# Probas unitarias

## 1.1 Introdución

Nesta parte da unidade didáctica preténdense os seguintes obxectivos:

- Definir o procedemento e os casos de proba de métodos Java e utilizar o contorno de desenvolvemento libre para executar os casos de probas con JUnit.
- Documentar a proba.

## 1.2 Probas unitarias

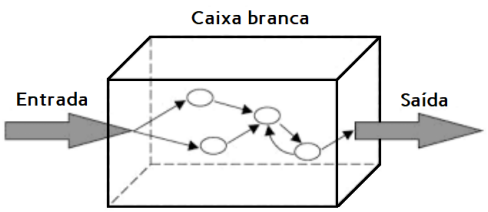
### Técnicas de deseño de casos de proba

O deseño de casos de proba está limitado pola imposibilidade de probar exhaustivamente o software. Por exemplo, de querer probar todos os valores que se poden sumar nun programa que suma dous números enteiros de dúas cifras (do 0 ao 99), deberiamos probar 10000 combinacións distintas (variacións con repetición de 100 elementos tomados de 2 en 2 = $100^2$) e aínda teriamos que probar todas as posibilidades de erro ao introducir datos (como teclear unha letra no canto dun número).

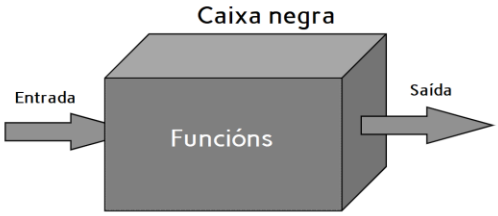
As técnicas de deseño de casos de proba teñen como obxectivo conseguir unha confianza aceptable en que se detectarán os defectos existentes, xa que a seguridade total só pode obterse da proba exhaustiva, que non é practicable sen consumir unha cantidade excesiva de recursos. Toda a disciplina de probas debe moverse nun equilibrio entre a dispoñibilidade de recursos e a confianza que achegan os casos para descubrir os defectos existentes.

Tendo en conta que non se poden facer probas exhaustivas, a idea fundamental subxacente para a elección de **casos de proba** consiste en elixir algúns deles que, polas súas características, se consideran **representativos** do resto. A dificultade desta idea é saber elixir os casos que se deben executar xa que unha elección puramente aleatoria non proporciona demasiada confianza en detectar os erros presentes. Existen tres enfoques principais para o deseño de casos, non excluíntes entre si e que se poden combinar para conseguir unha detección de defectos máis eficaz:

- Enfoque estrutural ou de **caixa branca** tamén chamado enfoque de caixa de cristal. Fíxase na implementación do programa para elixir os casos de proba.



- Enfoque funcional ou de **caixa negra**: Consiste en estudar a especificación das funcións, as súas entradas e saídas.

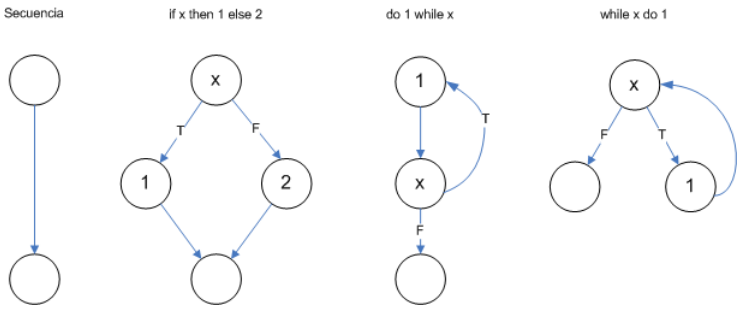



- Enfoque **aleatorio**: Utiliza modelos, moitas veces estatísticos, que representen as posibles entradas ao programa para crear a partir delas os casos de proba.

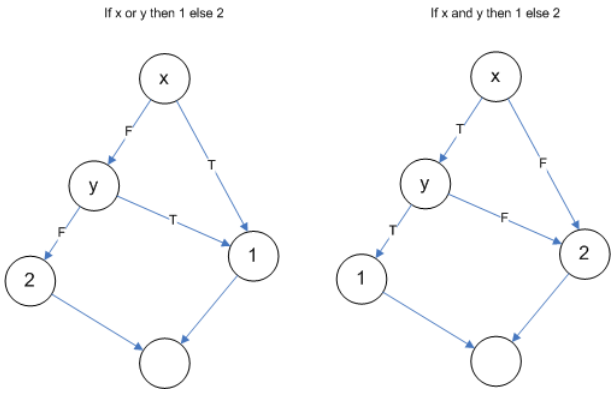
### Probas unitarias estruturais

Para traballar coas técnicas estruturais imos realizar **grafos de fluxo** dos programas ou funcións a probar. Isto non é estritamente necesario pero debuxalos axuda a comprender o funcionamento das técnicas de proba de caixa branca. Os grafos que se usarán serán **grafos fortemente conexos**, é dicir, sempre existe un camiño entre calquera par de nodos que se elixan e, para facer que o nodo primeiro e o último estean directamente conectados, engadirase un arco ficticio que os una.

Grafos básicos de fluxo:




Grafos para condicións múltiples:

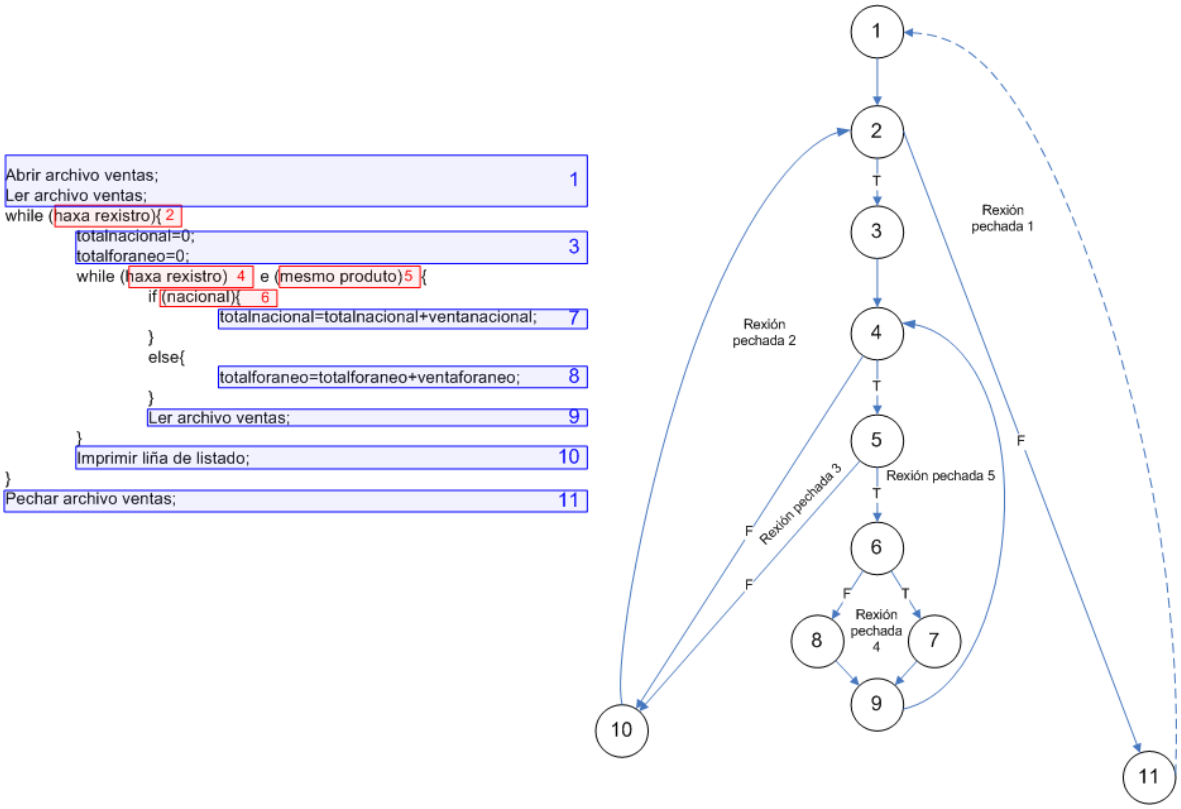


## Criterios de cobertura lóxica de Myers

O deseño de casos ten que basearse na elección de **camiños importantes** que ofrezan unha seguridade aceptable en descubrir defectos. Utilízanse os chamados criterios de cobertura lóxica. Os criterios de cobertura lóxica de Myers móstranse ordenados de menor a maior esixencia, e por tanto, custo económico:

- Cobertura de **sentenzas**: xerar os casos de proba necesarios para que cada sentenza se execute, polo menos, unha vez.
- Cobertura de **decisións**: xerar casos para que cada decisión teña, polo menos unha vez, un resultado verdadeiro e, polo menos unha vez, un falso.
- Cobertura de **condicións**: xerar os casos de proba necesarios para que cada condición de cada decisión adopte o valor verdadeiro polo menos unha vez e o falso polo menos unha vez. Por exemplo, a decisión: if ((a==3) || (b==2)) ten dúas condicións: (a==3) e (b==2).
- Criterio de **decisión/condición**: consiste en esixir o criterio de cobertura de condicións obrigando a que se cumpra tamén o criterio de decisións.
- Criterio de **condición múltiple**: No caso de que se considere que a avaliación das condicións de cada decisión non se realiza de forma simultánea, pódese considerar que cada decisión multicondicional se descompón en varias decisións unicondicionais.
- Criterio de **cobertura de camiños**: Débese executar cada un dos posibles camiños do programa polo menos unha vez. Defínese camiño como unha secuencia de sentenzas encadeadas desde a sentenza inicial do programa até a sentenza final. O número de camiños, mesmo nun programa pequeno, pode ser impracticable para as probas. Para reducir o número de camiños a probar, pode utilizarse a complexidade ciclomática de McCabe.

Exemplo de grafo de fluxo fortemente conexo dun anaco de pseudocódigo:




## Complexidade ciclomática de McCabe

A complexidade ciclomática é unha métrica que nos indica o número de camiños independentes que ten un grafo de fluxo. McCabe definiu como un bo criterio de proba o probar un número de camiños independentes igual ao da métrica. Un camiño independente é calquera camiño que introduce, polo menos, un novo conxunto de sentenzas de proceso ou unha condición, respecto dos camiños existentes. En termos do diagrama de fluxo, un camiño independente está constituído polo menos por unha aresta que non fose percorrida nos camiños xa definidos anteriormente. Na identificación dos distintos camiños débese ter en conta que cada novo camiño debe ter o mínimo número de sentenzas novas ou condicións novas respecto dos que xa existen. Desta maneira téntase que o proceso de depuración sexa máis sinxelo.

A **complexidade de McCabe V(G)** pódese calcular das seguintes tres maneiras, a partir dun grafo de fluxo G fortemente conexo:

- V(G) = a – n + 2
  - a: número de arcos ou arestas sen contar o que une o primeiro nodo có último
  - n: número de nodos

- V(G) = r
  - r: número de rexións pechadas do grafo contando a que forma o arco que une o primeiro nodo có último
- V(G) = c + 1
  - c: número de nodos de condición

Por exemplo, a complexidade ciclomática de McCabe para o grafo de fluxo do anaco de pseudocódigo anterior é 5, é dicir, é o valor obtido por calquera das tres fórmulas anteriores:

- V(G) = 14 – 11 + 2 = 5
- V(G) = 5
- V(G) = 4 + 1

Unha vez calculada a complexidade ciclomática debemos elixir tantos camiños independentes como o valor da complexidade. Para elixir estes camiños, comezamos definindo un camiño inicial e, a continuación, imos creando novos camiños variando o mínimo posible o camiño inicial. Para executar un camiño pode ser necesaria a súa concatenación con algún outro.

Para o exemplo anterior teriamos 5 camiños que se representan mediante o número dos nodos do grafo que recorren e os bucles mediante puntos suspensivos despois do nodo de control do bucle:

- 1-2-11
- 1-2-3-4-10-2-...
- 1-2-3-4-5-10-2-...
- 1-2-3-4-5-6-7-9-4-...
- 1-2-3-4-5-6-8-9-4-...

### Probas unitarias funcionais

Chegaría cunha proba de caixa branca para considerar probado un anaco de código?. A resposta é non, e demóstrase co seguinte código:

```
if ((x+y+z)/3==x)
   then print("X, Y, Z son iguales")
else print("X, Y, Z no son iguales")
```

Hai dous camiños posibles que se recorren cos valores x=5, y=5, z=5 e x=2, y=3, e z=7, que confirman a validez do código, pero cos valores x=4, y=5, e z=3 fallaría o código. Do que se deduce, que se necesitan outro tipo de probas como as probas funcionais para complementar as probas estruturais.

As probas funcionais ou de caixa negra céntranse no estudo da especificación do software, da análise das funcións que debe realizar, das entradas e das saídas. As probas funcionais exhaustivas tamén adoitan ser impracticables polo que existen distintas técnicas de deseño de casos de caixa negra.

### Clases de equivalencia

Cada caso de equivalencia debe cubrir o máximo número de entradas. Debe tratarse o dominio de valores de entrada dividido nun número finito de clases de equivalencia que cumpran que a proba dun valor representativo dunha clase permita supor “razoablemente” que o resultado obtido (se existen defectos ou non) será o mesmo que o obtido probando calquera outro valor da clase. O método para deseñar os casos consiste en identificar as clases de equivalencia e crear os casos de proba correspondentes.

Imos ver algunhas regras que nos axudan a identificar as clases de equivalencia tendo en conta as restricións dos datos que poden entrar ao programa:

- De especificar un rango de valores para os datos de entrada, como por exemplo, "o número estará comprendido entre 1 e 49", crearase unha clase válida: 1<= número <= 49 e dúas clases non válidas: número < 1 e número > 49.
- De especificar un número de valores para os datos de entrada, como por exemplo, "código de 2 a 4 caracteres", crearase unha clase válida: 2<=número de caracteres do código <=4, e dúas clases non válidas: menos de 2 caracteres e máis de 4 caracteres.
- Nunha situación do tipo "debe ser" ou booleana como por exemplo, "o primeiro carácter debe ser unha letra", identificarase unha clase válida: é unha letra e outra non válida: non é unha letra.
- De especificar un conxunto de valores admitidos que o programa trata de forma diferente, crearase unha clase para cada valor válido e outra clase para os non válidos. Por exemplo, se temos tres tipos de inmobles: pisos, chalés e locais comerciais, faremos unha clase de equivalencia por cada valor e unha non válida que representa calquera outro caso como, por exemplo, praza de garaxe.
- En calquera caso, de sospeitar que certos elementos dunha clase non se tratan igual que o resto da mesma, deben dividirse en clases de equivalencia menores.

Para crear os casos de proba séguense os pasos seguintes:

- Asignar un valor único a cada clase de equivalencia.
- Escribir casos de proba que cubran todas as clases de equivalencia válidas non incorporadas nos anteriores casos de proba.

- Escribir un caso de proba para cada clase non válida ata que estean cubertas todas as clases non válidas. Isto faise así xa que se introducimos varias clases de equivalencia non válidas xuntas, poida que a detección dun dos erros, faga que xa non se comprobe o resto.

Exemplo: Unha aplicación bancaria na que o operador proporciona un código de área (número de 3 díxitos que non empeza nin por 0 nin por 1), un nome para identificar a operación (6 caracteres) e unha orde que disparará unha serie de funcións bancarias ("cheque", "depósito", "pago factura" ou "retirada de fondos"). Todas as clases numeradas son:

| Entrada | Clases válidas | Clases inválidas |
|---|---|---|
| Código área | (1) 200 <=código <=999 | (2) código < 200<br>(3) código > 999 |
| Nome para identificar operación | (4) 6 caracteres | (5) menos de 6 caracteres<br>(6) más de 6 caracteres |
| Orde | (7) “cheque”<br>(8) “depósito”<br>(9) “pago factura”<br>(10) “retirada fondos” | (11) “divisas” |

Os casos de proba, supoñendo que a orde de entrada dos datos é: código-nome-orde son os seguintes:

- Casos válidos:

| Código | Nome | orde | Clases |
|---|---|---|---|
| 200 | Nómina | cheque | (1) (4) (7) |
| 200 | Nómina | depósito | (1) (4) (8) |
| 200 | Nómina | pago factura | (1) (4) (9) |
| 200 | Nómina | retirada fondos | (1) (4) (10) |

- Casos non válidos:

| Código | Nome | orde | Clases |
|---|---|---|---|
| 180 | Nómina | cheque | (2) |
| 1032 | Nómina | cheque | (3) |
| 200 | Nómin | cheque | (5) |
| 200 | Nóminas | cheque | (6) |
| 200 | Nómina | divisas | (11) |

### Análise de valores límite (AVL)

A experiencia constata que os casos de proba que exploran as condicións límite dun programa producen un mellor resultado para a detección de defectos. Podemos definir as condicións límite para as entradas, como as situacións que se encontran directamente arriba, abaixo e nas marxes das clases de equivalencia e dentro do rango de valores permitidos para o tipo desas entradas. Podemos definir as condicións límite para as saídas, como as situacións que provocan valores límite nas posibles saídas. É recomendable

utilizar o enxeño para considerar todos os aspectos e matices, ás veces sutís, para a aplicación da AVL. Algunhas regras para xerar os casos de proba:

- Se para unha entrada se especifica un rango de valores, débense xerar casos válidos para os extremos do rango e casos non válidos para situacións xusto máis aló dos extremos. Por exemplo a clase de equivalencia: "-1.0 <= valor <= 1.0", casos válidos: -1.0 e 1.0, casos non válidos: -1.01 e 1.01, no caso no que se admitan 2 decimais.
- Se para unha entrada se especifica un número de valores, hai que escribir casos para os números máximo, mínimo, un máis do máximo e un menos do mínimo. Por exemplo: "o ficheiro de entrada terá de 1 a 250 rexistros", casos válidos: 1 e 250 rexistros, casos non válidos: 251 e 0 rexistros.
- De especificar rango de valores para a saída, tentarán escribirse casos para tratar os límites na saída. Por exemplo: "o programa pode mostrar de 1 a 4 listaxes", casos válidos: 1 e 4 listaxes, casos non válidos: 0 e 5 listaxes.
- De especificar número de valores para a saída, hai que tentar escribir casos para tratar os limites na saída. Por exemplo: "desconto máximo será o 50%, o mínimo será o 6%", casos válidos: descontos do 50% e o 6%, casos non válidos: descontos do 5.99%, e 50.01% se o desconto é un número real.
- Se a entrada ou saída é un conxunto ordenado (por exemplo, unha táboa, un arquivo secuencial,...), os casos deben concentrarse no primeiro e no último elemento.

### Conxectura de erros:

A idea básica desta técnica consiste en enumerar unha lista de erros posibles que poden cometer os programadores ou de situacións propensas a certos erros e xerar casos de proba en base a esa lista. Esta técnica tamén se denominou xeración de casos (ou valores) especiais, xa que non se obteñen en base a outros métodos senón mediante a intuición ou a experiencia. Algúns valores a ter en conta para os casos especiais poderían ser os seguintes:

- valor 0 é propenso a xerar erros tanto na saída como na entrada.
- En situacións nas que se introduce un número variable de valores, como por exemplo unha lista, convén centrarse no caso de non introducir ningún valor e un só valor. Tamén pode ser interesante que todos os valores sexan iguais.
- É recomendable supor que o programador puidese interpretar mal algo nas especificacións.

- Tamén interesa imaxinar as accións que o usuario realiza ao introducir unha entrada, mesmo coma se quixese sabotar o programa. Poderíase comprobar como se comporta o programa se os valores de entrada están fóra dos rangos de valores límites permitidos para ese tipo de variable. Por exemplo, se unha variable de entrada é de tipo *int*, deberíase comprobar o que ocorre se o valor de entrada está fóra do rango de valores permitido para un *int* ou mesmo se ten decimais ou é unha letra. Poderíase comprobar que na entrada non vai camuflado código perigoso. Por exemplo, comprobar a posible inxección de código nunha entrada a unha base de datos.
- Completar as probas de caixa branca e de caixa negra para o caso de bucles. Procurar que un bucle se execute 0 veces, 1 vez ou máis veces. De coñecer o número máximo de iteracións do bucle (n), habería que executar o bucle 0, 1, n-1 e n veces. Se hai bucles aniñados, os casos de proba aumentarían de forma exponencial, polo que se recomenda comezar probando o bucle máis interior mantendo os exteriores coas iteracións mínimas e ir creando casos de proba cara o exterior do aniñamento.

### Probas unitarias aleatorias

Nas probas aleatorias simúlase a entrada habitual do programa creando datos para introducir nel que sigan a secuencia e frecuencia coas que poderían aparecer na práctica diaria, de maneira repetitiva (próbanse moitas combinacións). Para iso utilízanse habitualmente ferramentas denominadas xeradores automáticos de casos de proba.

### Enfoque recomendado para o deseño de casos

As distintas técnicas vistas para elaborar casos de proba representan aproximacións diferentes. O enfoque recomendado consiste na utilización conxunta das técnicas indicadas para lograr un nivel de probas “razoable”. Por exemplo:

- Elaborar casos de proba de caixa negra para as entradas e saídas utilizando clases de equivalencia, completar coa análise do valor límite e coa conxectura de erros para engadir novos casos non contemplados nas técnicas anteriores.
- Elaborar casos de proba de caixa branca baseándose nos camiños do código para completar os casos de proba de caixa negra.

## Junit

JUnit é un framework ou conxunto de clases Java que permite crear e executar probas automatizadas. Os casos de proba en programas Java quedan arquivados e poden volver executarse tantas veces como sexa necesario. Estes casos de proba permiten avaliar se a clase se comporta como se esperaba, é dicir, a partir dun valores de entrada, avalíase se o resultado obtido é o esperado. O desenvolvemento de JUnit iniciouse a finais de 1995 e, desde entón, a súa popularidade foi en aumento. JUnit foi escrito por Erich Gamma e Kent Beck .Trátase dun proxecto de código libre aloxado en GitHub. Hoxe en día, considérase o estándar de facto para probar aplicacións Java.

JUnit 5 é compatible coas versións 3 e 4 anteriores.

JUnit 5 creouse desde cero utilizando **Java 8**.

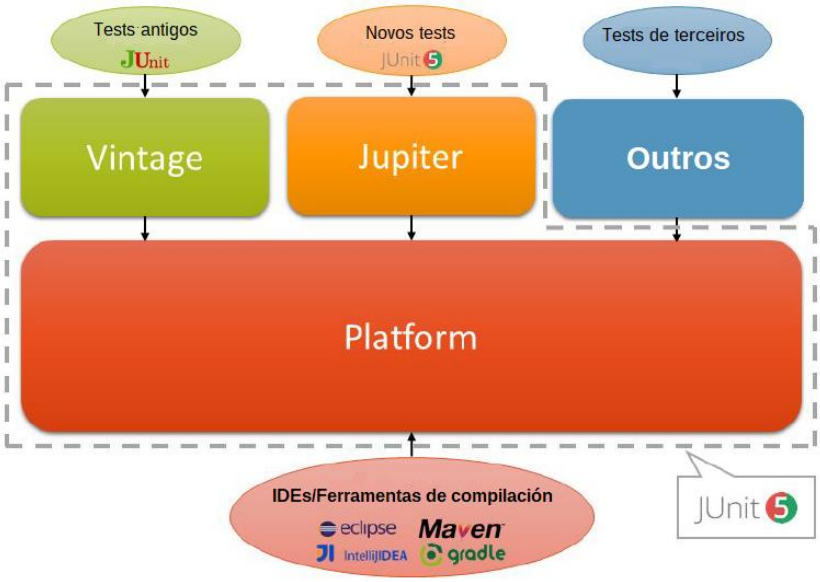
JUnit deseñouse para facer probas de unidade. Non obstante, pode usarse non só para probas unitarias senón tamén para outra clase de probas.

O código fonte de JUnit está aloxado en GitHub (https://github.com/junit-team/junit5/). As persoas que contribúen no desenvolvemento de JUnit non son só desenvolvedores senón tamén probadores, persoas que se dedican ao mantemento e comunicadores.

O sitio oficial de JUnit 5 é https://junit.org/junit5/.

Os IDEs como Netbeans e Eclipse contan con complementos para utilizar JUnit, permitindo que o programador se centre na proba e no resultado esperado e deixe ao IDE a creación das clases que permiten a proba.

Como se amosa na imaxe, o framework JUnit 5 componse de tres compoñentes principais chamados Platform, Jupiter e Vintage:




Arquitectura de JUnit 5

O primeiro compoñente de alto nivel chámase **Jupiter**. Proporciona o novo modelo de programación e extensión do framework JUnit 5.

No núcleo de JUnit 5 está o compoñente **Platform**. Este compoñente está orientado a converterse na base de calquera framework de probas executado na JVM. Proporciona mecanismos para executar probas de Jupiter, JUnit 4 herdado e tamén probas de terceiros.

O terceiro compoñente de JUnit 5 é o compoñente **Vintage**. Este compoñente permite executar probas JUnit herdadas.

Hai **tres tipos** de módulos:

- **APIs de probas**. Son os módulos que usan os desenvolvedores e probadores. Estes módulos proporcionan o modelo de programación para un motor de proba particular (por exemplo, junit-jupiter-api para realizar probas con JUnit 5 e junit para realizar probas con JUnit 4).
- **Motores de probas**. Estes módulos permiten executar un tipo de proba (probas Jupiter, probas herdadas JUnit 4 ou outras probas de Java) dentro da plataforma JUnit. Créanse herdando do motor Platform xeral (junit-platform-engine).
- **Lanzadores de probas**. Estes módulos proporcionan a capacidade de descubrir e executar probas dentro da plataforma JUnit para ferramentas de compilación externas e IDE. Esta API consúmena ferramentas como Maven, Gradle, IntelliJ, etc. usando o módulo junit-platform-launcher.

### Xerar probas en JUnit

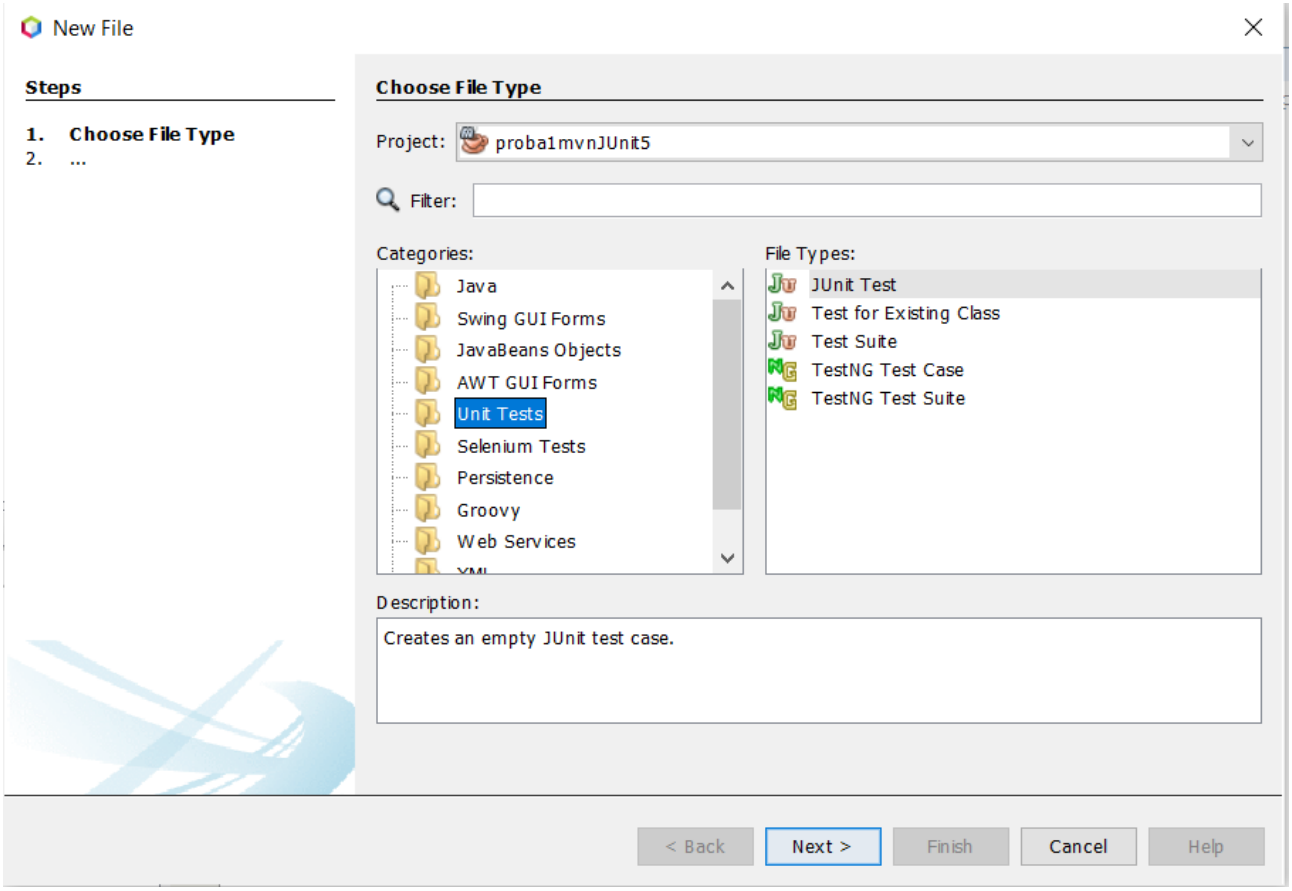
En Netbeans pódense xerar varios tipos de probas.

Para crear unha proba JUnit pódese seleccionar New File na opción File do menú principal. Despois selecciónase a categoría Unit Tests e o tipo de arquivo entre os 5 posibles (JUnit Test, Test for Existing Class, Test Suite, TestNG Test Case e TestNG Test Suite).

Un JUnit Test crea un caso de proba baleiro, Test for Existing Class crea un caso de proba para os métodos dunha clase, Test Suite crea probas para un paquete Java seleccionado. Os TestNG (Test Next Generation) introducen novas funcionalidades.

Dependendo do tipo de proba seleccionada, haberá que completar de forma diferente as seguintes ventás que aparecen.

En calquera caso aparece unha ventá diferente se a proba é para unha clase ou para un paquete.

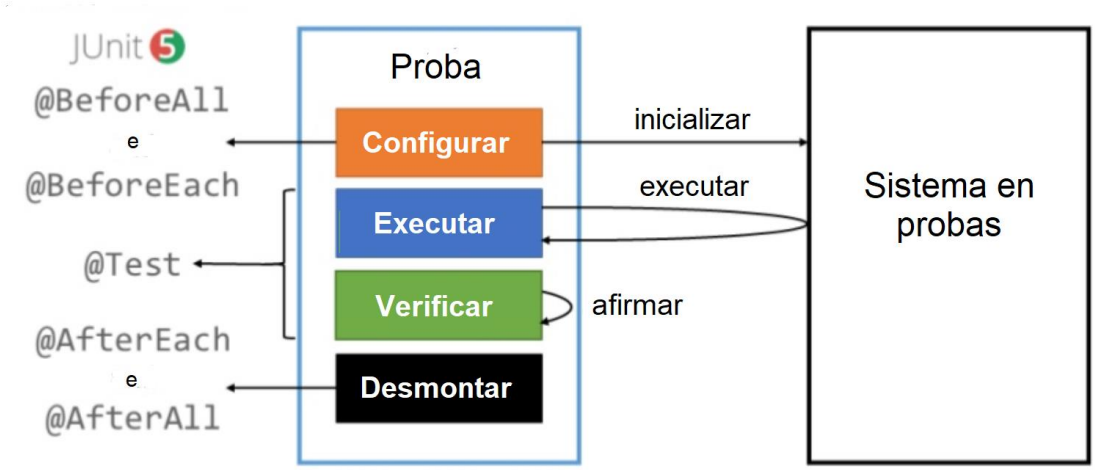



## Ciclo de vida das probas

Un caso de proba unitario componse de catro etapas:

1. **Configurar** (setup). Fase opcional. Fanse as inicializacións necesarias para a realización da proba.
2. **Executar** (exercise). A proba interactúa co sistema que se está a probar, obtendo algún resultado desta interacción.
3. **Verificar** (verify). O resultado obtido da interacción co sistema en probas compárase co valor esperado utilizando unha ou varias asercióǹs. Obtense un veredicto da proba.
4. **Desmontar** (teardown). Fase opcional. Faise o necesario para deixar o sistema en probas no seu estado inicial.

Na seguinte imaxe poden verse graficamente as catro etapas:




Para controlar as diferentes fases en JUnit 5 úsanse diferentes anotacións.

En JUnit 5 nin as clases de proba nin os métodos de proba necesitan ser públicos.

A anotación máis básica de JUnit é **@Test**, que marca os métodos que se deben executar como probas.

En Jupiter, todas estas anotacións están incluídas no paquete org.junit.jupiter.api.

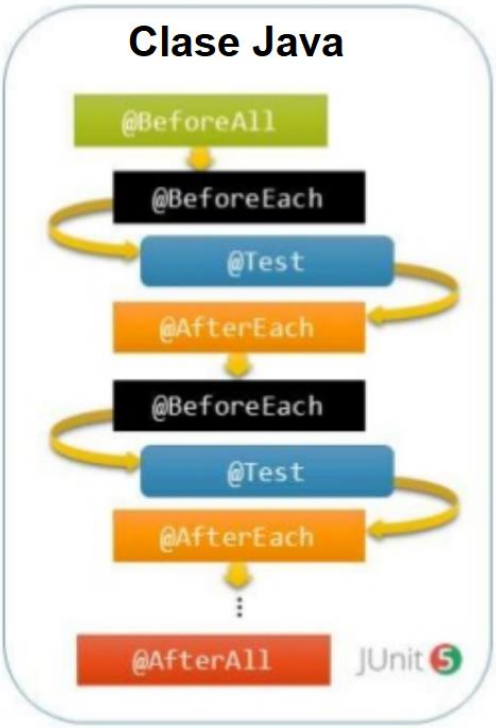
É posible executar código para configurar e rematar as probas. Hai catro anotacións para axudar a facelo:

- **@BeforeAll**. Execútase unha vez antes de todas probas e dos métodos marcados con @BeforeEach.
- **@BeforeEach**. Execútase antes de cada proba.
- **@AfterEach**. Execútase despois de cada proba.
- **@AfterAll**. Execútase unha vez despois de todas as probas e métodos marcados con @AfterEach.

A etapa de configuración lévase a cabo anotando os métodos con @BeforeAll e @BeforeEach.

A etapa de execución e verificación lévase a cabo utilizando as anotacións @Test nos métodos.

Finalmente, a desmontaxe lévase a cabo cos método anotados con @AfterEach e @AfterAll.



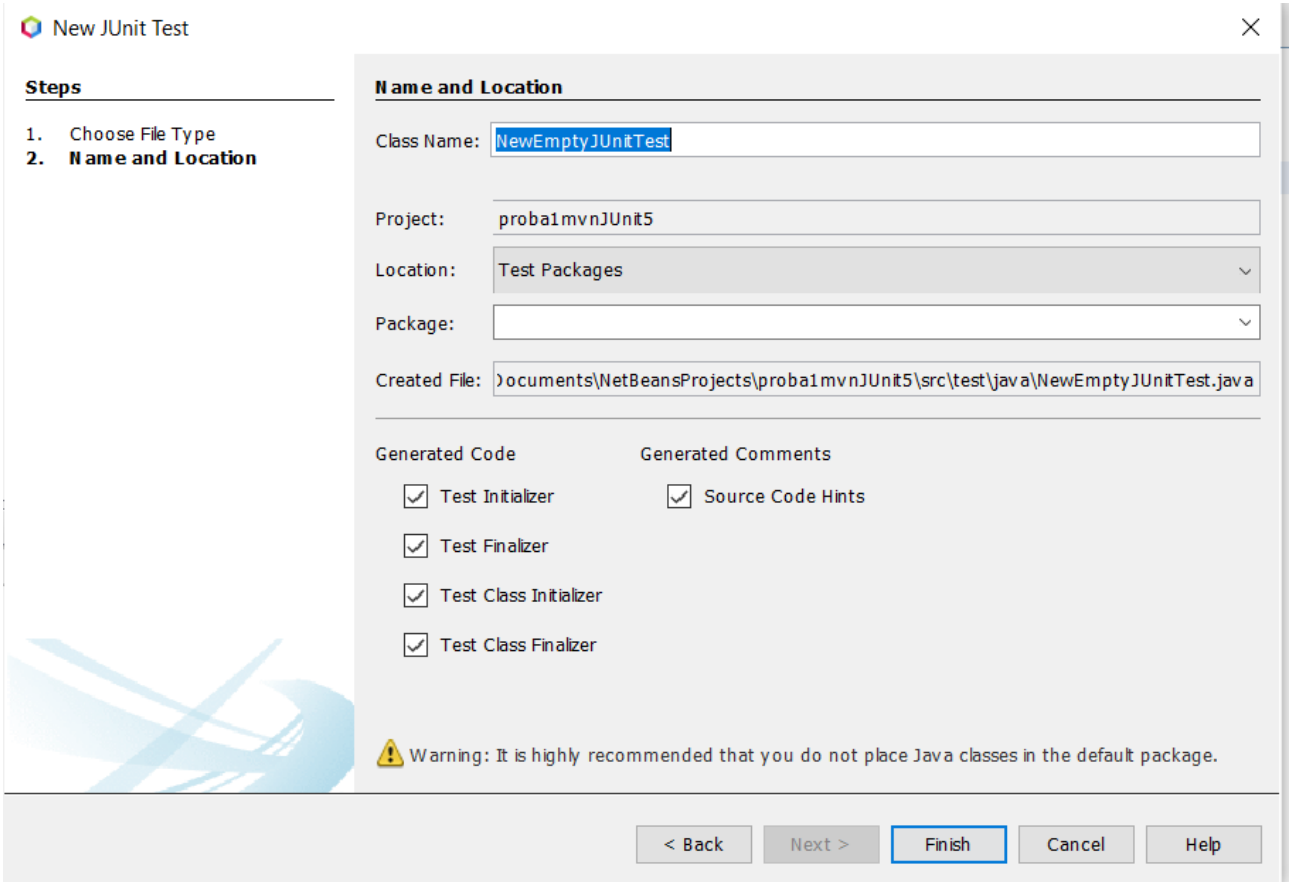

A anotación Jupiter **@Disabled** do paquete org.junit.jupiter.api pode usarse para saltar probas. Pode usarse a nivel de clase ou de método. Cando se executa unha clase de proba cun método ou unha clase coa anotación @Disabled, no resumo de execución o método debe contarse como saltado (skipped).

### Probas para unha clase

Cando se crean probas para unha clase, as características da xeración de código están divididas varios apartados:

- Apartado *Generated Code* para indicar se a proba terá métodos para executarse antes de iniciar a proba (*Test Initializer*), despois de finalizar a proba (*Test Finalizer*), antes de iniciar todas as probas (*Test Class Initializer*) ou antes de finalizar todas as probas (*Test Class Finalizer*).
- Apartado *Generated Comments* para indicar que as probas leven comentarios Javadoc e comentarios para suxerir como implementar os métodos de proba (*Source Code Hints).*

Déixase todo seleccionado e prémese en  OK.




Para probar o ciclo de vida de JUnit 5 en NetBeans imos crear un novo proxecto Java con Maven. Unha vez creado o proxecto, engadimos unha clase de proba (JUnit Test).

Se ao engadir a clase de proba os import non son de JUnit 5 (org.junit.jupiter...), imos ao cartafol Test Dependencies e borramos as dependencias de JUnit 4. Borramos a clase de proba e volvemos creala comprobando que os import son correctos.

O esqueleto que crea NetBeans para unha clase de proba JUnit 5 é o seguinte:

```
import org.junit.jupiter.api.AfterEach;
import org.junit.jupiter.api.AfterAll;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.BeforeAll;
import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.*;
public class CicloVidaJUnit5Test {

    public CicloVidaJUnit5Test() {
    }

    @BeforeAll
    public static void setUpClass() {
    }

    @AfterAll
    public static void tearDownClass() {
    }

    @BeforeEach
    public void setUp() {
    }
```

```
    @AfterEach
    public void tearDown() {
    }

    // TODO add test methods here.
    // The methods must be annotated with annotation @Test. For example:
    //
    // @Test
    // public void hello() {}
}
```

Para probar que o framework funciona correctamente poñemos código en todos os métodos como no seguinte exemplo:

```
import org.junit.jupiter.api.AfterEach;
import org.junit.jupiter.api.AfterAll;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.BeforeAll;
import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.*;

public class ProbaJUnit5Test {

    public ProbaJUnit5Test() {
    }

    @BeforeAll
    public static void setUpClass() {
        System.out.println("Antes de todos");
    }

    @AfterAll
    public static void tearDownClass() {
        System.out.println("Despois de todos");
    }

    @BeforeEach
    public void setUp() {
        System.out.println("Antes de cada un");
    }

    @AfterEach
    public void tearDown() {
        System.out.println("Despois de cada un");
    }

    @Test
    public void test1() {
        System.out.println("test 1");
        assertTrue(1==1);
    }

    @Test
    public void test2() {
        System.out.println("test2");
        assertTrue(2==2);
    }
}
```

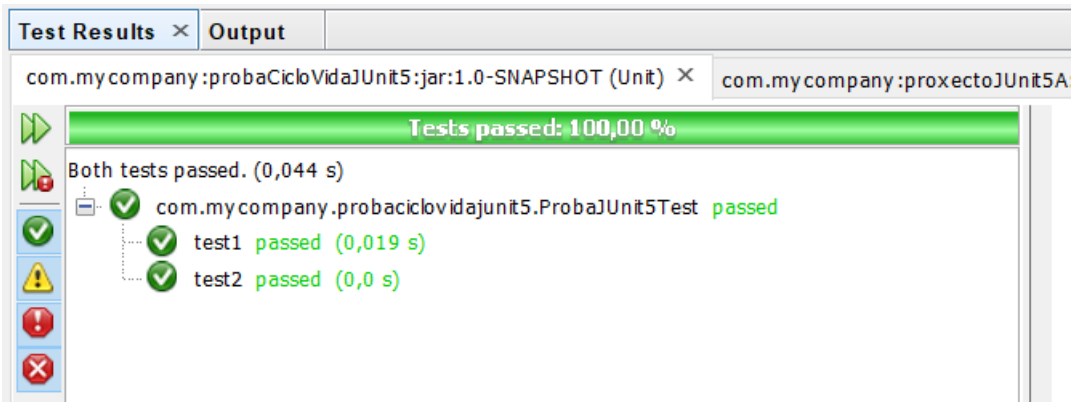
Para probar o test debemos premer no botón dereito e indicar *Test File*.

Na saída deben aparecer as mensaxes dos métodos que se executan antes e despois das probas xunto coas mensaxes das probas mesmas.

```
-------------------------------------------------------
 T E S T S
-------------------------------------------------------
Running com.mycompany.probaciclovidajunit5.ProbaJUnit5Test
Antes de todos
Antes de cada un
test 1
Despois de cada un
Antes de cada un
test2
Despois de cada un
Despois de todos
Tests run: 2, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.044 sec - in
com.mycompany.probaciclovidajunit5.ProbaJUnit5Test
Results :

Tests run: 2, Failures: 0, Errors: 0, Skipped: 0
------------------------------------------------------------------------
BUILD SUCCESS
------------------------------------------------------------------------
Total time:  5.594 s
Finished at: 2021-04-05T15:03:44+02:00
------------------------------------------------------------------------
```

O resultado do test ten que ser o seguinte:




No caso de que as mensaxes que deben aparecer antes e despois das probas non aparezan, é necesario facer un cambio no ficheiro **pom.xml** do proxecto. Un Project Object Model ou POM é a unidade fundamental de traballo en Maven. Trátase dun arquivo XML que contén información sobre o proxecto e os detalles de configuración utilizados por Maven para construír o proxecto. Para executar un proxecto Maven busca o POM no directorio actual, le o POM se o atopa obtendo a configuración necesaria e, posteriormente executa.

Un exemplo de pom.xml sería o seguinte:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <groupId>com.mycompany</groupId>
    <artifactId>probaCicloVidaJUnit5</artifactId>
    <version>1.0-SNAPSHOT</version>
    <packaging>jar</packaging>
```

```
    <dependencies>
        <dependency>
            <groupId>org.junit.jupiter</groupId>
            <artifactId>junit-jupiter-api</artifactId>
            <version>5.6.0</version>
            <scope>test</scope>
        </dependency>
    </dependencies>
    <properties>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
        <maven.compiler.source>11</maven.compiler.source>
        <maven.compiler.target>11</maven.compiler.target>
    </properties>
    <build>
    <plugins>
        <plugin>
            <artifactId>maven-surefire-plugin</artifactId>
            <version>3.0.0-M5</version>
            <executions>
                <execution>
                    <goals>
                        <goal>test</goal>
                    </goals>
                    <id>test</id>
                </execution>
            </executions>
            <configuration>
                <foo>bar</foo>
            </configuration>
        </plugin>
    </plugins>
</build>
</project>
```

É importante verificar que o plugin de Maven surefire está no ficheiro pom.xml.

Imos probar agora unha clase, a clase Division.java.

O código da clase é o seguinte:

```
public class Division {
   public float calcularDivision(float dividendo, float divisor) throws Excep-
tion {
       if (divisor==0){
           throw new Exception("Erro. O divisor non pode ser 0.");
       }
       float resultado=dividendo/divisor;
       return resultado;
   }
}
```

O código que xera NetBeans por defecto para a proba do método *calcularDivision()* contempla un só caso de proba (0/0=0). Pódese modificar este método e engadir novos casos de proba ou engadir outros métodos de proba. NetBeans suxire borrar as dúas últimas liñas de código da anotación *Test*. Pódense borrar ou comentar as liñas das anotacións agás a anotación *Test* e as liñas *import* correspondentes ás anotacións borradas, se é que non van a ser utilizadas.

O código xerado para as probas é o seguinte:

```
import org.junit.jupiter.api.AfterEach;
import org.junit.jupiter.api.AfterAll;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.BeforeAll;
import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.*;

public class DivisionTest {

    public DivisionTest() {
    }

    @BeforeAll
    public static void setUpClass() {
    }

    @AfterAll
    public static void tearDownClass() {
    }

    @BeforeEach
    public void setUp() {
    }

    @AfterEach
    public void tearDown() {
    }

    /**
     * Test of calcularDivision method, of class Division.
     * @throws java.lang.Exception
     */
    @Test
    public void testCalcularDivision() throws Exception {
        System.out.println("calcularDivision");
        float dividendo = 0.0F;
        float divisor = 0.0F;
        Division instancia = new Division();
        float resultadoEsperado = 0.0F;
        float resultado = instancia.calcularDivision(dividendo, divisor);
        assertEquals(resultadoEsperado, resultado, 0.0);
        fail("The test case is a prototype");
    }
}
```

### Estrutura da proba

As anotacións e métodos que aparecen por defecto son:

- A anotación *@BeforeAll* marca ao método *setUpClass()* para ser executado antes de empezar a proba de clase, é dicir, execútase unha soa vez e antes de empezar a execución dos métodos de proba. Pódese utilizar por exemplo para crear unha conexión cunha base de datos.
- A anotación *@AfterAll* marca ao método *tearDownClass()* para ser executado ao finalizar a proba de clase. Pódese utilizar por exemplo para pechar a conexión coa base de datos realizada antes.

- A anotación *@Before* marca ao método *setUp()* para ser executado antes da execución de cada un dos métodos de proba, é dicir, execútase tantas veces como métodos de proba existan. Utilízase para inicializar recursos, variables de clase ou atributos que sexan iguais para tódalas probas.
- A anotación *@After* marca ao método *tearDown()* para executarse xusto despois da execución de cada un dos métodos de proba.
- A anotación *@Test* marca cada método de proba.
- método *assertEquals*. Este método afirma que o primeiro argumento (resultado esperado) é igual ao segundo (resultado obtido). Se os dous argumentos son reais, pode ter un terceiro argumento chamado valor delta, que é un número real igual á máxima diferenza en valor absoluto entre o valor esperado e o actual para que a afirmación sexa un éxito: `Math.abs(esperado-obtido)<delta`

Na proba para o método *calcularDivision()* utilizada de exemplo, pódese modificar o test para poder comprobar que a división entre 1 e 3 dá como resultado 0.33 cun valor delta de 1E-2. Se consideramos que o método *calcularDivision(1,3)* devolve 0.333, os valores esperados 0.34 e 0.33 serían equiparables ao valor real e o valor 0.32 non, xa que abs(0.333-0.33) < 0.01, abs(0.333-0.34) < 0.01 e abs(0.333-0.32) > 0.01.

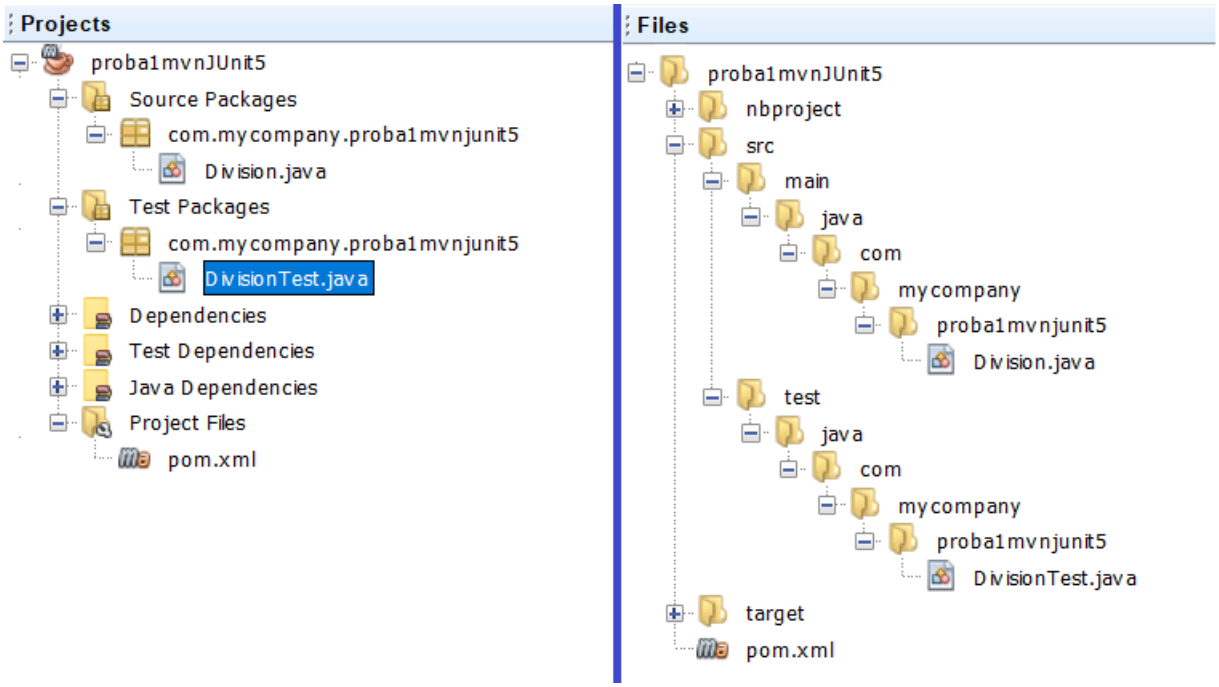
O código do test podería ser:

```
import org.junit.jupiter.api.AfterEach;
import org.junit.jupiter.api.AfterAll;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.BeforeAll;
import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.*;
public class DivisionTest {

    public DivisionTest() {
    }
    /**
     * Test of calcularDivision method, of class Division.
     */
    @Test
    public void testCalcularDivision() throws Exception {
        System.out.println("Caso: 1/3=0.33 con valor delta 1E-2");
        Division instancia = new Division();
        float resultado = instancia.calcularDivision(1F,3F);
        assertEquals(0.33, resultado, 1E-2);
    }
}
```

### Cartafol para as probas

Nas ventás de arquivos e de proxectos poden verse a estrutura lóxica e física das carpetas nas que se gardan as probas JUnit. Pódense agregar outras carpetas de proba no cadro de diálogo de propiedades do proxecto pero tendo en conta que os arquivos de proba e os fontes non poden estar no mesmo cartafol.




## Executar a proba

Para executar a proba dunha clase, pódese elixir unha das dúas posibilidades seguintes:

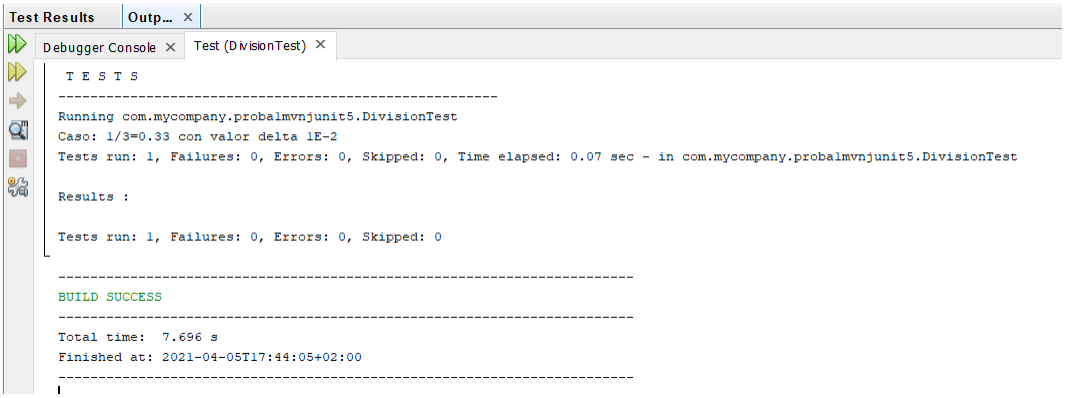
- Selecciónase a clase na ventá de proxectos ou na de arquivos, clic dereito e elíxese *Test File* ou prémese Ctrl-F6.
- Selecciónase a proba da clase e elíxese no menú principal *Run -> Test File* ou prémese Mayús-F6.

Pasos para executar un caso de proba:

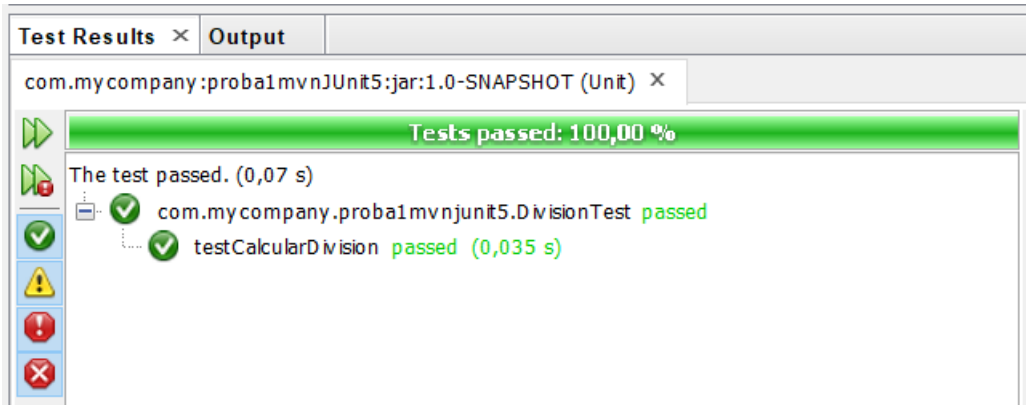
- Execútase a proba que contén ese caso de proba.
- Na ventá de resultados faise clic dereito sobre o método e elíxese *Run Again*.

## Ventás de saída e resultados

A ventá de saída reflicte o detalle do proceso de execución das probas en formato texto.




A ventá de resultados contén un resumo dos casos de proba superados e non superados e unha descrición deles en formato gráfico. Ao pasar o rato por riba da descrición dun método de proba, aparece nun recadro a saída correspondente a ese método.



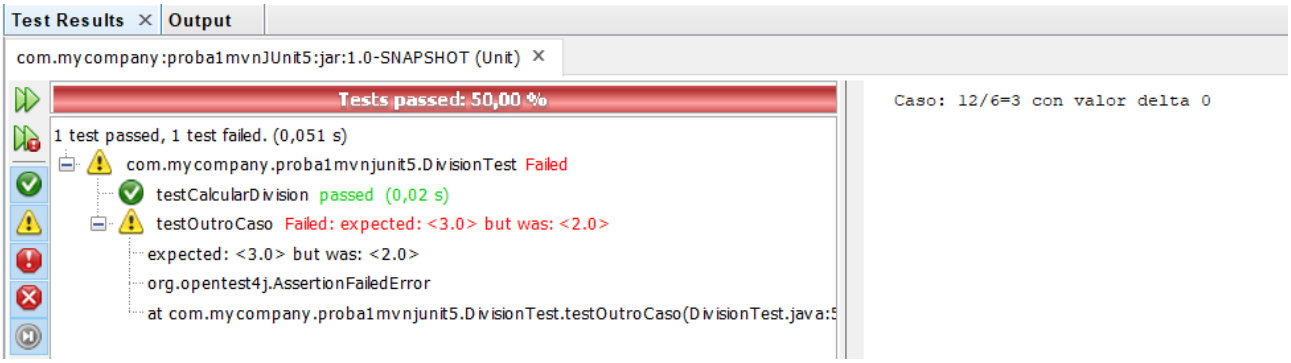

Algunhas das iconas que se poden utilizar na ventá de resultados son:

| |
|---|
| volver a executar a proba |
| volver a executar as probas non superadas |
| mostrar en detalle as probas superadas |
| mostrar en detalle as probas non superadas |
| mostrar en detalle os erros das probas non superadas |
| moverse polos métodos non superados cara arriba ou cara abaixo |

Estas ventás cambian se non se supera algunha proba. Por exemplo, ao executar un método de proba que espera un resultado 3 para o caso de proba 12/6 como o seguinte:

```
@Test
    public void testOutroCaso() throws Exception {
        System.out.println("Caso: 12/6=3 con valor delta 0");
        float dividendo=12.0F;
        float divisor=6.0F;
        Division instancia=new Division();
        float resultadoEsperado=3F;
        float resultado=instancia.calcularDivision(dividendo,divisor);
        assertEquals(resultadoEsperado,resultado,0.00);
    }
```

Aparecerá na ventá de resultados o detalle sobre o caso de proba non superado.



### Afirmacións

A clase *Assert* permite comprobar se a saída do método que se está probando concorda cos valores esperados. Unha afirmación é unha declaración booleana que xeralmente se usa para razonar sobre a corrección do software.

Pódese ver a sintaxe completa dos métodos asertos (afirmacións) que se poden usar para as probas en JUnit 5 en: https://junit.org/junit5/docs/current/api/org.junit.jupiter.api/org/junit/jupiter/api/Assertions.html.

Desde un punto de vista técnico, a afirmación componse de tres partes:

1. Atopamos o **valor esperado** que ven do que chamamos oráculo de proba. Un **oráculo de proba** é unha fonte confiable de resultados esperados, por exemplo, a especificación do sistema.
2. Atopamos o **resultado real**, que se obtén de realizar a proba no sistema en probas.
3. **Compárase o valor esperado co resultado real**. Esta comparación pode facerse de moitas maneiras. Como resultado obtemos un veredicto de proba que, ao final, vai determinar se a proba tivo éxito ou non.


Todas as afirmacións de Junit Jupiter son métodos estáticos da clase Assertions situada no paquete org.junit.jupiter.

Os diferentes tipos básicos de afirmacións en Jupiter son os seguintes:

- **fail**. Unha proba falla cunha mensaxe determinada ou excepción.
- **assertTrue**. Afirma que unha condición subministrada é verdadeira.
- **assertFalse**. Afirma que unha condición subministrada é falsa.
- **assertNull**. Afirma que un obxecto subministrado é null.
- **assertNotNull**. Afirma que un obxecto subministrado non é null.
- **assertEquals**. Afirma que dous obxectos subministrados son iguais. É un método con sobrecarga para cada tipo en java e que permite comprobar se a chamada a

un método devolve un valor esperado. No caso de valores reais ten un terceiro argumento para indicar o valor delta ou número real igual á máxima diferenza en valor absoluto entre o valor esperado e o actual para que a afirmación sexa un éxito.

- **assertArrayEquals**. Afirma que dous arrays subministrados son iguais.
- **assertIterableEquals**. Afirma que dous obxetos iterables son iguais.
- **assertLinesMatch**. Afirma que dúas listas de Strings son iguais.
- **assertNotEquals**. Afirma que dos obxectos subministrados non son iguais.
- **assertSame**. Afirma que dous obxectos subministrados son o mesmo, comparados con ==.
- **assertNotSame**. Afirma que dous obxectos son diferentes, comparados con !=.

Para cada unha das afirmacións anteriores, pode indicarse unha mensaxe opcional de fallo. Esta mensaxe sempre é o último parámetro no método de afirmación.

### Anotacións

As anotacións aportan información sobre un programa e poden ser usadas polo compilador (por exemplo: *@Override* para informarlle que se está sobrescribindo un método, *@Deprecated* para indicarlle que está en desuso, *@SupressWarnings* para indicarlle que non avise de *warnings* ou advertencias), por ferramentas de software que as poden procesar (por exemplo para xerar código ou arquivos xml) ou para ser procesadas en tempo de execución. As anotacións poden aplicarse a declaracións de clases, campos, métodos e outros elementos dun programa.

As anotacións máis importantes nunha proba son *@Disabled* que serve para desactivar un test e colócase xusto antes de *@Test*, e *@Test*.

A anotación *@Test* serve para anotar unha proba.

A anotación @Timeout permite indicar que un test debería fallar se o tempo de execución excede a duración indicada.