



INSTITUTO INFNET
ENGENHARIA DE SOFTWARE

DAVID WILLIAM ALVES LOPES

TAQUARA
RIO GRANDE DO SUL
2024

Importância dos Testes:

Testar o software durante o desenvolvimento é essencial para garantir sua qualidade e confiabilidade. Os testes, e especificamente os testes unitários, são uma prática fundamental nesse processo. Eles ajudam a identificar problemas no código desde cedo, o que é crucial para evitar complicações mais tarde no desenvolvimento. Com os testes unitários, podemos detectar esse tipo de problema rapidamente e corrigi-lo antes que se torne um grande obstáculo.

Benefícios dos Testes Unitários:

Tomando como base o código do último TP, os testes unitários nos permitem detectar bugs precocemente. Por exemplo, se tivéssemos um teste unitário para o método `calcularTotal()` da classe `PedidoPizza`, poderíamos verificar se o preço total é calculado corretamente para diferentes combinações de sabores, tamanhos e quantidades. Isso nos ajudaria a identificar erros na lógica de cálculo antes mesmo de executar o programa. Além disso, os testes unitários ajudam a reduzir os custos de correção, pois é muito mais barato corrigir um bug logo após ele ser introduzido do que mais tarde, quando o código já está integrado ao sistema. Eles também contribuem para a melhoria da manutenibilidade do código, pois fornecem uma forma rápida e automatizada de verificar se as alterações introduzidas no código introduzem novos bugs ou afetam o comportamento existente.

Correlação com Exemplos Práticos:

Utilizei testes unitários para verificar se o cálculo do preço total está correto, pois então posso facilmente detectar se houve uma mudança na lógica de cálculo que resultou em preços errados. Por exemplo, se alguém alterasse acidentalmente o valor do preço da pizza média para R\$ 30, os testes unitários falhariam, indicando que algo está errado. Assim, podemos corrigir o problema imediatamente, garantindo que os preços das pizzas estejam sempre corretos para os clientes.

Tipos de Testes:

Os testes unitários focam em testar unidades individuais de código, como métodos ou funções, de forma isolada. Eles são escritos pelos próprios desenvolvedores e são executados com frequência durante o desenvolvimento.

Os testes de integração, por outro lado, verificam como diferentes partes do sistema funcionam juntas. Eles garantem que os componentes individuais se integrem corretamente e funcionem em conjunto como esperado. Por fim, os testes de aceitação avaliam se o software atende aos requisitos do usuário final e são geralmente realizados por testadores ou usuários finais em ambientes semelhantes ao de produção.

Comparação e Contraste:

Os testes unitários são realizados em unidades individuais de código e são escritos pelos próprios desenvolvedores. Eles são rápidos de executar e fornecem feedback imediato sobre a qualidade do código.

Os testes de integração garantem que diferentes partes do sistema funcionem juntas de forma harmoniosa. Eles são mais lentos e complexos do que os testes unitários, pois envolvem a interação entre diferentes componentes. Já os testes de aceitação verificam se o software atende aos requisitos do usuário final e são realizados em um ambiente semelhante ao de produção, garantindo que o software seja funcional e usável.

Utilização de Exemplos Práticos:

Os testes unitários podem ser usados para verificar se uma função de adicionar itens ao carrinho funciona corretamente, se o cálculo do preço total está correto e se o processo de checkout é executado sem problemas.

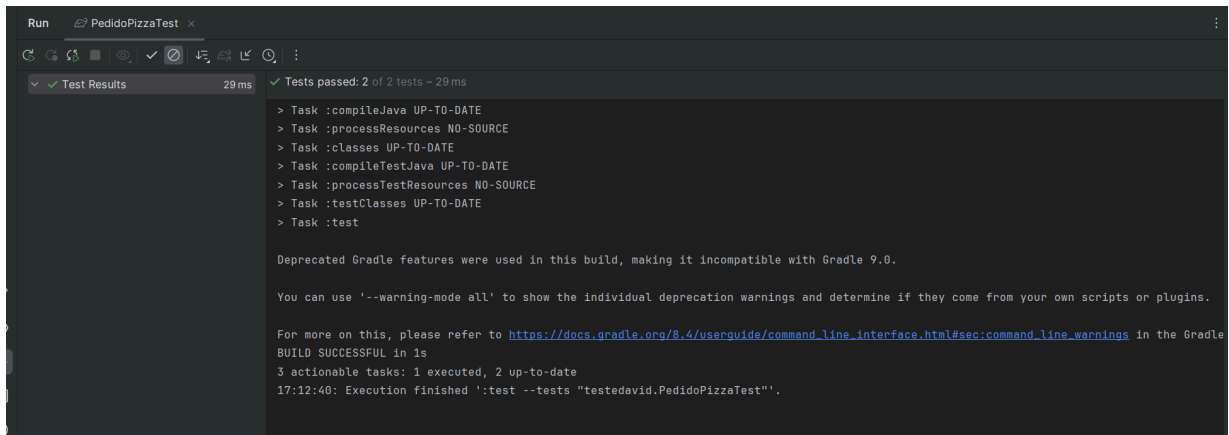
Os testes de integração, por sua vez, garantiriam que o sistema de pagamentos se integre corretamente com o sistema de carrinho de compras e que todos os componentes funcionem em conjunto. Por fim, os testes de aceitação seriam realizados pelos usuários finais para garantir que o sistema atenda às suas expectativas e requisitos de uso.

Escrita de Códigos de Teste:

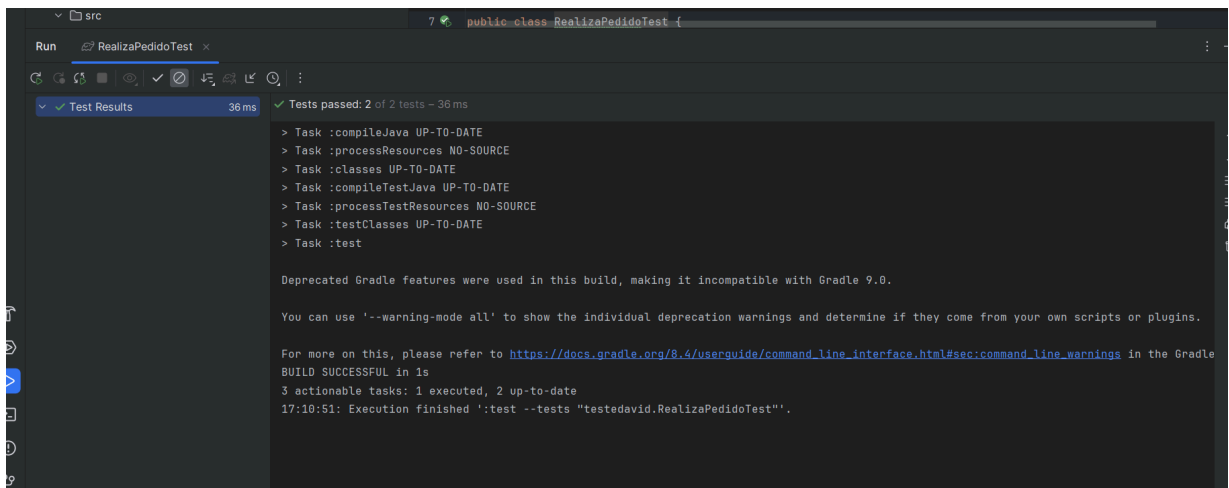
```
1 package testedavid;
2
3 import static org.junit.jupiter.api.Assertions.assertEquals;
4
5 import testedavid.example.PedidoPizza;
6 import org.junit.jupiter.api.BeforeEach;
7 import org.junit.jupiter.api.Test;
8
9 public class PedidoPizzaTest {
10
11     private PedidoPizza pedido;
12
13     @BeforeEach
14     public void setUp() { pedido = new PedidoPizza(sabor: "Calabresa e mocoto", tamanho: "média", quantidade: 2, paraEntrega: true); }
15
16     @Test
17     public void testCalcularTotalParaEntrega() {
18         double totalEsperado = (25.0 * 2) + 5.0; // preço médio * quantidade + taxa de entrega
19         assertEquals(totalEsperado, pedido.calcularTotal());
20     }
21
22     @Test
23     public void testCalcularTotalSemEntrega() {
24         pedido.setParaEntrega(false);
25         double totalEsperado = 25.0 * 2; // preço médio * quantidade
26         assertEquals(totalEsperado, pedido.calcularTotal());
27     }
28 }
29
30
31
```

```
PedidoPizzaTest.java RealizaPedidoTest.java x
1 package testedavid;
2 import testedavid.example.PedidoPizza;
3 import org.junit.jupiter.api.Test;
4
5 import static org.junit.jupiter.api.Assertions.assertEquals;
6
7 public class RealizaPedidoTest {
8
9     @Test
10     public void testPedido1() {
11         PedidoPizza pedido1 = new PedidoPizza(sabor: "Calabresa e mocoto", tamanho: "média", quantidade: 2, paraEntrega: true);
12         double totalEsperado = (25.0 * 2) + 5.0; // preço médio * quantidade + taxa de entrega
13         assertEquals(totalEsperado, pedido1.calcularTotal());
14     }
15
16     @Test
17     public void testPedido2() {
18         PedidoPizza pedido2 = new PedidoPizza(sabor: "Salame com churros", tamanho: "grande", quantidade: 1, paraEntrega: true);
19         double totalEsperado = 30.0; // preço grande * quantidade
20         assertEquals(totalEsperado, pedido2.calcularTotal());
21     }
22
23     // Adicione mais testes para outros pedidos com diferentes cenários
24 }
25
26
27
28
29
30
31
```

Relatório de Cobertura de Testes:



The screenshot shows the 'Run' window of an IDE with the tab 'PedidoPizzaTest'. The 'Test Results' section indicates 'Tests passed: 2 of 2 tests - 29 ms'. The console output lists the following tasks: `> Task :compileJava UP-TO-DATE`, `> Task :processResources NO-SOURCE`, `> Task :classes UP-TO-DATE`, `> Task :compileTestJava UP-TO-DATE`, `> Task :processTestResources NO-SOURCE`, `> Task :testClasses UP-TO-DATE`, and `> Task :test`. It also displays a deprecation warning about Gradle 9.0 compatibility and a success message: 'BUILD SUCCESSFUL in 1s'. The final line of the console output is '17:12:40: Execution finished ':test --tests "testedavid.PedidoPizzaTest"'.



The screenshot shows the 'Run' window of an IDE with the tab 'RealizaPedidoTest'. The 'Test Results' section indicates 'Tests passed: 2 of 2 tests - 36 ms'. The console output lists the following tasks: `> Task :compileJava UP-TO-DATE`, `> Task :processResources NO-SOURCE`, `> Task :classes UP-TO-DATE`, `> Task :compileTestJava UP-TO-DATE`, `> Task :processTestResources NO-SOURCE`, `> Task :testClasses UP-TO-DATE`, and `> Task :test`. It also displays a deprecation warning about Gradle 9.0 compatibility and a success message: 'BUILD SUCCESSFUL in 1s'. The final line of the console output is '17:10:51: Execution finished ':test --tests "testedavid.RealizaPedidoTest"'.

github: <https://github.com/SirDavos2/tp02-java>