Análise e Teste de Software

Testes Unitários e Cobertura

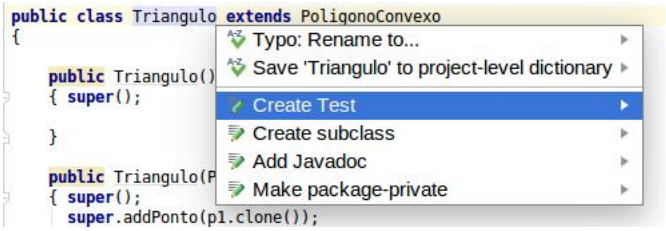
Universidade do Minho

2025/2026

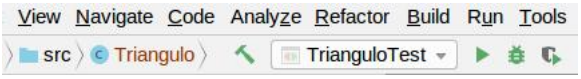
1 Introdução ao JUnit5

O JUnit5 é uma plataforma para definir e executar testes unitários. Estes testes pretendem executar pequenas partes do código e validar os resultados obtidos.

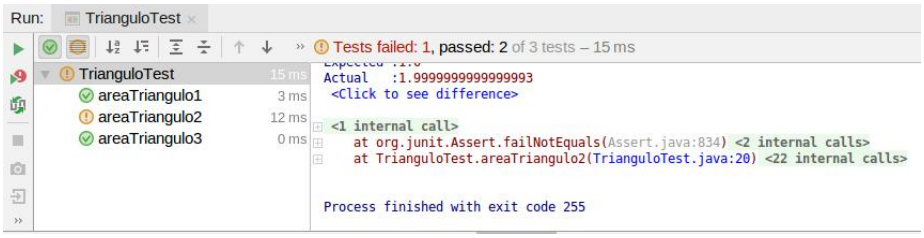
Ambientes de desenvolvimento modernos têm suporte à criação de novos testes e à execução dos mesmos. Por exemplo, o IntelliJ IDEA, usando o atalho `Alt+Enter` sobre o nome da classe, dá as seguintes opções:



permitindo criar a estrutura para novos testes usando as diversas plataformas de teste que suporta. Depois de escrever o corpo dos testes, é possível executá-los:



e observar os resultados:



2 Exercícios

1. Tendo em conta os ficheiros dados:

```
Poligono
├── NaoConvexoException.java
├── PoligonoConvexo.java
├── Poligono.java
├── Ponto.java
├── Retangulo.java
└── Triangulo.java
```

   (a) Teste a funcionalidade do código com testes unitários.

   (b) Este software tem um erro, escreva testes unitários de modo a o encontrar.

   (c) Quando encontrado o erro, não o corrija imeditamente. Escreva 3 testes unitários que falhem por causa do erro no código.

   (d) Corrija o erro no código, execute os testes unitários e verifique que todos passam.

Referência rápida do JUnit5

Anotações:

`@Test` – método de teste

`@BeforeEach` – método a executar antes de cada teste

`@AfterEach` – método a executar depois de cada teste

`@BeforeAll` – método a executar antes de qualquer outro método da classe

`@AfterAll` – método a executar depois de qualquer outro método da classe

`@Timeout(value = 500, unit = TimeUnit.MILLISECONDS)` – sinaliza que teste falha se passar do tempo especificado (neste caso em ms)

Verificação do resultado:

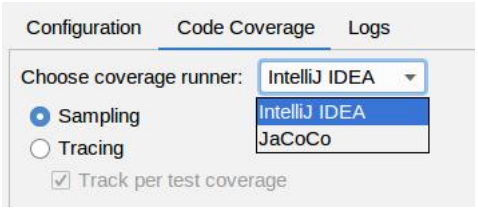
```
assertTrue(obtido, mensagem);

assertEquals(esperado, obtido, mensagem);

assertEquals(esperado, obtido, delta, mensagem);

assertSame(esperado, obtido, mensagem);

assertTimeout(duracaoMaxima, executavel, mensagem);

assertThrows(excepcaoEsperada, executavel, mensagem);

...
```

(ver documentação em `https://junit.org/junit5/docs/current/user-guide/#writing-tests-assertions` ou Javadoc respectivo)

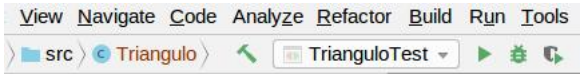
3 Analise de Cobertura de Código

Usando certas ferramentas é possível descobrir quais partes do código são executadas e quantas vezes o são, quando se executa testes unitários. Estas ferramentas são úteis para decidir se é necessário acrescentar testes ao software a ser testado, principalmente quando vemos certas partes do código que são críticas e não têm nenhum teste que as executam.

Certos IDEs, como por exemplo o IntelliJ IDEA[1], já têm a funcionalidade de análise da cobertura de código pelos testes, mas existem também ferramentas próprias, e.g., o JaCoCo[2] , e que IDEs também conseguem executar.



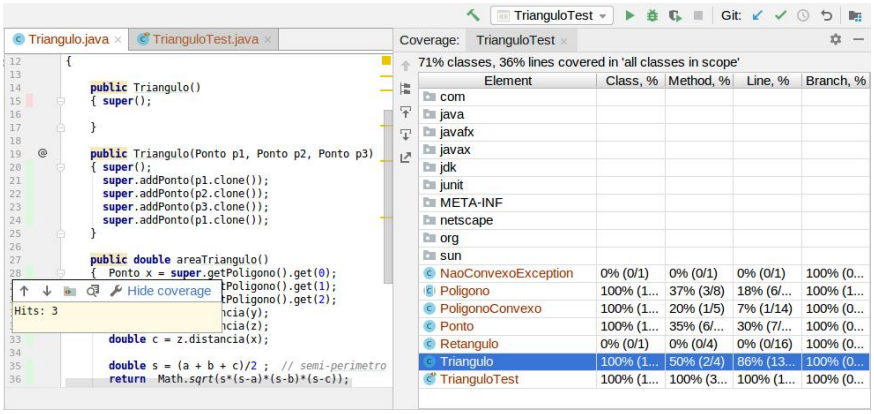
Depois de configurar o método de análise de cobertura, é necessário executar o código especificamente para esse efeito. Na figura abaixo (demonstrando o IntelliJ IDEA), o botão mais à direita tem esse objectivo.



Também pode-se executar indo ao menu `Run > Run ... with Coverage`.

Executando no IntelliJ IDEA, obtemos resultados do género:

[1] https://www.jetbrains.com/help/idea/code-coverage.html
[2] http://www.eclemma.org/jacoco/



que permitem ter uma ideia da cobertura de código (à direita na figura), e quais as linhas que foram executadas bem como o número de vezes que o foram (à esquerda na figura). Estes resultados podem também ser extraídos para um relatório (página HTML) usando o menu `Run > Generate Coverage Report`.

4 Exercícios

1. Tendo em conta os ficheiros dados:

```
Poligono
├── NaoConvexoException.java
├── PoligonoConvexo.java
├── Poligono.java
├── Ponto.java
├── Retangulo.java
└── Triangulo.java
```

   (a) Adicione testes de forma a que todo o código não trivial esteja coberto por testes.

2. Tendo em conta o projeto que desenvolveu na unidade curricular de **Programação Orientada a Objetos**:

   (a) Teste a funcionalidade do código com alguns testes unitários.

   (b) Utilize os testes unitários desenvolvidos para tentar localizar erros no projeto.

   (c) Analise a cobertura dos testes unitários.

   (d) Desenvolva mais testes unitários por forma a maximizar a cobertura.

   (e) Note-se que nem sempre é fácil de escrever testes unitários para blocos de código específicos! Por exemplo, se um método fizer leitura e

escrita de ficheiros, torna-se complicado ter um teste unitário que consiga lidar com isso. Analise o seu projeto e conclua se possui algum método que se encaixa nesta descrição.