TEMA 3-A03: Probas unitarias

Índice

A03. Probas unitarias

1.1 Introdución

Na actividade que nos ocupa 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 Actividade

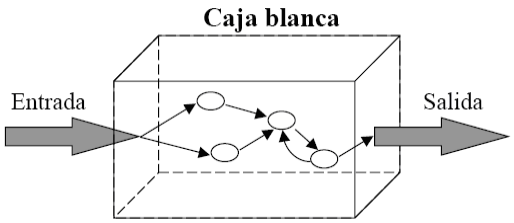
Técnicas de deseño de casos de proba

O deseño de casos de proba está limitado pola imposibilidade de probar exhaustivamente o software. Por exemplo, de querer probar tódolos 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 elevado a 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.

Xa que non se poden facer probas exhaustivas, a idea fundamental para o deseño de casos de proba consiste en elixir algúns deles que, polas súas características, considéranse 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:

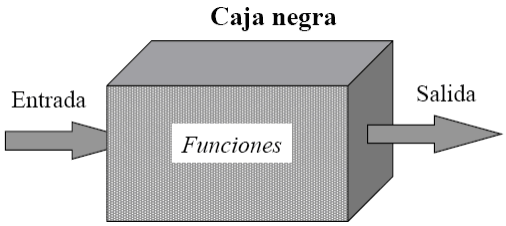
- Enfoque estrutural ou de caixa branca tamén chamado enfoque de caixa de cristal: Fíxase na implementación do programa para elixir os casos de proba[1].



- Enfoque funcional ou de caixa negra: Consiste en estudar a especificación das funcións, as súas entradas e saídas[2].

[1] Imaxe extraída de http://osl2.uca.es/wikihaskell/index.php/Pruebas_Unitarias_para_Haskell

[2] Imaxe extraída de http://osl2.uca.es/wikihaskell/index.php/Pruebas_Unitarias_para_Haskell

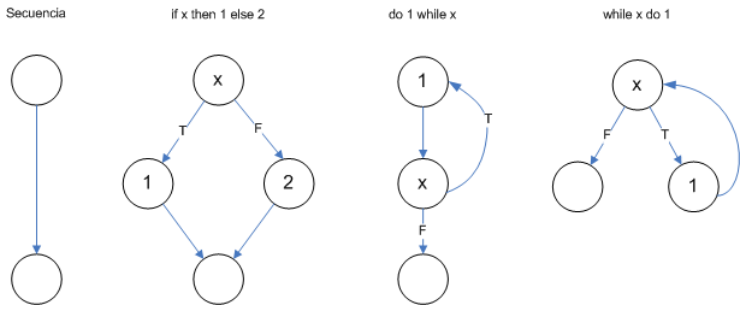


- Enfoque aleatorio: Utiliza modelos, moitas veces estatísticos, que representen as posibles entradas ao programa para crear a partir delas os casos de proba.

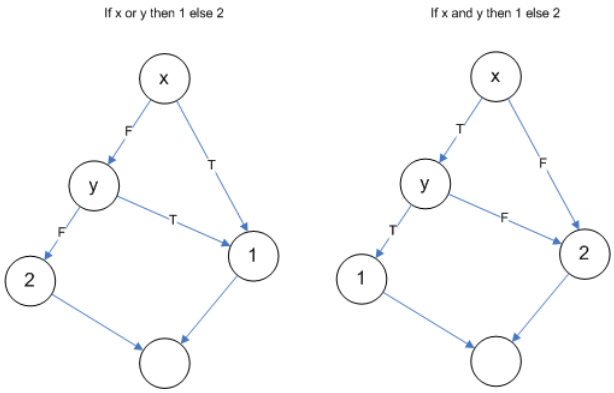
Probas unitarias estruturais

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 subsanar 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:



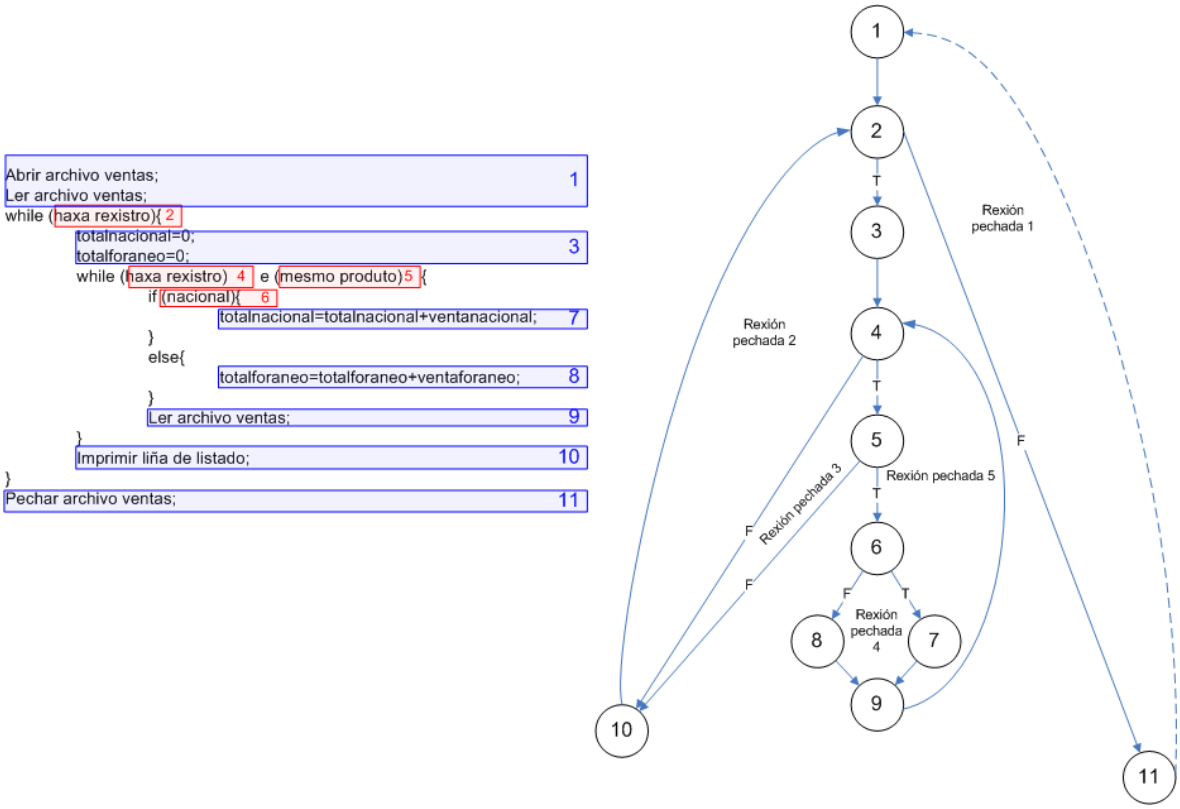
Criterios de cobertura lóxica de Myers

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 descomponse en varias decisións unicondicionales.
- 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-...

Tarefa 1. Representar grafos de fluxo, calcular a complexidade ciclomática de McCabe e obter os camiños.

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=4, y=3, e z=5, que confirman a validez do código, pero cos valores x=2, 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 permite 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 non válida. 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 (3) código > 999
Nome para identificar operación	(4) 6 caracteres	(5) menos de 6 caracteres (6) más de 6 caracteres
Orde	(7) “cheque” (8) “depósito” (9) “pago factura” (10) “retirada fondos”	(11) “divisas”

Os casos de proba, supoñendo que a orde de entrada dos datos é: código-nome-orden 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 no 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 do AVL. Algunhas regras para xerar os casos de proba:

- Se para unha entrada especifícase 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 especifícase 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 dita 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:

- O 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.

Tarefa 2. Definir clases de equivalencia, realizar análise dos valores límite e conxectura de erros.

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 de ditas técnicas 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 co 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.

Tarefa 3. Elaborar casos de proba.

Junit

JUnit é un framework ou conxunto de clases Java que permiten ao programador construír e executar automaticamente casos de proba para métodos dunha clase Java. Os casos de proba quedan reflectidos en programas Java que quedan arquivados e poden volver a executarse tantas veces como sexa necesario. Estes casos de proba permiten avaliar se a clase se comporta como se esperaba, é dicir, a partir duns valores de entrada, avalíase se o resultado obtido é o esperado. JUnit foi escrito por Erich Gamma e Kent Beck e é un produto

de código aberto distribuído baixo unha licenza Common Public License - v 1.0. A páxina oficial de JUnit é : https://junit.org/junit5/

Os IDE como NetBeans, JBuilder 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.

Neste documento utilizarase JUnit5 para o IDE NetBeans 8.2 e faranse as probas deste apartado sobre o sinxelo proxecto *proyecto_division* composto das clases Division.java e Main.java seguintes:

Xerar probas en JUnit

En NetBeans pódense xerar tres tipos de probas: para probar os métodos dunha clase, un conxunto de probas para probar un conxunto de clases dun paquete ou un caso de probas JUnit baleiro.

Para crear unha proba JUnit, pódese seleccionar *Archivo Nuevo* na opción *Archivo* do menú principal, seleccionar a categoría *Unit Tests* e o tipo de arquivo de entre os tres posibles: *JUnit Test* que crea un caso de proba baleiro, *Test for Existing Class* que crea un caso de proba para os métodos dunha clase e *Test Suite* que crea probas para un paquete Java seleccionado. Dependendo do tipo de proba seleccionada, haberá que completar de forma diferente as seguintes ventás que aparecen.

Outra maneira de poder crear as probas de clase é seleccionar a clase ou package sobre o que se queren facer as probas e elixir no menú principal *Herramientas->Create/Update Tests*.

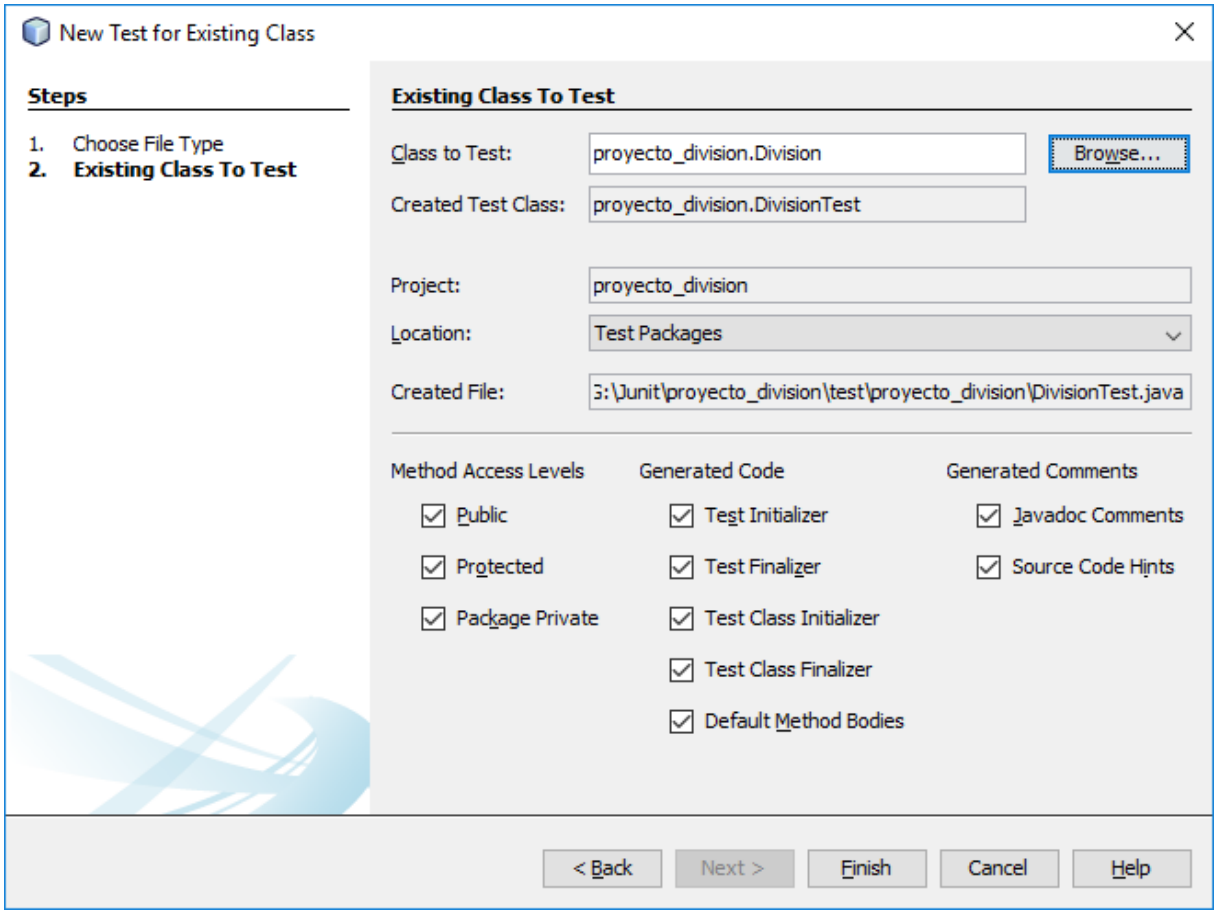
En calquera caso aparece unha ventá diferente se a proba é para unha clase ou para un paquete. Nos apartados seguintes só se contemplarán os casos de proba para unha clase e para un package.

Para unha clase

De elixir no menú principal *File->New File-> Unit Tests->Test for Existing Class*, aparecerá a ventá *Existing Class To Test* na que hai que indicar: o nome da clase de proba (recoméndase deixar o nome por defecto: nome da clase a probar seguido de *Test*), a localización onde se gardará a proba (recoméndase deixar o valor por defecto) e as características da xeración de código dividida en tres apartados:

- Apartado *Method Access Levels* para indicar o nivel de acceso aos métodos de proba.
- 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*) ou código exemplo para unha proba (*Default Method Bodies*).
- 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 Finish.



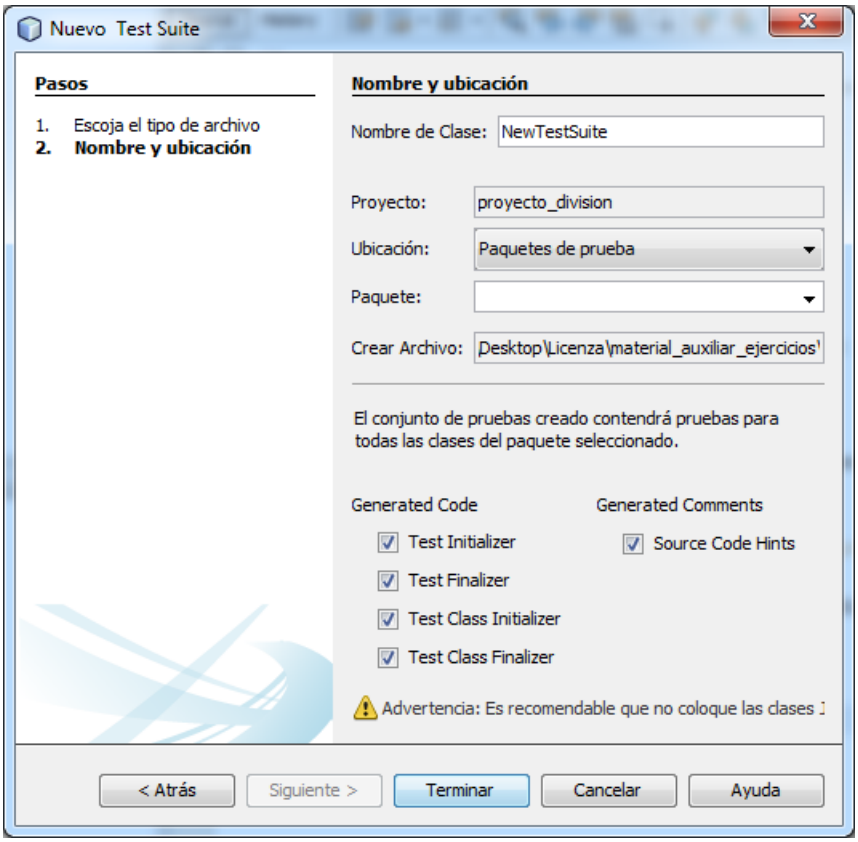
O código que xera NetBeans por defecto para a proba do método *calcularDivision()* contempla un só caso de proba (0/0=0). Pódese modificar este método e engadir novos casos de proba ou engadir outros métodos de proba. NetBeans suxire borrar as dúas últimas liñas de código da anotación *Test*. Pódense 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.

```
/*
 * To change this license header, choose License Headers in Project Properties.
 * To change this template file, choose Tools | Templates
 * and open the template in the editor.
 */
package proyecto_division;

import org.junit.After;
import org.junit.AfterClass;
import org.junit.Before;
import org.junit.BeforeClass;
import org.junit.Test;
import static org.junit.Assert.*;

/**
 *
 * @author Profesor
 */
public class DivisionTest {

    public DivisionTest() {
    }

    @BeforeClass
    public static void setUpClass() {
    }

    @AfterClass
    public static void tearDownClass() {
    }

    @Before
    public void setUp() {
    }

    @After
```

```
    public void tearDown() {
    }

    /**
     * Test of calcularDivision method, of class Division.
     */
    @Test
    public void testCalcularDivision() throws Exception {
        System.out.println("calcularDivision");
        float dividendo = 0.0F;
        float divisor = 0.0F;
        Division instance = new Division();
        float expResult = 0.0F;
        float result = instance.calcularDivision(dividendo, divisor);
        assertEquals(expResult, result, 0.0);
        // TODO review the generated test code and remove the default call to fail.
        fail("The test case is a prototype.");
    }

}
```

Para un package

De elixir no menú principal Archivo->*Archivo Nuevo->Unit Tests->Test Suite*, aparecerá a ventá *New Test Suite*, onde se dará nome á clase de proba e elixirase o paquete a probar, premendo ao final en *Finish*.



Na ventá de edición pódese ver o código que por defecto xera NetBeans para a proba. Na liña 20 pódese ver que as clases de proba que xa existían (como neste exemplo a clase *DivisionTest*) están incluídas no conxunto de probas. Pódense engadir novas probas.

```
/*
 * To change this license header, choose License Headers in Project Properties.
 * To change this template file, choose Tools | Templates
 * and open the template in the editor.
 */
package proyecto_division;

import org.junit.After;
import org.junit.AfterClass;
import org.junit.Before;
import org.junit.BeforeClass;
import org.junit.runner.RunWith;
import org.junit.runners.Suite;
```

```
/**
 *
 * @author Profesor
 */
@RunWith(Suite.class)
@Suite.SuiteClasses({proyecto_division.DivisionTest.class})
public class NewTestSuite {

    @BeforeClass
    public static void setUpClass() throws Exception {
    }

    @AfterClass
    public static void tearDownClass() throws Exception {
    }

    @Before
    public void setUp() throws Exception {
    }

    @After
    public void tearDown() throws Exception {
    }

}
```

Estrutura da proba

As anotacións e métodos que aparecen por defecto son:

- A anotación *@BeforeClass* 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 *@AfterClass* 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.
- O 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
```

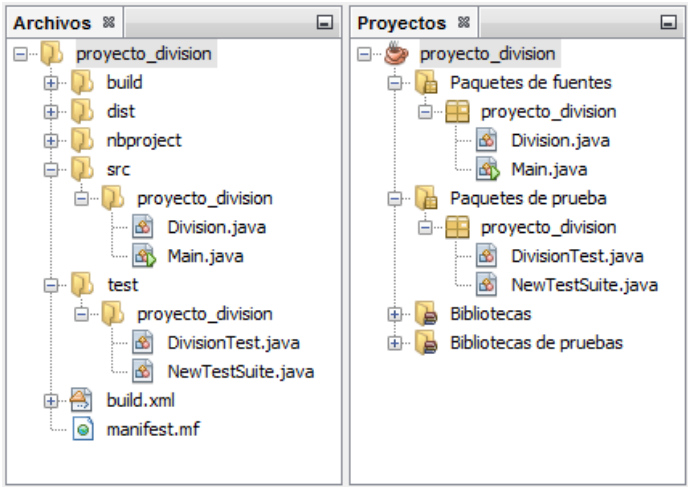
Na proba para o método *calcularDivision()* utilizada de exemplo, pódese modificar o test para poder comprobar que a división entre 1 e 3 dá como resultado 0.33 cun valor delta de 1E-2. Se consideramos que o método *calcularDivision(1,3)* devolve 0.333, os valores esperados 0.34 e 0.33 serían equiparables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

```
package proyecto_division;

import org.junit.Test;
import static org.junit.Assert.*;

public class DivisionTest {
```

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

    }
}
```

Carpeta 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 na mesma carpeta.



Executar a proba

Pasos para executar a proba dun proxecto enteiro:

- Selecciónase calquera nodo ou arquivo do proxecto na ventá de proxectos ou na de arquivos e elíxese no menú principal *Ejecutar->Probar Project(nome_do_proxecto)* ou prémese Alt-F6.
- O IDE executa tódolos métodos de proba do proxecto. De querer executar un subconxunto das probas do proxecto ou executar as probas nun orden específico, debe crearse unha *Test Suite* que especifique as probas a executar.

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

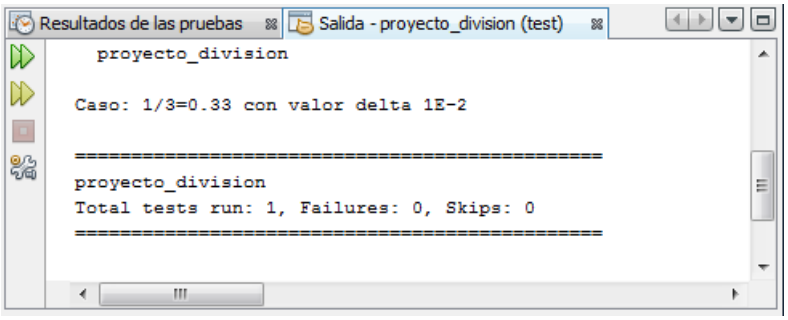
- Selecciónase a clase na ventá de proxectos ou na de arquivos, clic dereito e elíxese *Probar archivo* ou prémese Ctrl-F6.
- Selecciónase a proba da clase e elíxese no menú principal *Ejecutar -> Ejecutar archivo* ou prémese Mayús-F6.

Pasos para executar un caso de proba:

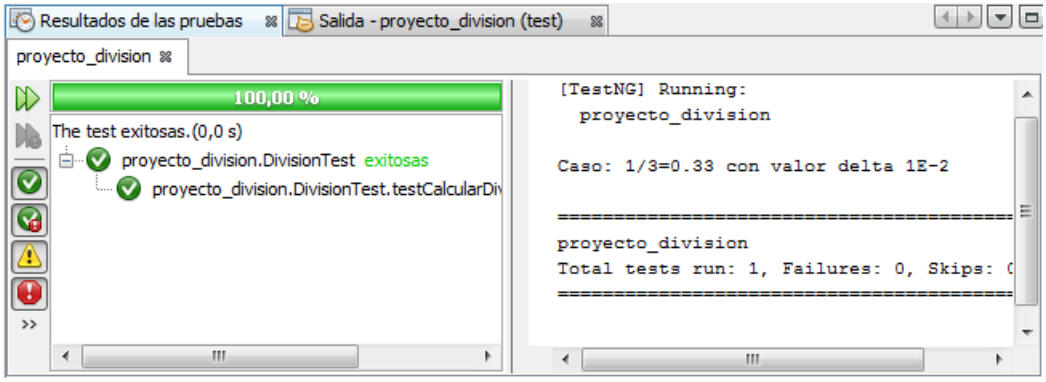
- Execútase a proba que contén ese caso de proba.
- Na ventá de resultados faise clic dereito sobre o método e elíxese *Run Again*.

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 ten dúas zonas. A zona da esquerda 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. A zona da dereita contén a saída textual.



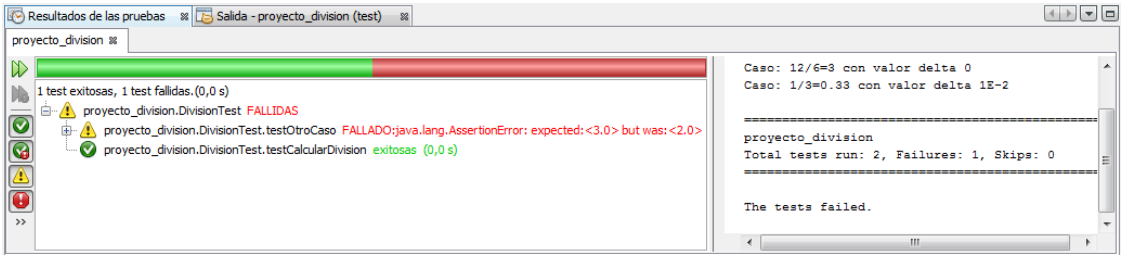
Algunhas das iconas que se poden utilizar na ventá de resultados son:

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 testOtroCaso() throws Exception {
    System.out.println("Caso: 12/6=3 con valor delta 0");
    float dividendo = 12.0F;
    float divisor = 6.0F;
    Division instance = new Division();
    float expResult = 3F;
    float result = instance.calcularDivision(dividendo, divisor);
    assertEquals(expResult, result, 0.00);
}
```

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



Asertos

A clase *Assert* permite comprobar se a saída do método que se está probando concorda cos valores esperados. Pódese ver a sintaxe completa dos métodos asertos (afirmacións) que se poden usar para as probas en https://junit.org/junit5/docs/5.3.0/api/org/junit/jupiter/api/Assertions.html

Unha breve descrición deles é:

- *void assertEquals(valor esperado, valor actual)* 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.
- *void fail()* para cando se espera que o programa falle. Utilízase cando a proba indica que hai un erro ou cando se espera que o método que se está probando chame a unha excepción.
- *void assertTrue(boolean)* a proba é un éxito se a expresión booleana é certa.
- *void assertFalse(boolean)* a proba é un éxito se a expresión booleana é falsa.
- *void assertNull(Object)* a proba é un éxito se o obxecto é nulo.
- *void assertNotNull(Object)* a proba é un éxito se o obxecto non é nulo.
- *void assertSame(Object,Object)* a proba é un éxito se os dous obxectos son o mesmo.
- *void assertNotSame(Object,Object*) a proba é un éxito se os dous obxectos non son mesmo.

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 *@Ignore* que serve para desactivar un test e colócase xusto antes de *@Test*, e *@Test*.

A anotación *@Test* serve para anotar unha proba. Pode ter dous parámetros opcionais *expected* e *timeout*. O primeiro define a clase de excepción que se espera que lance a proba para que sexa superada e o segundo define os milisegundos que como máximo debe durar a execución para que a proba sexa superada.

Exemplo con *timeout*: o seguinte test non se superará xa que a execución da proba supera os 100 milisegundos:

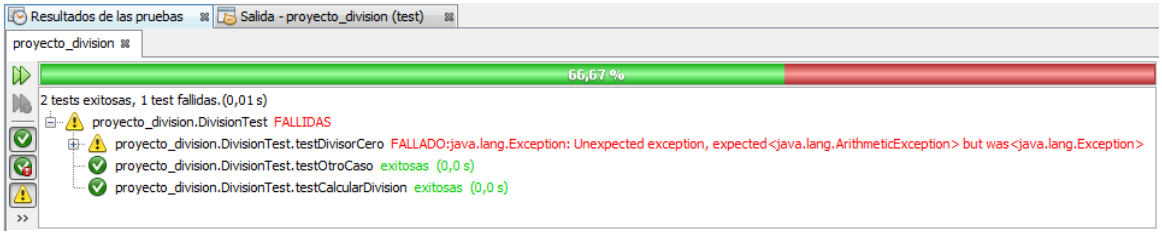
```
@Test(timeout = 100)
public void infinity() {
        while (true);
}
```

Exemplo con *expected*: se na proba da clase *Division* se engade un novo caso de proba

para a división entre 0 contemplando unha excepción de tipo aritmético.

```
@Test(expected = ArithmeticException.class)
public void testDivisorCero() throws Exception {
        System.out.println("Caso: Divisor 0");
        Division instance = new Division();
        float resultado = instance.calcularDivision(10F, 0F);
}
```

Isto provocará que a proba non sexa exitosa xa que a división por cero está contemplada no código da clase mediante unha excepción creada polo programador.



Unha posible solución sería contemplar unha excepción *java.lang.Exception*.

```
@Test(expected = java.lang.Exception.class)
public void testDivisorCero() throws Exception {
        System.out.println("Caso: Divisor 0");
        Division instance = new Division();
        float resultado = instance.calcularDivision(10F, 0F);
}
```

Tarefa 4. Xerar e executar probas en JUnit e documentar incidencias

1.3 Tarefas

Breve explicación sobre os métodos Java de proxectos NetBeans, anexionados a esta actividade, e utilizados na maior parte das tarefas:

- O método *calcularDivision* do proxecto *proyecto_division* recibe un dividendo e un divisor de tipo float e devolve o resultado da división tamén de tipo float sempre que o divisor non sexa 0, en cuxo caso xera unha excepción.

```
package proyecto_division;

public class Division {
    public float calcularDivision(float dividendo, float divisor) throws Exception {
        if (divisor == 0) {
            throw (new Exception("Error. El divisor no puede ser 0."));
        }
        float resultado = dividendo / divisor;
        return resultado;
    }
}
```

- O método *factorial* do proxecto *proyecto_factorial* recibe un número n de tipo byte e devolve o seu factorial de tipo float agás no caso de que sexa negativo, en cuxo caso xera unha excepción. O factorial dun número n é o produto de tódolos números menores que el ata o número 2. Casos especiais do factorial son factorial(0)=1 e factorial(1)=1.

```
package proyecto_factorial;

public class Factorial {
    public float factorial(byte n) throws Exception {
        if (n < 0) {
            throw new Exception("Error. El número tiene que ser >=0");
        }
        float resultado = 1;
        for (int i = 2; i <= n; i++) {
            resultado *= i;
        }
        return resultado;
    }
}
```

- O método *busca* do proxecto *proyecto_arrays* recibe un caracter *c* e un array de caracteres *v* de 10 elementos como máximo ordenados de forma ascendente. Devolve o valor booleano *true* ou *false* segundo encontre o caracter no array ou non. A busca é dicotómica, é dicir, a primeira busca faise tendo en conta todo o array pero nas seguintes só se ten en conta un segmento del obtido mediante o cálculo do índice metade do segmento e a comparación de c có elemento almacenado nese índice; se coincide, finaliza a busca e encontrouse o caracter no array; se c é menor, o seguinte segmento será a primeira metade do actual; se c é maior, o seguinte segmento será a segunda metade do actual. Se este proceso finaliza cun segmento nulo e non se encontrou o caracter é que non existe.

```
package buscarcaracter;

public class OperacionsArrays {

    public boolean busca(char c, char[] v) {
        int a, z, m;
        a = 0;
        z = v.length - 1;
        boolean resultado=false;
        while (a <= z && resultado==false) {
            m = (a + z) / 2;
            if (v[m] == c) {
                resultado=true;
            }
            else
            {
                if (v[m] < c) {
                    a = m + 1;
                }
                else{
                    z = m - 1;
                }
            }
        }
        return resultado;
    }
}
```

- O método *obtenerAcronimo* do proxecto *proyecto_acronimo* recibe unha cadea de caracteres e retorna unha cadea co acrónimo correspondente. O acrónimo está formado polo primeiro caracter de cada palabra seguidos dun punto cando o caracter é diferente de espazo en branco.

```
package proyecto_acronimos;

public class Acronimos {

    public String obtenerAcronimo(String cadena){
        String resultado="";
        char caracter;
        int n=cadena.length();
        for(int i=0;i<n;i++){
            caracter=cadena.charAt(i);
            if(caracter!=' '){
                if (i==0){
                    resultado=resultado+caracter+'.';
                }
                else{
                    if(cadena.charAt(i-1)==' '){
                        resultado=resultado+caracter+'.';
                    }
                }
            }
        }
        return resultado;
    }
}
```

1.3.1 Tarefa 1. Representar grafos de fluxo, calcular a complexidade ciclomática de McCabe e obter camiños

A tarefa consiste en representar o grafo de fluxo, calcular a complexidade de McCabe e detallar os camiños para os métodos seguintes:

- Tarefa 1.1. Método *calcularDivision* do proxecto *proyecto_division*.
- Tarefa 1.2. Método *factorial* do proxecto *proyecto_factorial*.
- Tarefa 1.3. Método *busca* do proxecto *proyecto_arrays*.
- Tarefa 1.4. Método *obtenerAcronimo* do proxecto *proyecto_acronimo*.

1.3.2 Tarefa 2. Definir clases de equivalencia, realizar análise dos valores límite e conxectura de erros

A tarefa consiste en definir as clases de equivalencia, realizar a análise dos valores límite e realizar conxectura de erros para:

- Tarefa 2.1. O método *calcularDivision* do proxecto *proyecto_division*.
- Tarefa 2.2. O método *factorial* do proxecto *proyecto_factorial*.
- Tarefa 2.3. O método *busca* do proxecto *proyecto_arrays*.
- Tarefa 2.4. O método *obtenerAcronimo* do proxecto *proyecto_acronimo*.

1.3.3 Tarefa 3. Elaborar casos de proba

A tarefa consiste en elaborar casos de proba para:

- Tarefa 3.1. O método *calcularDivision* do proxecto *proyecto_division*.
- Tarefa 3.2. O método *factorial* do proxecto *proyecto_factorial*.
- Tarefa 3.3. O método *busca* do proxecto *proyecto_arrays*.
- Tarefa 3.4. O método *obtenerAcronimo* do proxecto *proyecto_acronimo*.

1.3.4 Tarefa 4. Xerar e executar probas en JUnit e documentar incidencias

A tarefa consiste en xerar probas en JUnit, executalas e documentar incidencias, para os métodos:

- Tarefa 4.1. O método *calcularDivision* do proxecto *proyecto_division*.
- Tarefa 4.2. O método *factorial* do proxecto *proyecto_factorial*.
- Tarefa 4.3. O método *busca* do proxecto *proyecto_arrays*.
- Tarefa 4.4. O método *obtenerAcronimo* do proxecto *proyecto_acronimo*.

Recoméndase utilizar un método de proba para cada caso de proba nas tres primeiras tarefas xa que é máis fácil ver o resultado de cada un deles. Na última tarefa deberanse de agrupar varios casos de proba nun mesmo método de proba.