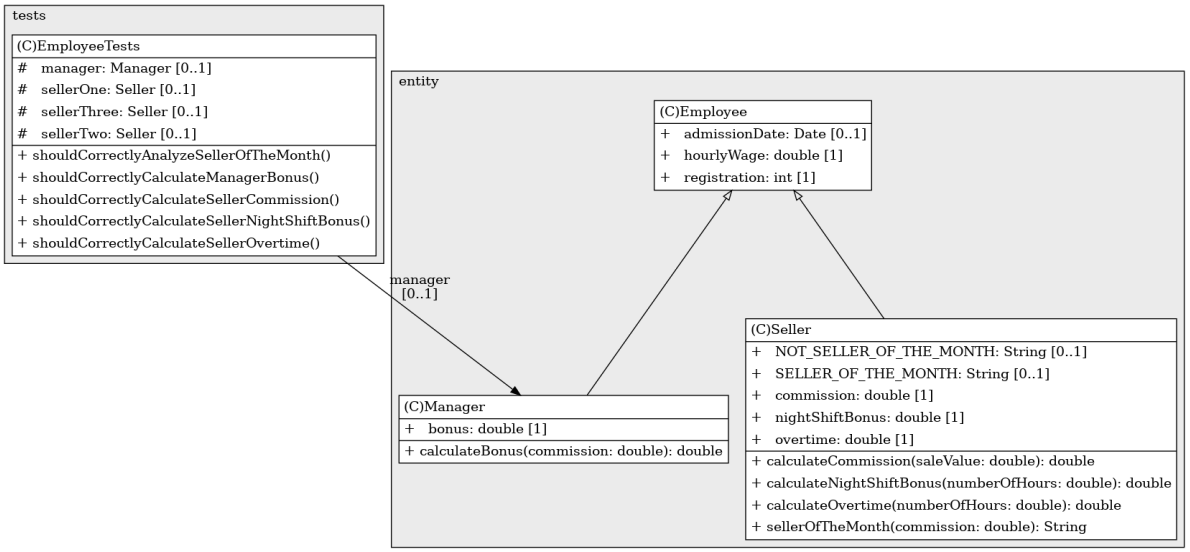
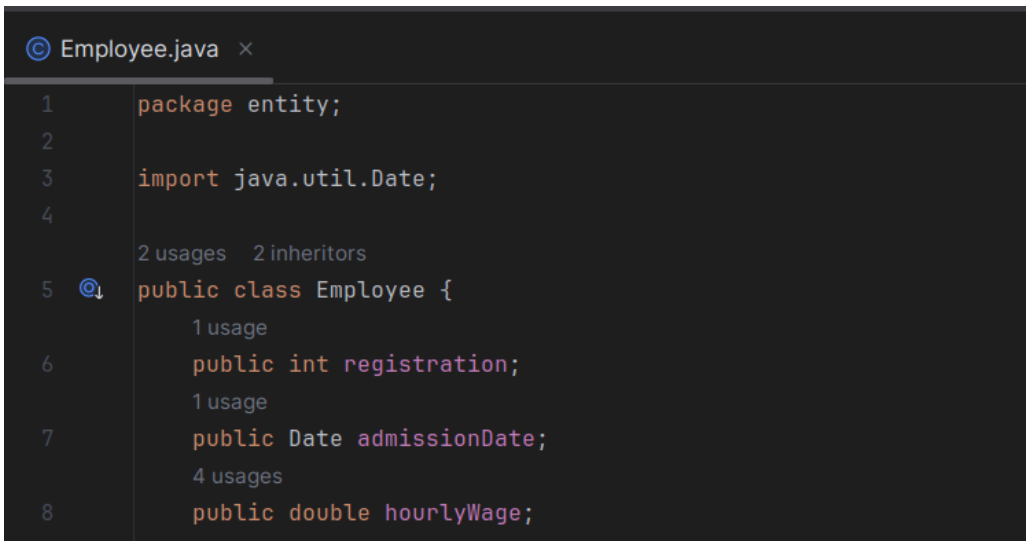


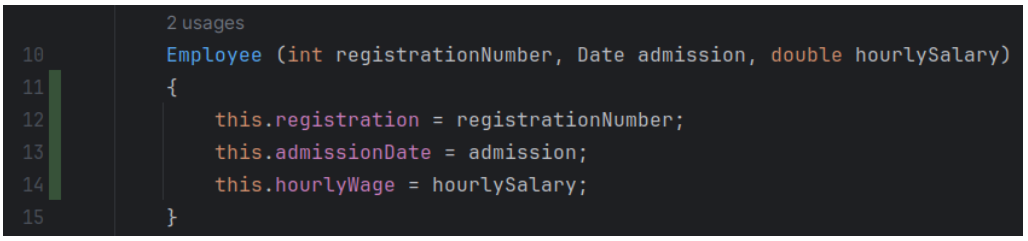
Diagrama em inglês que representa o código final + a classe de testes

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.



Declaração da classe “Employee” e de variáveis

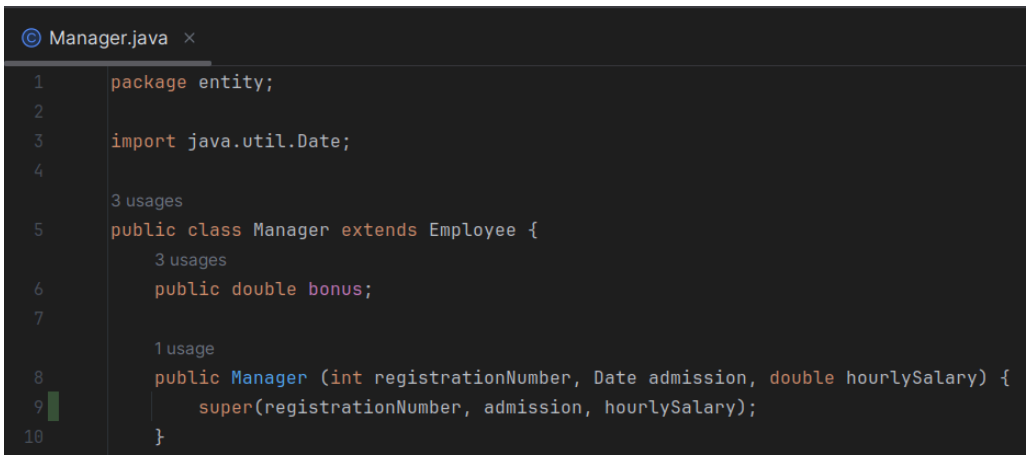


Construtor que inicializa as variáveis da classe

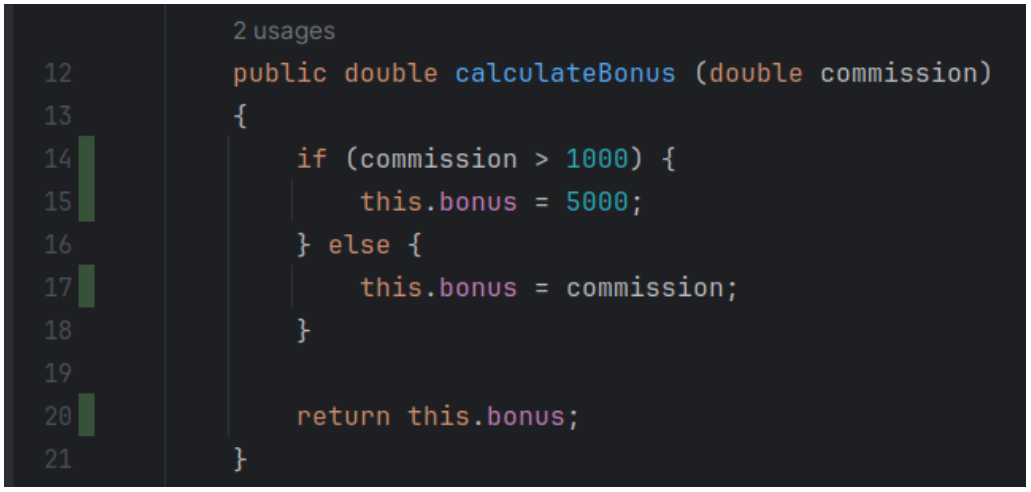
b. Classe “Manager”

A classe “Manager” extende 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 comisão de três vendedores.



Declaração da classe “Manager”, variável e construtor



Função que calcula o bônus recebido pelo gerente

c. Classe “Seller”

Assim como a classe “Manager”, a classe “Seller” extende 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

```java
package entity;

import java.util.Date;

7 usages
public class Seller extends Employee {
    4 usages
    public double overtime = 0;
    8 usages
    public double commission = 0;
    4 usages
    public double nightShiftBonus = 0;

    2 usages
    public final String SELLER_OF_THE_MONTH = "Seller of the month!";
    3 usages
    public final String NOT_SELLER_OF_THE_MONTH = "Not seller of the month.";

    3 usages
    public Seller (int registrationNumber, Date admission, double hourlySalary) {
        super(registrationNumber, admission, hourlySalary);
    }
```

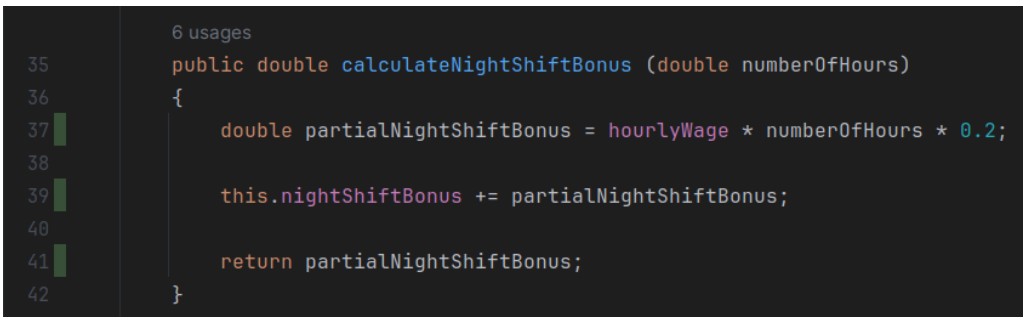
Declaração da classe “Seller”, variáveis, contantes e construtor

```java
    12 usages
    public double calculateCommission (double saleValue)
    {
        double partialCommission = hourlyWage * saleValue * 0.01;

        this.commission += partialCommission;

        return partialCommission;
    }
```

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

```java
    6 usages
    public double calculateOvertime (double numberOfHours)
    {
        double partialOvertime = hourlyWage * numberOfHours;

        this.overtime += partialOvertime;

        return partialOvertime;
    }
```

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



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

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

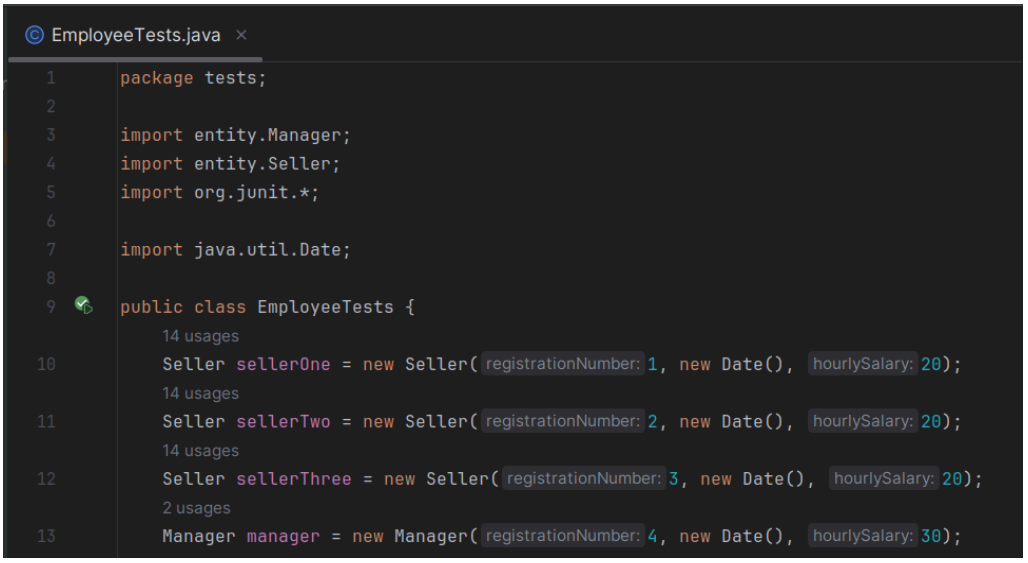
4. TESTES DESENVOLVIDOS

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”.

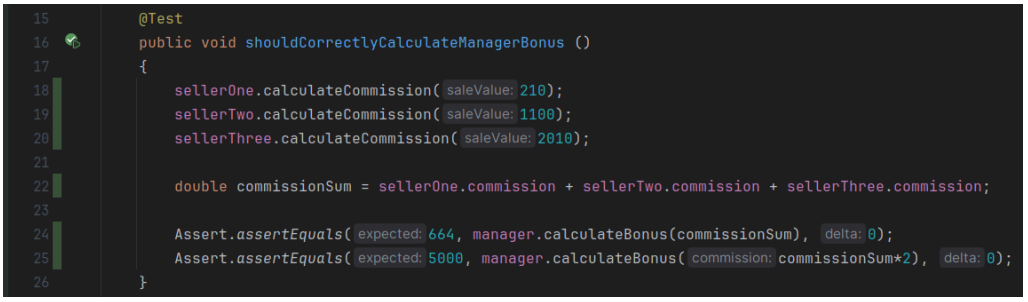
Ao começo da classe, são feitas as importações da classes que serão testadas, da biblioteca que será utilizada para o desenvolvimento dos testes (“jUnit”) e são instanciados objetos de nome “manager” do tipo “Manager”, e “sellerOne”, “sellerTwo” e “sellerThree” do tipo “Seller”, que serão utilizados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

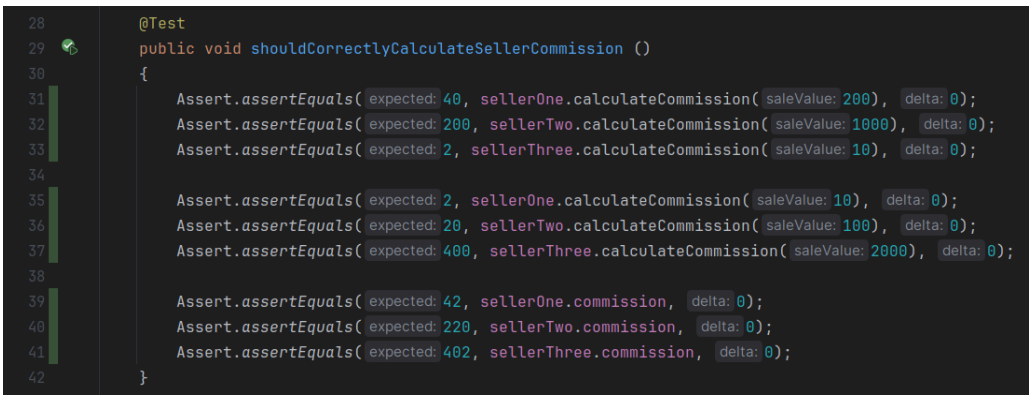
Um teste passa somente se o resultado retornado pela função testada, quando recebe os valores predefinidos como parâmetro, coincide com o resultado esperado. Todos os casos de teste de um método de teste devem passar para que esse método passe.



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



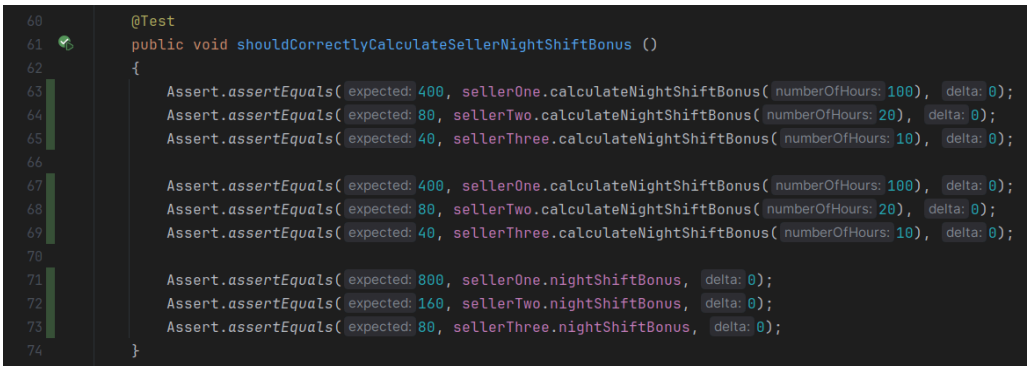
Método que testa o cálculo do prêmio/bônus de gerentes



Método que testa o cálculo da comissão de vendedores

```
@Test
public void shouldCorrectlyCalculateSellerOvertime ()
{
    Assert.assertEquals( expected: 2000, sellerOne.calculateOvertime( numberOfHours: 100), delta: 0);
    Assert.assertEquals( expected: 400, sellerTwo.calculateOvertime( numberOfHours: 20), delta: 0);
    Assert.assertEquals( expected: 200, sellerThree.calculateOvertime( numberOfHours: 10), delta: 0);

    Assert.assertEquals( expected: 2000, sellerOne.calculateOvertime( numberOfHours: 100), delta: 0);
    Assert.assertEquals( expected: 400, sellerTwo.calculateOvertime( numberOfHours: 20), delta: 0);
    Assert.assertEquals( expected: 200, sellerThree.calculateOvertime( numberOfHours: 10), delta: 0);

    Assert.assertEquals( expected: 4000, sellerOne.overtime, delta: 0);
    Assert.assertEquals( expected: 800, sellerTwo.overtime, delta: 0);
    Assert.assertEquals( expected: 400, sellerThree.overtime, delta: 0);
}
```

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



```
@Test
public void shouldCorrectlyCalculateSellerNightShiftBonus ()
{
    Assert.assertEquals( expected: 400, sellerOne.calculateNightShiftBonus( numberOfHours: 100), delta: 0);
    Assert.assertEquals( expected: 80, sellerTwo.calculateNightShiftBonus( numberOfHours: 20), delta: 0);
    Assert.assertEquals( expected: 40, sellerThree.calculateNightShiftBonus( numberOfHours: 10), delta: 0);

    Assert.assertEquals( expected: 400, sellerOne.calculateNightShiftBonus( numberOfHours: 100), delta: 0);
    Assert.assertEquals( expected: 80, sellerTwo.calculateNightShiftBonus( numberOfHours: 20), delta: 0);
    Assert.assertEquals( expected: 40, sellerThree.calculateNightShiftBonus( numberOfHours: 10), delta: 0);

    Assert.assertEquals( expected: 800, sellerOne.nightShiftBonus, delta: 0);
    Assert.assertEquals( expected: 160, sellerTwo.nightShiftBonus, delta: 0);
    Assert.assertEquals( expected: 80, sellerThree.nightShiftBonus, delta: 0);
}
```

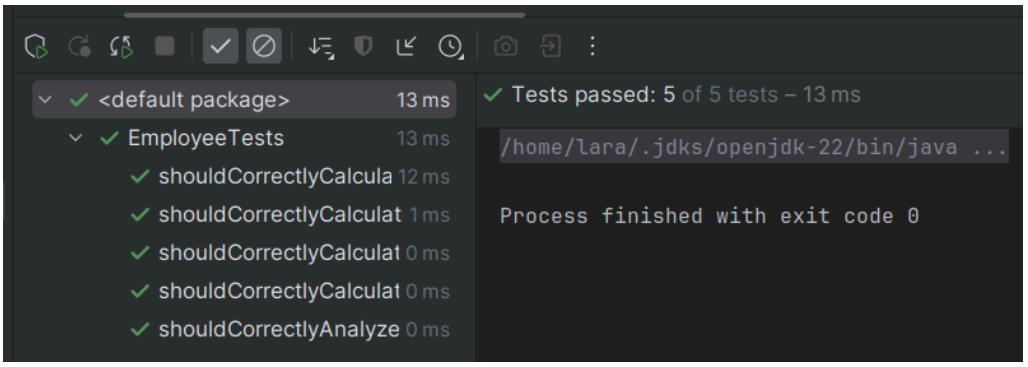
Método que testa o cálculo do adicional noturno de vendedores

```
@Test
public void shouldCorrectlyAnalyzeSellerOfTheMonth ()
{
    double biggestCommission = 402;

    sellerOne.calculateCommission( saleValue: 2000);
    sellerTwo.calculateCommission( saleValue: biggestCommission*5);
    sellerThree.calculateCommission( saleValue: 1200);

    Assert.assertEquals(
            sellerOne.NOT_SELLER_OF_THE_MONTH,
            sellerOne.sellerOfTheMonth(biggestCommission)
    );

    Assert.assertEquals(
            sellerTwo.SELLER_OF_THE_MONTH,
            sellerTwo.sellerOfTheMonth(biggestCommission)
    );

    Assert.assertEquals(
            sellerThree.NOT_SELLER_OF_THE_MONTH,
            sellerThree.sellerOfTheMonth(biggestCommission)
    );
}
```

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

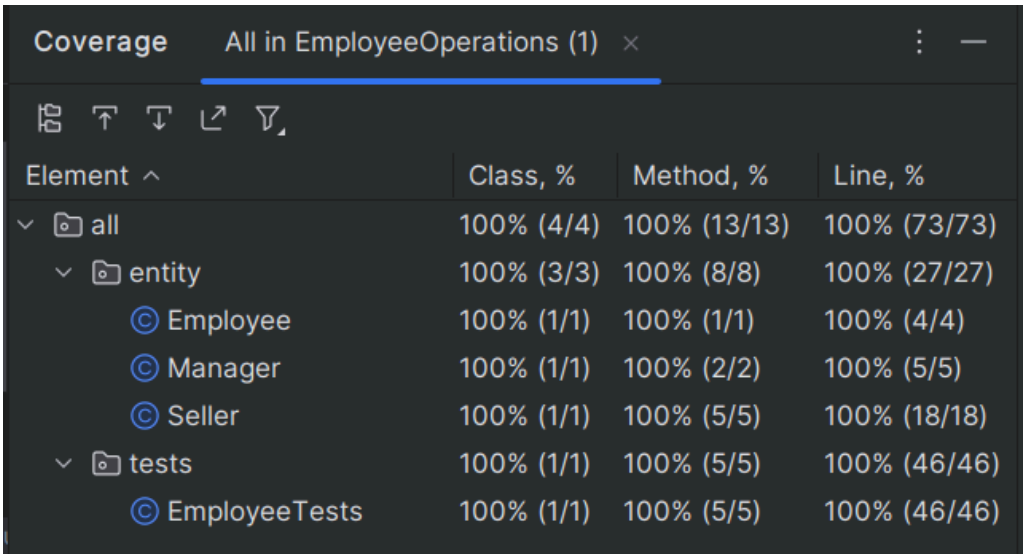
5. RESULTADOS E CONCLUSÃO

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



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.