ENTORNS DE DESENVOLUPAMENT (ED)

UD7. DISSENY I REALITZACIÓ DE PROVES

7.1 Introducció

Les proves de programari consisteixen a verificar i validar un producte software abans de la seva posada en marxa. Constitueixen una de les etapes del desenvolupament de programari, i bàsicament consisteix a provar l'aplicació construïda. S'integren dins de les diferents fases del cicle de vida del programari dins de l'enginyeria de software.

L'execució de proves d'un sistema involucra una sèrie d'etapes: planificació de proves, disseny i construcció dels casos de prova, definició dels procediments de prova, execució de les proves, registre de resultats obtinguts, registre d'errors trobats, depuració dels errors i informe dels resultats obtinguts.

Sent realistes, és pràcticament impossible realitzar proves exhaustives a un programa, atès que són molt costoses. Normalment, el que es fa és arribar a un punt intermedi en el qual es garanteix que no hi haurà defectes importants o molts defectes i l'aplicació és completament operativa amb un funcionament acceptable.

L'objectiu de les proves és convèncer, tant als usuaris com als mateixos desenvolupadors, que el programari és prou robust per a poder treballar amb ell de manera productiva.

Quan un programari supera unes proves exhaustives, les probabilitats que aquest programari doni problemes en producció s'atenuen i, per tant, la seva fiabilitat augmenta.

En aquest tema s'estudien dos punts de vista diferents per al disseny de casos de prova i diferents tècniques en cada un d'ells per a provar el codi dels programes.

7.2. Procediments i documentació per a les proves

Un procediment de prova és la definició de l'objectiu que desitja aconseguir-se amb les proves, què és el que es provarà i com. L'objectiu no sempre és detectar errors. A vegades es vol arribar a un rendiment concret, que la interfície tengui una aparença determinada, o que compleixi unes característiques donades. A vegades, que no hi hagi errors no vol dir que el programari superi aquestes proves.

Aquests procediments es dissenyen amb unes condicions concretes, especificant també les persones que faran les proves. No sempre són els programadors els que fan les proves. Sempre hi ha d'haver personal extern a l'equip de desenvolupament, ja que, en cas de saber on es troben els errors, ja els haurien corregit.

L'estàndard IEEE 829-1998 descriu el conjunt de documents que poden produir-se durant el procés de prova. Són els següents:

- **Pla de Proves**: descriu l'abast, l'enfocament, els recursos i el calendari de les activitats de prova. Identifica els elements a provar, les tasques que es realitzaran, el personal responsable de cada tasca i els riscos associats al pla.
- **Especificacions de prova**: estan cobertes per tres tipus de documents: l'especificació del disseny de la prova (s'identifiquen els requisits, casos de prova i procediments de prova necessaris per a dur a terme les proves i s'especifica la funció dels criteris de passa no-passa), l'especificació dels casos de prova (documenta els valors reals utilitzats per a l'entrada, juntament amb els resultats previstos), i l'especificació dels procediments de

prova (on s'identifiquen els passos necessaris per a fer funcionar el sistema i executar els casos de prova especificats).

- **Informes de prova**: es defineixen quatre tipus de documents: un informe que identifica els elements que estan sent provats, un registre de les proves (on es registra el que ocorre durant l'execució de la prova), un informe d'incidents de prova (descriu qualsevol esdeveniment que es produeix durant l'execució de la prova que requereix major recerca) i un informe resum de les activitats de prova.

7.3. Tècniques de disseny de casos de prova

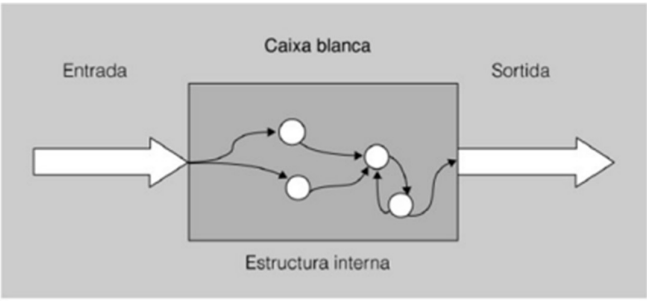
Un cas de prova és un conjunt d'entrades, condicions d'execució i resultats esperats, desenvolupat per aconseguir un objectiu concret o condició de prova. Per a dur a terme un cas de prova, és necessari definir les pre condicions i post condicions, identificar uns valors d'entrada i conèixer el comportament que hauria de tenir el sistema davant aquests valors. Després de fer aquest anàlisi i introduir aquestes dades en el sistema, s'observarà si el seu comportament és el previst o no i per què. D'aquesta manera es determinarà si el sistema ha passat o no la prova.

Per a dur a terme el disseny de casos de prova s'utilitzen dues tècniques o enfocaments: **prova de caixa blanca** i **prova de caixa negra**. Les primeres se centren a validar l'estructura interna del programa (necessiten conèixer els detalls procedimentals del codi) i les segones se centren a validar els requisits funcionals sense fixar-se en el funcionament intern del programa (necessiten saber la funcionalitat que el codi ha de proporcionar). Aquestes proves no són excloents i es poden combinar per a descobrir diferents tipus d'errors.

7.3.1 Proves de caixa blanca

També se les coneix com a proves estructurals o de caixa de cristall (*clear box testing*). Es basen a examinar minuciosament els detalls procedimentals del codi de l'aplicació. Mitjançant aquesta tècnica es poden obtenir casos de prova que:

- Garanteixin que s'executen almenys una vegada tots els camins independents de cada mòdul.
- Executin totes les sentències almenys una vegada.
- Executin totes les decisions lògiques en la seva part veritable i en la seva part falsa.
- Executin tots els bucles en els seus límits.
- Utilitzin totes les estructures de dades internes per a assegurar la seva validesa.
- Una de les tècniques utilitzades per a desenvolupar els casos de prova de caixa blanca és la prova del camí bàsic.



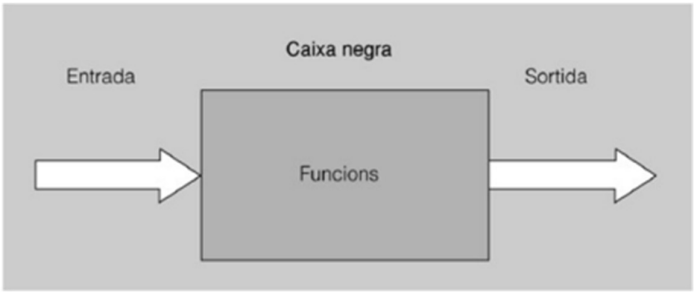
7.3.2 Proves de caixa negra

Aquestes proves es duen a terme sobre la interfície del programari, no fa falta conèixer l'estructura interna del programa ni el seu funcionament. Es pretén obtenir casos de prova que demostrin que les funcions del programari són operatives, és a dir, que les sortides que retorna l'aplicació són les esperades en funció de les entrades que es proporcionin.

A aquesta mena de proves també se'n diu prova de comportament. El sistema es considera com una caixa negra el comportament de la qual només es pot determinar estudiant les entrades i les sortides que retorna en funció de les entrades subministrades.

Amb aquest tipus de proves s'intenta trobar errors de les següents categories:

- Funcionalitats incorrectes o absents.
- Errors d'interfície.
- Errors en estructures de dades o en accés a bases de dades externes.
- Errors de rendiment.
- Errors d'inicialització i finalització.

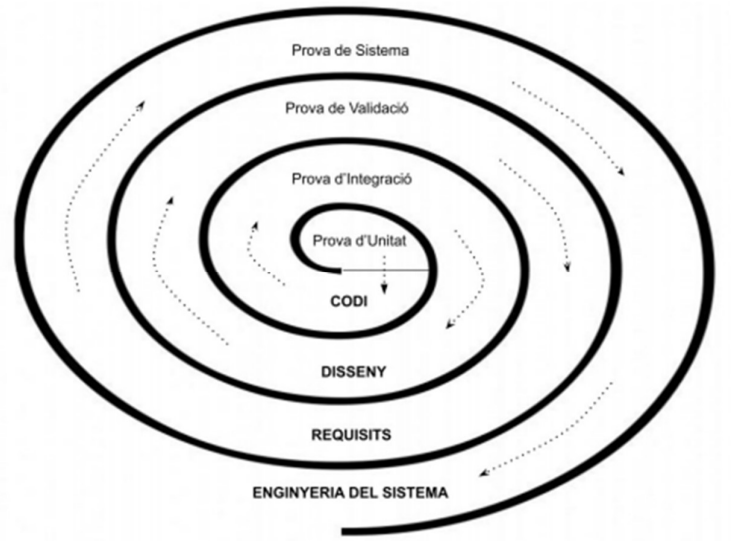


Existeixen diferents tècniques per a confeccionar els casos de prova de caixa negra, algunes d'elles són: classes d'equivalència, anàlisi de valors límit, mètodes basats en grafs, proves de comparació, etc.

7.4. Estratègies de proves del programari

L'estratègia de prova de programari es pot veure en el context d'una espiral:

- En el vèrtex de l'espiral comença la **prova d'unitat**. Se centra en la unitat més petita de programari, el mòdul tal com està implementat en codi font.
- La prova avança per a arribar a la **prova d'integració**. Es prenen els mòduls provats mitjançant la prova d'unitat i es construeix una estructura de programa que estigui d'acord amb el que dicta el disseny. El focus d'atenció és el disseny.



- L'espiral avança arribant a la **prova de validació** (o d'acceptació). Prova del programari en l'entorn real de treball amb intervenció de l'usuari final. Es validen els requisits establerts com a part de l'anàlisi de requisits del programari, comparant-los amb el sistema que ha estat construït.
- Finalment s'arriba a la **prova del sistema**. Verifica que cada element encaixa de manera adequada i s'aconsegueix la funcionalitat i rendiment total. Es prova com un tot el programari i altres elements del sistema.

7.4.1 Prova d'unitat

Solen realitzar-se durant les primeres fases de disseny i desenvolupament. Òbviament, no cal demorar molt la realització d'aquestes proves, ja que després cal integrar tot el programari (les diferents unitats) i els errors van acumulant-se i la localització i diagnòstic es compliquen.

En el cas de la POO, les proves unitàries haurien de realitzar-se a nivell **d'objecte**, i després, a nivell de **paquet** o llibreria. No té sentit provar un paquet per separat si no s'han dut a terme proves dels objectes individualment.

En aquest nivell es prova cada unitat o mòdul amb l'objectiu d'eliminar errors en la interfície i en la lògica interna. Aquesta activitat utilitza tècniques de caixa negra i caixa blanca, segons convingui. Es fan proves sobre:

- La **interfície** del mòdul, per a assegurar que la informació flueix adequadament.
- Les **estructures de dades** locals, per a assegurar que mantenen la seva integritat durant tots els passos d'execució del programa.
- Les **condicionis límit**, per a assegurar que funciona correctament en els límits establerts durant el procés.
- Tots els **camins independents** de l'estructura de control, amb la finalitat d'assegurar que totes les sentències s'executen almenys una vegada.
- Tots els **camins de maneig d'errors**.

Algunes de les eines que s'utilitzen per a les proves unitàries són: **JUnit**, CPPUnit, PHPUnit, etc.

7.4.2 Prova d'integració

En aquest tipus de prova s'observa com interaccionen els diferents mòduls. Es podria pensar que aquesta prova no és necessària, ja que, si tots els mòduls funcionen per separat, també haurien de funcionar junts. Realment el problema és aquest, comprovar si funcionen junts.

Existeixen dos enfocaments fonamentals per a dur a terme les proves:

- Integració **no incremental** o big bang. Es prova cada mòdul per separat i després es combinen tots d'una vegada i es prova tot el programa complet. En aquest enfocament es troben gran quantitat d'errors i la correcció es fa difícil.
- Integració **incremental**. El programa complet es va construint i provant en petits segments. En aquest cas els errors són més fàcils de localitzar. Es donen dues estratègies: **ascendent** i **descendent**. En la integració ascendent la construcció i prova del programa comença des dels mòduls dels nivells més baixos de l'estructura del programa. En la descendent la integració comença en el mòdul principal (programa principal) movent-se cap avall per la jerarquia de control.

7.4.3 Prova de validació

La validació s'aconsegueix quan el programari funciona **d'acord amb les expectatives** raonables del client definides en el document d'especificació de requisits del programari. Es duen a terme una sèrie de proves de caixa negra que demostren la conformitat amb els requisits. Les tècniques que utilitzarem són:

- Proves **alfa**. Es duu a terme pel client o usuari en el lloc de desenvolupament. El client utilitza el programari de manera natural sota l'observació del desenvolupador que anirà registrant els errors i problemes d'ús.
- Proves **beta**. Es duu a terme pels usuaris finals del programari en el seu lloc de treball. El desenvolupador no és present. L'usuari registra tots els problemes que troba, reals i/o imaginaris, i informa el desenvolupador en els intervals definits en el pla de prova. Com a resultat dels problemes informats, el desenvolupador duu a terme les modificacions i prepara una nova versió del producte.

7.4.4 Prova del sistema

La prova del sistema està formada per un conjunt de proves, que tenen com a objectiu exercitar profundament el programari. Són les següents:

- **Prova de recuperació**: En aquest tipus de prova es força la fallada del programari i es verifica que la recuperació es duu a terme apropiadament.
- **Prova de seguretat**: Aquesta prova intenta verificar que el sistema està protegit contra accessos il·legals.
- **Prova de resistència** (stress test): Tracta d'enfrontar el sistema amb situacions que demanden gran quantitat de recursos, per exemple, dissenyant casos de prova que requereixin el màxim de memòria, incrementant la freqüència de dades d'entrada, que donin problemes en un sistema operatiu virtual, etc.

7.5. Proves de codi

La prova del codi consisteix en l'execució del programa (o part d'ell) amb l'objectiu de trobar errors. Es parteix per a la seva execució d'un conjunt d'entrades i una sèrie de condicions d'execució; s'observen i registren els resultats i es comparen amb els resultats esperats. S'observarà si el comportament del programa és el previst o no o per què.

Hi ha diverses tècniques per a les proves de codi:

- La **prova del camí bàsic** és una tècnica de prova de caixa blanca que permet al dissenyador de casos de prova obtenir una mesura de la complexitat lògica d'un disseny procedimental i usar aquesta mesura com a guia per a la definició d'un conjunt bàsic de camins d'execució. Els casos de prova aconseguits del conjunt bàsic garanteixen que durant la prova s'executa almenys una vegada cada sentència del programa.
- La **partició o classes d'equivalència** és un mètode de caixa negra que divideix els valors dels camps d'entrada d'un programa en classes d'equivalència. Per exemple, si tenim un camp d'entrada anomenat número d'empleat amb aquestes condicions: numèric de 3 dígits, i el primer no pot ser 0. Aleshores es pot definir una classe d'equivalència no vàlida: número d'empleat < 100; i una altra de vàlida: número d'empleat entre 100 i 999.
- L'**anàlisi de valors límit** es basa en el fet que els errors tendeixen a produir se amb més probabilitat en els límits o extrems dels camps d'entrada. Aquesta tècnica complementa l'anterior i els casos de prova elegits exerciten els valors just per damunt i per davall dels marges de la classe d'equivalència.

7.6. Proves unitàries amb JUnit

En aquest apartat aprendrem a utilitzar una eina per a implementar proves que verifiquin que el nostre programa genera els resultats que d'ell esperem.

JUnit és una eina per a realitzar proves unitàries automatitzades. Està integrada en Eclipse, per la qual cosa no és necessari descarregar-se cap paquet per a poder usar-la. Les proves unitàries es fan sobre una classe per a provar el seu comportament de manera aïllada independentment de la resta de classe de l'aplicació. Encara que això no sempre és així perquè una classe a vegades depèn d'altres classes per a poder dur a terme la seva funció.

En cas que no aparegui al menú File >>New >> JUnit Test Case es pot descarregar el JUnit d'aquesta url: https://github.com/junit-team/junit4/wiki/Download-and-Install

S'han de descarregar els dos fitxers

- junit.jar
- hamcrest-core.jar

S'emmagatzemen dins un directori per exemple c:\JUnit

I s'han de crear variables d'entorn

JUNIT_HOME = c:\JUnit

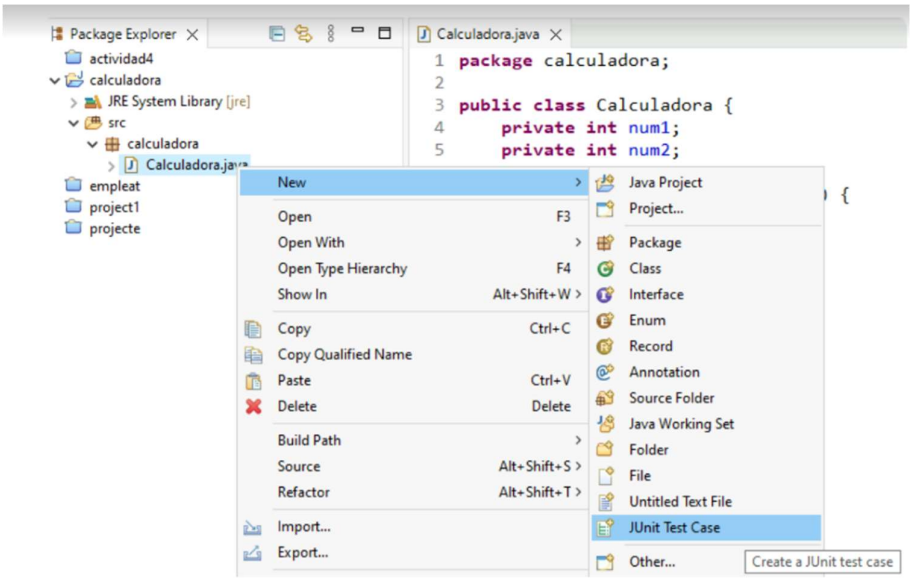
CLASSPATH=%CLASSPATH%;%JUNIT_HOME%\junit.jar

7.6.1 Creació d'una classe de prova

Per a començar a emprar JUnit creem un nou projecte a Eclipse i creem la classe a provar, en aquest cas es diu Calculadora:

```
public class Calculadora {
	private int num1;
	private int num2;
	public Calculadora(int a, int b) {
		num1 = a;
		num2 = b;
	}
	public int suma() {
		int resul = num1 + num2;
		return resul;
	}
	public int resta() {
		int resul = num1 - num2;
		return resul;
	}
	public int multiplica() {
		int resul = num1 * num2;
		return resul;
	}
	public int divideix() {
		int resul = num1 / num2;
		return resul;
	}
}
```

A continuació, cal crear la classe de prova. Amb la classe *Calculadora* seleccionada premem el botó dret del ratolí i seleccionem ***New >> JUnit Test Case***.



També es pot fer des del menú ***File >> New >> JUnit Test Case***. En qualsevol cas s'obre una finestra de diàleg. Des d'aquí hem de marcar ***New JUnitJupiter test***, la resta d'opcions les deixem

amb els valors per defecte. Com a nom de classe es generarà el nom *CalculadoraTest*. Premem el botó *Next*. A continuació, hem de seleccionar els mètodes que volem provar, marquem els quatre mètodes i premem *Finish*. S'obre una finestra indicant-nos que la llibreria JUnit 5 no està inclosa en el nostre projecte, premem el botó OK perquè s'inclogui.

Perquè la classe de prova s'inclogui en un paquet diferent al de la classe a provar s'escriu el nom del paquet en el camp Package (en l'exemple la classe de prova i la classe a provar estan dins del mateix paquet).

```
import static org.junit.jupiter.api.Assertions.*;
import org.junit.jupiter.api.Test;

class CalculadoraTest {

	@Test
	void testSuma() {
		fail("Not yet implemented");
	}

	@Test
	void testResta() {
		fail("Not yet implemented");
	}

	@Test
	void testMultiplica() {
		fail("Not yet implemented");
	}

	@Test
	void testDivideix() {
		fail("Not yet implemented");
	}
}
```

La classe de prova es crea automàticament, s'observen una sèrie de característiques:

- Es creen quatre mètodes de prova, un per a cada mètode triat abans.
- Els mètodes són públics, no retornen res i no reben cap argument.
- El nom de cada mètode inclou la paraula test al començament: testSuma(), testResta(), testMultiplica(), testDivideix().
- Sobre cadascun dels mètodes apareix l'anotació @Test que indica al compilador que és un mètode de prova.
- Cadascun dels mètodes de prova té una crida al mètode fail() amb un missatge indicant que encara no s'ha implementat el mètode. Aquest mètode fa que el test acabi amb error llançant el missatge tancat entre cometes.

7.6.2 Preparació i execució de les proves

Abans de preparar el codi pels mètodes de prova anem a veure una sèrie de **mètodes per fer les comprovacions**. Totes les assercions de *JUnit Jupiter* són mètodes estàtics de la classe *org.junit.jupiter.api.Assertions*. Tots aquests mètodes retornen un tipus *void*:

MÈTODES	MISSIÓ
assertTrue(boolean expressió) **assertTrue(boolean expressió, String missatge)**	Comprova que l'expressió s'avaluï a *true*. Si no és *true* i s'inclou l'String, en produir-se error es llançarà el *missatge*.
assertFalse(boolean expressió) **assertFalse(boolean expressió, String missatge)**	Comprova que l'expressió s'avaluï a false. Si no és false i s'inclou l'String, en produir-se error es llançarà el *missatge*.
assertEquals(double valorEsperat, double valorReal, double delta) **assertEquals(double valorEsperat, double valorReal, double delta, String missatge)** (Es pot emprar qualsevol tipus de dada: Integer, Short, Object, etc; *delta* s'empra en tipus float i double)	Comprova que el *valorEsperat* sigui igual al *valorReal*. Si no són iguals i s'inclou l'String, aleshores es llançarà el *missatge*. *ValorEsperat* i *valorReal* poden ser de tipus diferents. El *delta,* descriu la diferència admissible entre el valor esperat i el valor real per considerar que ambdós nombres són iguals. Un valor 0 indica que han de ser iguals. Un valor de 0.01 consideraria aquests dos nombres iguals: 1259.9916 i 1259.9917. Un valor de 0 els consideraria diferents.
assertNull(Object objecte) **assertNull(Object objecte, String missatge)**	Comprova que l'*objecte* sigui *null*. Si no és *null* i s'inclour l'*String*, en produir-se error es llançarà el *missatge*.
assertNotNull(Object objecte) **assertNotNull(Object objecte, String missatge)**	Comprova que l'objecte no sigui *null*. Si és *null* i s'inclou l'*String*, en produir-se error es llançarà el missatge.
assertSame(Object objecteEsperat, Object objecteReal) **assertSame(Object objecteEsperat, Object objecteReal, String missatge)**	Comprova que *objecteEsperat* i *objecteReal* siguin el mateix objecte. Si no ho són i s'inclou l'*String*, aleshores es llançarà el missatge en cas d'error.

assertNotSame(Object objecteEsperat, Object objecteReal) **assertNotSame(Object objecteEsperat, Object objecteReal, String missatge)**	Comprova que *objecteEsperat* no sigui el mateix objecte que *objecteReal*. Si ho són i s'inclou l'*String*, aleshores es llançarà el missatge en cas d'error.
fail() **fail(String missatge)**	Fa que la prova falli. Si s'inclou un *String* la prova falla llançant el *missatge*.

Es pot trobar més informació d'aquests i altres mètodes a la següent web:

https://junit.org/junit5/docs/current/api/org.junit.jupiter.api/org/junit/jupiter/api/Assertions.html

Anem ara a definir els següents casos de prova per provar els mètodes de la classe Calculadora.

MÈTODE A PROVAR	ENTRADA	SORTIDA ESPERADA
Suma	20, 10	30
Resta	20, 10	10
Multiplica	20, 10	200
Divideix	20, 10	2

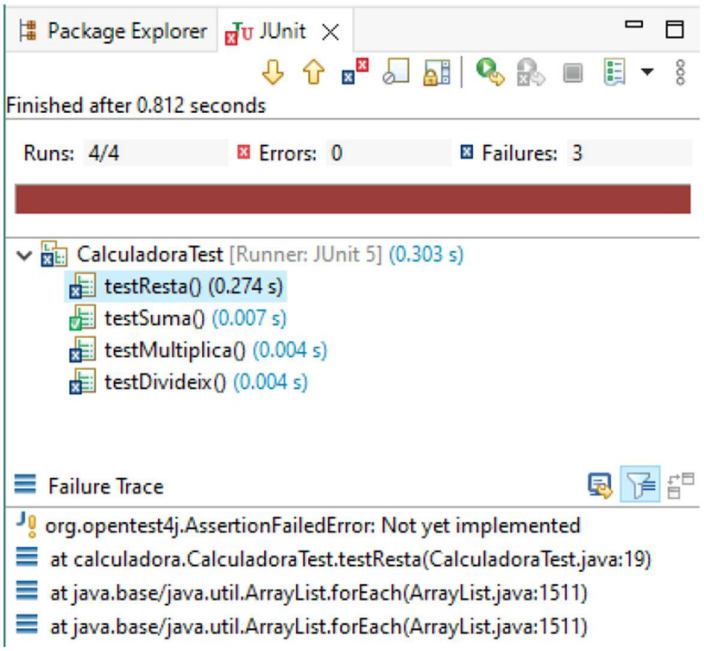
Tot cas de prova està format per tres parts:

1. S'indica en una variable quin és el **valor esperat** pel mètode que anem a provar.
2. S'executa el mètode que volem provar amb les **dades d'entrada** adequats i es guarda el **resultat** en una altra variable.
3. Es comprova la relació entre el **valor esperat** i el resultat de la crida al mètode (**valor real**), a través d'una asserció. En aquest cas l'asserció és ***assertEquals()***.

Cream el codi de prova pel mètode testSuma() que provarà el mètode suma() de la classe Calculadora:

```
	@Test
	public void TestSuma() {
		double valorEsperat = 30;
		Calculadora calcu = new Calculadora(20, 10);
		double resultat = calcu.suma();
		assertEquals(valorEsperat, resultat, 0);
	}
```

Per executar la classe de prova premem el botó ***Run***, o bé, si esteim sobre la classe, amb el botó dret del ratolí seleccionam l'opció ***Run As >> Junit Test***. S'obre la pestanya de *Junit* on es mostren els resultats d'execució de les proves.



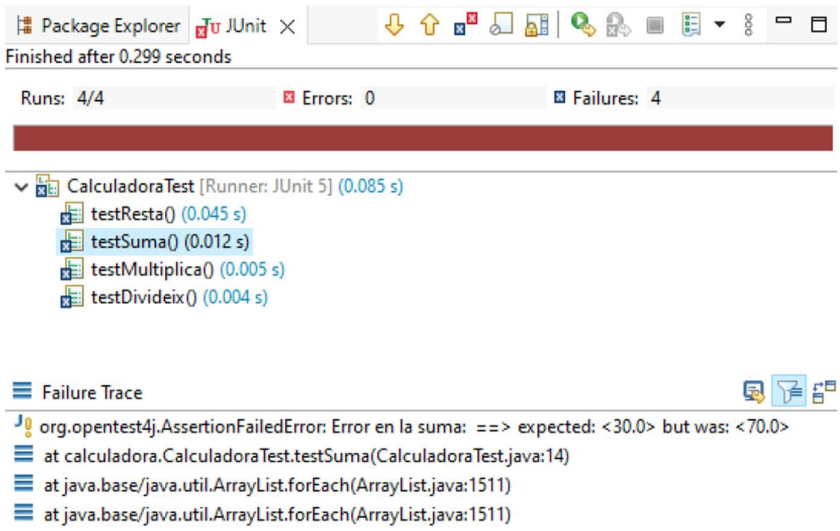
Al costat de cada prova apareix una icona amb una marca: una marca de verificació de color verd indica que la prova ha anat bé, una creu blava indica que hi ha un error. En el panell inferior es mostra informació sobre l'error i el número de línia del cas de test on s'ha donat l'error.

El resultat de l'execució de la prova mostra: Runs: 4/4, Errors: 0, Failures: 3; això ens indica que s'han fet 4 proves, cap d'elles ha provocat error i 3 d'elles han provocat fallada.

En el context de Junit una fallada és una comprovació que no es compleix, un error és una excepció durant l'execució del codi. En aquesta prova només s'ha realitzat satisfactòriament la prova amb el mètode *testSuma()* que mostra una icona amb una marca de verificació al costat. La resta de proves han fallat, mostren la icona amb una creu blava (recordem que per defecte els mètodes inclouen el mètode fail() que fa fallar la prova).

Per veure la diferència entre una fallada i un error canviarem el codi del mètode suma(). Per fer-ho li direm que el valor esperat no coincideixi amb el resultat.

```
	@Test
	public void TestSuma() {
		double valorEsperat = 30;
		Calculadora calcu = new Calculadora(20, 50);
		double resultat = calcu.suma();
		assertEquals(valorEsperat, resultat, 0,  "Error en la suma: ");
	}
```



En prémer en els test que han produït fallades o errors es mostra la traça d'execució. Per veure la traça completa d'execució podeu pitjar el botó ***Filter Stack Trace*** (embut).

El missatge mostrat en el panell inferior en seleccionar *testSuma()* és el següent: "Error en la suma: ==> expected: <30.0> but was: <70.0>" que ens informa que ha fallat perquè esperava 30 i ha rebut com a resposta 70. És una fallada perquè el cas de prova està mal definit.

Exercici 7.1: Donada la classe Calculadora, crea la classe de test pels 4 mètodes donats: suma, resta, multiplica i divideix i acaba d'escriure els mètodes per testejar la resta, la multiplicació i la divisió.

7.6.3 Provar excepcions controlades

JUnit ens permet comprovar que un mètode llança una excepció controlada. Amb JUnit 5 s'ha d'emprar el mètode org.junit.jupiter.Assertions.assertThrows() per a provar casos d'excepció. Aquestes excepcions han d'extendre de Throwable.

MÈTODE	MISSIÓ
assertThrows(Exception.class, () -> {}) **assertThrows(Exception.class, () -> {}, missatge)**	Comprova que el mètode que se li passa al segon paràmetre llanci una excepció del tipus especificat al primer argument i la retorni.

Abans de veure un exemple, anotar breument què és l'expressió () -> {}. Això es diu **expressió lambda** i seria com treballar amb un mètode anònim.

Normalment crearíem un mètode amb un nom, per exemple

```
public int suma (int a, int b){
 return a+b;
}
```

Però si volem passar aquest mètode com un paràmetre a un altre mètode es pot abreviar utilitzant una expressió lambda d'aquesta manera tan senzilla:

```
( a , b ) -> { a + b }
```

I si simplement volem un mètode que no té paràmetres d'entrada i que executi una sèrie d'instruccions podem construir una expressió lambda amb aquest aspecte:

```
( ) -> { instruccio1; instruccio2;}
```

En aquest cas serveix per poder passar comportament (una funció) com a paràmetre. Ho veurem amb un exemple a continuació.
Per exemple, per a fer que el mètode *divideix0()* llanci l'excepció ArithmeticException si el denominador és 0, es pot utilitzar la instrucció ***throw*** per a llançar una ArithmeticException de la següent manera:

```
	public int divideix() {
		if(num2 == 0) {
			throw new java.lang.ArithmeticException("Divisió per 0");
		}
		else{
				int resul= num1 / num2;
				return resul;
			}
	}
```

En la classe de prova, per a poder verificar que es llança aquesta excepció, s'utilitza el mètode ***assertThrows()*** de la següent manera:

```
	@Test
	public void testDivideix() {
		Calculadora calcu = new Calculadora(20, 0);
		assertThrows(ArithmeticException.class, () ->
calcu.divideix0());
	}
```

Aquí el codi executable és *calcu.divideix0()* que llança l'excepció *ArithmeticException* en executar el mètode *divideix0()*. A continuació teniu una altra manera d'escriure el test anterior emprant *assertEquals()*:

```
	@Test
	public void testDivideix() {
		Calculadora calcu = new Calculadora(20, 0);
		Exception exception = assertThrows(ArithmeticException.class, ()
-> calcu.divideix());
		assertEquals("Divisió per 0", exception.getMessage());
	}
```

La prova fallarà si no es produeix l'excepció.

Exercici 7.2: Modifica el mètode test*Divideix()* perquè es testegin diferents objectes de tipus Calculadora uns amb valors que provoquin el llançament de l'excepció i altres que no. Per exemple Calculadora (20,0) i Calculadora (20,5).

Exercici 7.3: Modifica el mètode resta() de la classe Calculadora i afegeix els mètodes resta2() i divideix2() que s'exposen a continuació. Crea després els test per provar els 3 mètodes. Els mètodes són:

```
	public int resta() {
		int resul;
		if (resta2()){
			resul = num1 - num2;
		 }else {
			resul = num2 - num1;
		 }
		return resul;
	}
	public boolean resta2() {
		if (num1 >= num2) return true;
		else return false;
	}
	public Integer divideix2() {
		if (num2 == 0) return null;
		int resul = num1 / num2;
		return resul;
	}
```

Empra els mètodes ***assertTrue()***, ***assertFalse()***, ***assertNull()***, ***assertNotNull()*** o ***assertEquals()*** segons convingui.

7.6.4 Emprar assertAll

Per poder combinar més d'un assert podem emprar assertAll. Això ho farem quan tenim moltes comprovacions a realitzar. Si en el nostre codi tenim una llista d'asserts, un a continuació de l'altre i en falla un, no s'executen la resta d'asserts que tinguem escrits, i, per tant, no sabrem el seu resultat.

AssertAll **executa tots els assert** independentment que només un falli i ens diu on hi ha els errors i per quin motiu s'ha donat. Això ens donarà una visió més completa de la qualitat d'aquest test, del que s'està provant.

Aquesta seria la manera d'utilitzar aquest assert:

```
	assertAll(String message, () -> {});
```

Podem incloure diverses expressions lambda.

A continuació podem veure un exemple d'ús. Així seria el codi amb els *asserts* un a continuació de l'altre:

```
	@Test
	public void TestSuma() {
		Calculadora calcu = new Calculadora();
		assertEquals(100, calcu.suma(1,1), "No suma correctament dos
nombres positius");
		assertEquals(100, calcu.suma(-1,1), "No suma correctament un
nombre negatiu i un nombre positiu");
		assertNotNull(calcu, "La variable calcu s'ha d'inicialitzar");
	}
```

I així amb *assertAll*:

```
	@Test
	public void TestSuma() {
		Calculadora calcu = new Calculadora();
		assertAll(
				() -> assertEquals(2, calcu.suma(1,1), "No suma
correctament dos nombres positius"),
				() -> assertEquals(0, calcu.suma(-1,1), "No suma
correctament un nombre negatiu i un nombre positiu"),
				() -> assertNotNull(calcu, "La variable calcu s'ha
d'inicialitzar")
				);
	}
```

Exercici 7.4: Escriu una classe de test per provar el mètode *calcul()* de la classe Factorial. En el mètode es comprova si el nombre és menor que 0. En aquest cas es llança una excepció *IllegalArgumentException* amb el missatge *"Número n no pot ser menor que 0".* Si el valor del factorial calculat és menor que 0 és que hi ha hagut un error de desbordament. En aquest cas es llança l'excepció *ArithmeticException* i es llança el missatge "*Overflow, número n massa gran*".

S'ha de provar:

- Que el càlcul és correcte quan el rang de números està dins els límits.
- Que surt una excepció *IllegalArgumentException* quan li introduïm per paràmetre un número negatiu
- Que surt una excepció *ArithmeticExcepcion* quan es passa un número massa gran per paràmetre.

La classe a provar és la següent:

```
public class Factorial {
	public static int calculo(int n) {
		if (n < 0 ) {
			throw new IllegalArgumentException("Número "+ n +
					" no pot ser menor que 0");
		}
		int fact = 1;
		for (int i = 2; i <= n; i++)
			fact *= i;
		if (fact <= 0 ) {
			throw new ArithmeticException("Overflow, número "+
					n +" massa gran");
		}
		return fact;
	}
}
```

7.6.5 Tipus d'anotacions

En tots els mètodes de prova anteriors es repetia la línia *Calculadora calcu = new Calculadora(20, 10);*. Aquesta sentència d'inicialització es pot escriure una sola vegada dins de la classe.

JUnit disposa d'una sèrie d'anotacions que permeten executar codi abans i després de les proves:

- **@beforeEach**: si anotem un mètode amb aquesta etiqueta, el codi s'executarà abans de qualsevol mètode de prova. Aquest mètode es pot utilitzar per a inicialitzar dades. Per exemple, en una aplicació d'accés a base de dades es pot preparar la base de dades, inicialitzant un array per a les proves. Pot haver-hi diversos mètodes en la classe prova amb aquesta anotació.
- **@AfterEach**: si anotem un mètode amb aquesta etiqueta el codi serà executat després de l'execució de cada mètode de prova. Es pot utilitzar per a netejar dades. Pot haver-hi diversos mètodes en la classe de prova amb aquesta anotació.

La classe *CalculadoraTest* incloent dos mètodes amb les anotacions @BeforeEach i @AfterEach quedaria de la següent manera, en el primer mètode ***creaCalculadora()*** inicialitzem l'objecte calculadora i en el segon ***borrarCalculadora()*** el netegem:

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

public class CalculadoraTest2 {
	private Calculadora calcu;

	@BeforeEach
	public void creaCalculadora() {
		calcu = new Calculadora(20, 10);
	}

	@AfterEach
	public void borraCalculadora() {
		calcu = null;
	}

	@Test
	public void testSuma() {
		double valorEsperat = 30;
		double resultat = calcu.suma();
		assertEquals(valorEsperat, resultat, 0);
	}

	@Test
	public void testResta() {
		double valorEsperat = 10;
		double resultat = calcu.resta();
		assertEquals(valorEsperat, resultat, 0);
	}

	@Test
	public void testMultiplica() {
		double valorEsperat = 200;
		double resultat = calcu.multiplica();
		assertEquals(valorEsperat, resultat, 0,
				"Fallada en la multiplicació: ");
	}

	@Test
	public void testDivideix() {
```

```
            double valorEsperat = 2;
            double resultat = calcu.divideix();
            assertEquals(valorEsperat, resultat, 0);
      }
}
```

Altres anotacions a destacar són **@BeforeAll** i **@AfterAll**, que tenen algunes diferències respecte a les anteriors:

- **@BeforeAll**: només pot haver-hi un mètode amb aquesta etiqueta. El mètode marcat amb aquesta anotació és invocat una vegada al principi del llançament de totes les proves. Se sol utilitzar per a inicialitzar atributs comuns a totes les proves o per a realitzar accions que triguen un temps considerable a executar-se.
- **@AfterAll**: només pot haver-hi un mètode amb aquesta anotació. Aquest mètode serà invocat una sola vegada quan finalitzin totes les proves.

En aquest cas els mètodes anotats amb @BeforeAll i @AfterAll han de ser *static* i, per tant, els atributs als quals accedeixen també. Els mètodes afegits anteriorment quedarien així:

```
import static org.junit.jupiter.api.Assertions.*;
import org.junit.jupiter.api.AfterAll;
import org.junit.jupiter.api.BeforeAll;
import org.junit.jupiter.api.Test;
public class CalculadoraTest3 {
      private static Calculadora calcu;
      private double resultat;
      @BeforeAll
      public static void creaCalculadora() {
            calcu = new Calculadora(20, 10);
      }
      @AfterAll
      public static void borraCalculadora() {
            calcu = null;
      }
      ..................
      ..................
}
```

Exercici 7.5: Escriu una classe de proves per a provar els mètodes de la classe *TablaEnteros*. En aquesta classe de prova crea un mètode amb l'anotació ***@BeforeAll*** en el qual inicialitzis un array d'enters per a emprar-lo en les proves dels mètodes. El mètode *sumaTabla()* suma els elements de l'array i retorna la suma. El mètode *mayorTabla()* retorna l'element major de la taula. I el mètode *posicionTabla()* retorna la posició ocupada per l'element el valor del qual s'envia.

En el constructor es comprova si el nombre d'elements de la taula és nul o 0, en aquest cas es llança l'excepció *IllegalArgumentException* amb el missatge No hi ha elements. El mètode *posicionTabla* també llança l'excepció *java.util.NoSuchElementException*, en el cas que no es trobi l'element en la taula.

Cal afegir dos mètodes de prova més per a provar aquestes excepcions.

La classe a provar és la següent:

```
public class TablaEnteros {
```

```
	private Integer[] tabla;
	TablaEnteros(Integer[] tabla) {
		if (tabla == null || tabla.length == 0)
			throw new IllegalArgumentException("No hay elementos");
		this.tabla = tabla;
	}
	// devuelve la suma de los elementos de la tabla
	public int sumaTabla() {
		int suma = 0;
		for (int i = 0; i < tabla.length; i++)
			suma += tabla[i];
		return suma;
	}
	// devuelve el mayor elemento de la tabla
	public int mayorTabla() {
		int max = -999;
		for (int i = 0; i < tabla.length; i++)
			if (tabla[i] > max)
				max = tabla[i];
		return max;
	}
	// devuelve la posición de un elemento cuyo valor se pasa
	public int posicionTabla(int n) {
		for (int i = 0; i < tabla.length; i++)
			if (tabla[i] == n)
				return i;
		throw new java.util.NoSuchElementException("No existe: " +n);
	}
}
```

7.6.6 Proves parametritzades

Les proves parametritzades permeten executar diverses vegades una prova amb diferents valors en els seus arguments. Es declaren utilitzant l'anotació ***@ParametrizedTest*** en lloc de l'anotació *@Test*. A més, s'ha de declarar almenys una font que proporcionarà els arguments per a cada invocació i després consumir els arguments en el mètode de prova.

Si el nostre mètode de prova necessita que li passem només un paràmetre de tipus *String* o qualsevol tipus primitiu de dada, usarem l'anotació ***@ValueSource***. Aquesta anotació ens permet especificar una única matriu de valors literals i només es pot emprar per a proporcionar un únic argument al mètode de prova.

El següent exemple mostra una prova parametritzada que empra l'anotació *@ValueSource* per provar el mètode *mensajeNoNulo(String cadena)*. L'argument pel mètode el prendrà de l'array de cadenes, fent tantes proves com cadenes hi ha a l'array. En l'exemple es fan dues proves del mètode, una amb el valor "Hola" i una altra el valor "Mundo":

```
	@ParameterizedTest
	@ValueSource(strings = { "Hola", "Mundo" })
	void mensajeNoNulo(String cadena) {
		assertNotNull(cadena);
	}
```

Si volem fer un mètode de prova que rebi un valor enter hem d'especificar un array d'enters. Emprarem l'atribut ***ints*** en lloc de *strings* de la següent manera *@ValueSource(ints = {10, 20, 9})*. Per dades de tipus *double* emprarem l'atribut ***doubles***, pels de tipus *float* emprarem ***floats***, i així amb la resta de tipus de dades primitives.

Per a proporcionar a un mètode de prova diversos paràmetres hem d'utilitzar l'anotació ***@CsvSource***. Quan afegim aquesta anotació, hem de configurar les dades de prova emprant una matriu d'objectes *String*. En especificar les dades de prova, hem de seguir aquestes regles:

- Un objecte *String* ha de contenir tots els paràmetres per a una invocació del mètode.
- Els diferents paràmetres han d'estar separats amb una coma.
- Els valors trobats en cada línia han de seguir el mateix ordre que els paràmetres definits en el mètode de prova.

En els següents exemples es prova el mètode *divideix()* de la classe *Calculadora*. En el primer exemple configuram els paràmetres que es passen al mètode de prova *testDivideix0()(int, int, int)*, que accepta tres paràmetres int. El valor del tercer paràmetre especifica el resultat de la divisió esperada dels dos valors anteriors, per exemple (20, 10 i 2).

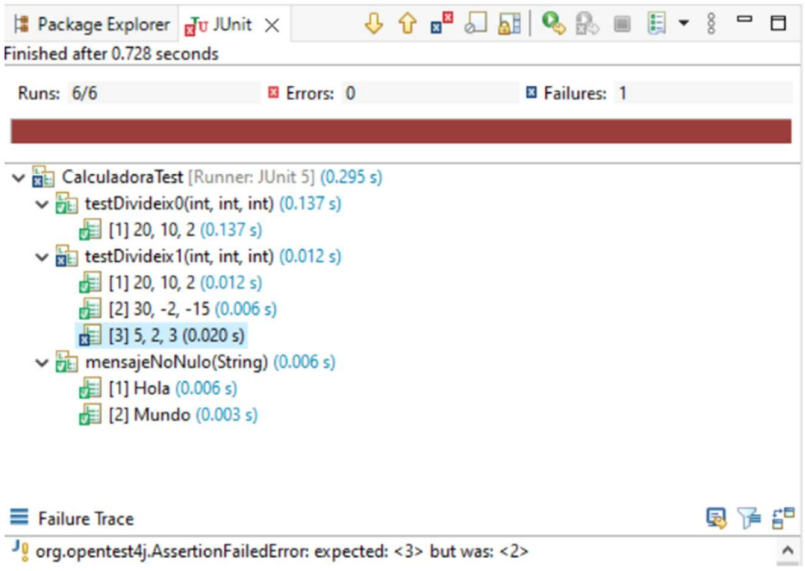
Els valors per cada cas de prova es defineixen entre claus i tancats entre cometes dobles i els paràmetres separats per comes. Aquests s'assignaran als paràmetres del mètode en l'ordre en que estan definits: *a* = 20, *b* = 10 i *valorEsperat* = 2:

```
@ParameterizedTest
@CsvSource({ "20, 10, 2" })
public void testDivideix0(int a, int b, int valorEsperat) {
	Calculadora calcu = new Calculadora(a, b);
	int resultat = calcu.divideix();
	assertEquals(valorEsperat, resultat);
```

En el segon mètode de prova *testDivideix1(int, int, int)*, es donen tres casos de prova, és a dir, tres objectes String; cada prova proporcionarà els paràmetres necessaris al mètode:

```
@ParameterizedTest
@CsvSource({ "20, 10, 2", "30, -2, -15","5, 2, 3" })
public void testDivide1(int a, int b, int valorEsperat) {
	Calculadora calcu = new Calculadora(a, b);
	int resultat = calcu.divideix();
	assertEquals(valorEsperat, resultat);
}
```

L'execució produeix la sortida mostrada a continuació. Sota cada mètode de prova es mostra entre claudàtors la prova que es tracta. Per exemple, dins de *testDivideix0()* es prova el grup de valors {20,10,2}, mentre que dins de *testDivideix1()* es proven els tres grups {20, 10, 2}, {30, -2, -15} i {5, 2, 3}.



Exercici 7.6: Fes les proves parametritzades pels mètodes *suma()*, *resta()* i *multiplica()* de la classe Calculadora.

Exercici 7.7: Fes les proves parametritzades per la classe Factorial de l'exercici 6. Una classe de prova per a cada excepció i una altra pel càlcul correcte.

Exercici 7.8: Desenvolupa una bateria de proves per a provar el mètode *DevuelveFecha()* de la classe *Fecha* que s'exposa a continuació. El mètode rep un nombre enter i retorna un String amb un format de data que dependrà del valor d'aquest número. Si el número rebut és diferent de 1, 2 i 3 el mètode retorna "ERROR". La classe és la següent:

```
import java.text.SimpleDateFormat;
import java.util.Date;
public class Fecha {
	SimpleDateFormat formato;
	Date hoy;
	public Fecha() {
		hoy = new Date();
	}
	public String DevuelveFecha(int tipo) {
		String cad = "";
		switch (tipo) {
		case 1: {
			formato = new SimpleDateFormat("yyyy/MM");
			cad = formato.format(hoy);
			break;
		}
		case 2: {
			formato = new SimpleDateFormat("MM/yyyy");
			cad = formato.format(hoy);
			break;
		}
		case 3: {
			formato = new SimpleDateFormat("MM/yy");
			cad = formato.format(hoy);
			break;
```

```
                }
                default: {
                        cad = "ERROR";
                }
                }
                return cad;
        }
}
```

7.6.7 Suite de proves

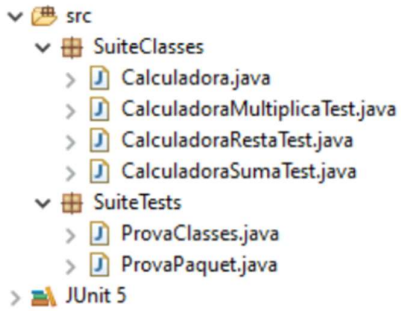
A vegades ens interessa executar diverses classes de prova l'una després de l'altra. Per a això, JUnit proporciona un mecanisme anomenat Test Suites que permet agrupar diverses classes de prova perquè s'executin una a continuació de l'altra. En la suite de proves es poden agrupar les classes que formen part d'un paquet, o bé es poden indicar els noms de les classes que formaran part de la suite.

Per a crear la suite de proves hem de crear una classe amb algunes de les següents anotacions:

- ***@Suite***: aquesta anotació és necessària incloure-la.
- ***@SelectPackages("nom de paquet")***: si les classes a provar estan dins d'un paquet, usem aquesta anotació. El nom de les classes ha d'acabar amb la paraula Test o Tests.
- ***@SelectClasses({Class1Test.class, Class2Test.class, ...})***: empram aquesta anotació si volem indicar el nom de les classes que formen part del conjunt de proves i que són les que s'executaran.
- ***@SuiteDisplayName("Text")***: aquesta anotació s'empra per declarar un nom personalitzat que s'executa com un conjunt. Aquest nom es mostrarà en executar la suite.
- Dins de la classe que es crea no es genera cap línia de codi.

Per exemple, si tenim el paquet *SuiteClasses* amb les classes a provar, totes acaben amb la paraula Test: *CalculadoraMultiplicaTest*, *CalculadoraSumaTest* i *CalculadoraRestaTest*.

En el paquet *SuiteTests* tenim les suites de proves: *ProvaClasses* emprarà l'anotació *@SelectClasses()* per provar un conjunt de classes; i *ProvaPaquet* emprarà *@SelectPackages()* per fer les proves de les classes del paquet *SuiteClasses*.



El codi de la classe *ProvaClasses* és el següent:

```
package SuiteTests;
```

```java
import org.junit.platform.suite.api.SelectClasses;
import org.junit.platform.suite.api.Suite;
import org.junit.platform.suite.api.SuiteDisplayName;
import org.junit.runners.Suite.SuiteClasses;

@Suite
@SuiteDisplayName("Demostració Prova dues classes")
@SelectClasses({SuiteClasses.CalculadoraMultiplicaTest.class,
      SuiteClasses.CalculadoraSumaTest.class
})
public class ProvaClasses {
}
```

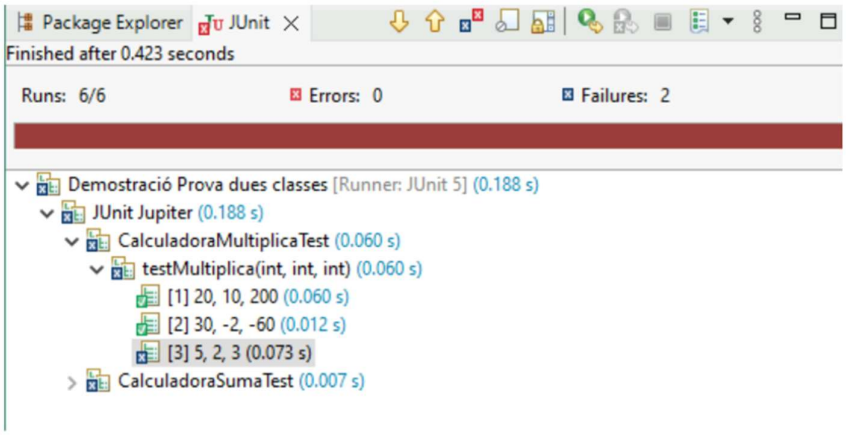
El codi de la classe *ProvaPaquet* és el següent:

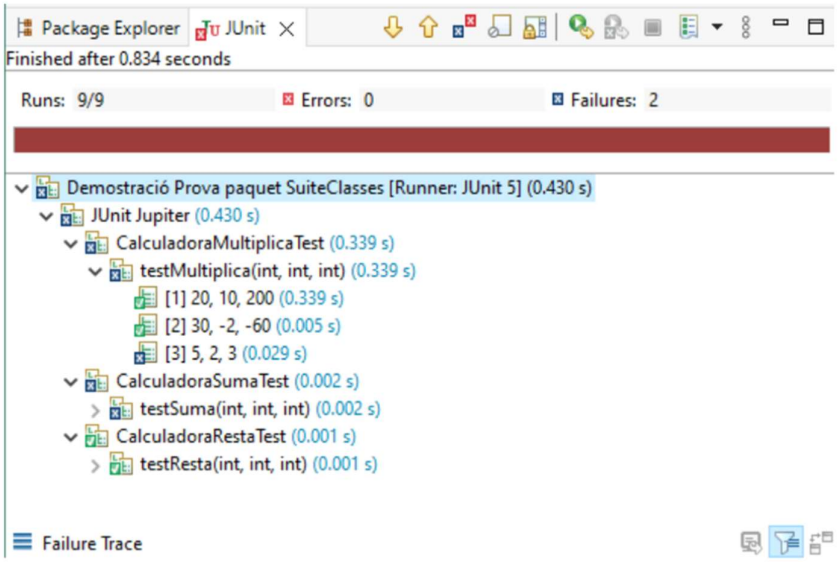
```java
package SuiteTests;
//import org.junit.platform.suite.api.IncludeClassNamePatterns;
import org.junit.platform.suite.api.SelectPackages;
import org.junit.platform.suite.api.Suite;
import org.junit.platform.suite.api.SuiteDisplayName;

@Suite
@SuiteDisplayName("Demostració Prova paquet SuiteClasses")
@SelectPackages("SuiteClasses")
//@IncludeClassNamePatterns(".*Test")

public class ProvaPaquet {
}
```

L'execució de les suites de proves es mostra a continuació. A cada prova apareix marcat el nom que se li ha assignat amb l'anotació *@SuiteDisplayName()*.

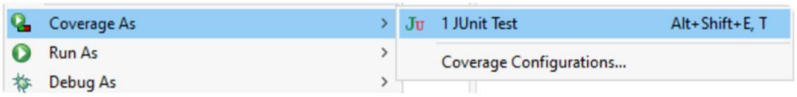




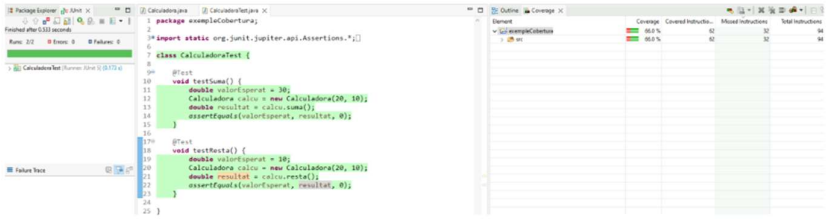
7.6.8 Mesurament de la cobertura del codi

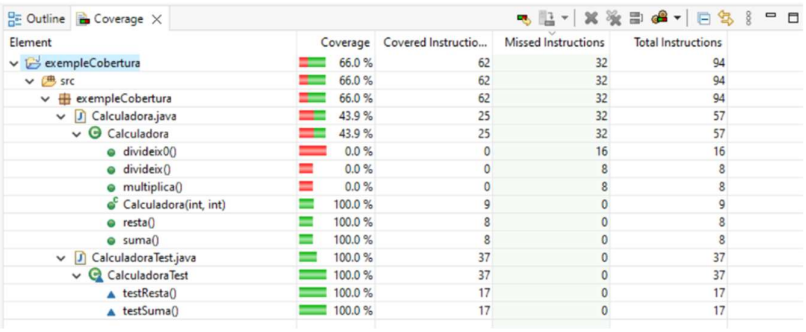
Una vegada que hem executat els casos de prova i hem vist que funcionen correctament, tenim la seguretat que els nostres casos de prova cobreixen totes les branques d'execució del programa?

Des de l'entorn Eclipse podem visualitzar la cobertura del codi dels tests unitaris que hem fet emprant l'eina **EclEmma**. Es pot accedir a aquesta opció des del menú ***Run >> Coverage***, o des de la barra d'eines prement en el botó ***Coverage*** . La cobertura de codi és el mètode d'anàlisi que determina quines parts del programari han estat executades pels casos de prova i quines parts no han estat executades. Per a aquest exemple partim de la classe *Calculadora* inicial i *CalculadoraTest* que només prova els mètodes *suma()* i *resta()*. Per a utilitzar la cobertura executem el cas de prova seleccionant ***Coverage As >> JUnit Test***.



Es mostrarà una finestra, la pestanya ***Coverage***, generalment al costat de la finestra de consola mostrant els resultats. Es pot anar desglossant la icona de cada classe per a saber quines parts d'aquesta han estat provades.

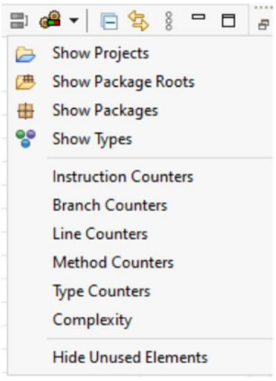




Les columnes mostrades en el resum ens donen informació sobre la cobertura dels mètodes:

- ***Coverage***: percentatge cobert de la prova. En vermell es mostra el que no s'ha provat, en verd el que sí s'ha provat. En la classe *Calculadora* s'observa un 66%, la qual cosa indica que les proves només abasten el 66% de les instruccions de la classe.
- ***Covered Instructions***: instruccions cobertes per la prova. De tot el paquet són 62. Podem veure desglosat el detall, 25 es corresponen a instruccions dels mètodes suma i resta de la classe *Calculadora* i 37 de la classe *CalculadoraTest*.
- ***Missed Instructions***: instruccions no cobertes per la prova. En aquest cas han estat 32, totes elles dels mètodes no provats de la classe *Calculadora*
- ***Total Instructions***: ens diu el nombre total d'instruccions de cada classe i del paquet, que en aquest cas són 94.

Prement el botó *View Menu* es pot seleccionar el tipus de cobertura a mostrar: d'instruccions, branques de codi com (if, switch), línies de codi, mètodes, tipus o de complexitat ciclomática.



El codi font de les classes apareixerà acolorit, tant la classe a provar com el test de prova. El color verd indica si s'ha executat totalment, groc si s'ha executat parcialment i vermell si no s'ha executat. Un cas de prova que falla es marca com no provat (en vermell), ja que no ha donat el resultat esperat.

Calculadora.java | CalculadoraTest.java

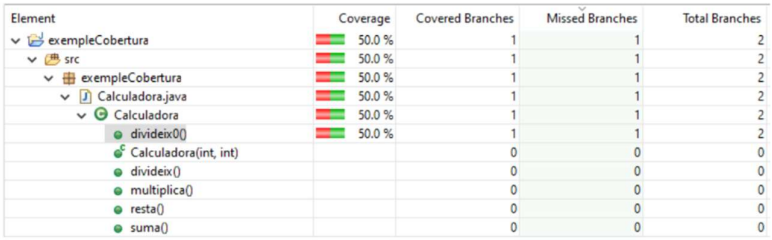
```
1  package exempleCobertura;
2
3  public class Calculadora {
4      private int num1;
5      private int num2;
6
7      public Calculadora(int a, int b) {
8          num1 = a;
9          num2 = b;
10     }
11     public int suma() {
12         int resul = num1 + num2;
13         return resul;
14     }
15     public int resta() {
16         int resul = num1 - num2;
17         return resul;
18     }
19     public int multiplica() {
20         int resul = num1 * num2;
21         return resul;
22     }
23     public int divideix() {
24         int resul = num1 / num2;
25         return resul;
26     }
27
28     public int divideix0() {
29         if(num2 == 0)
30             throw new java.lang.ArithmeticException("Divisió per 0");
31         else{
32             int resul= num1 / num2;
33             return resul;
34         }
35     }
36 }//
```

Afegim a la classe *CalculadoraTest* el següent mètode per provar la divisió per 0:

```
	@Test
	public void testDivideix0() {
		Calculadora calcu = new Calculadora(20, 0);
		Exception exception = assertThrows(ArithmeticException.class,
() -> calcu.divideix0());
		assertEquals("Divisió per 0", exception.getMessage());
	}
```

Seguidament, tornem a executar de nou la cobertura. Observem els colors en el mètode divideix0(). I triem Branch Counters des de View Menu de la pestanya de Coverage per veure el resum per branques de codi.

```
28     public int divideix0() {
29         if(num2 == 0)
30             throw new java.lang.ArithmeticException("Divisió per 0");
31         else{
32             int resul= num1 / num2;
33             return resul;
34         }
35     }
```



Element	Coverage	Covered Branches	Missed Branches	Total Branches
exempleCobertura	50.0 %	1	1	2
src	50.0 %	1	1	2
exempleCobertura	50.0 %	1	1	2
Calculadora.java	50.0 %	1	1	2
Calculadora	50.0 %	1	1	2
divideix0()	50.0 %	1	1	2
Calculadora(int, int)		0	0	0
divideix()		0	0	0
multiplica()		0	0	0
resta()		0	0	0
suma()		0	0	0

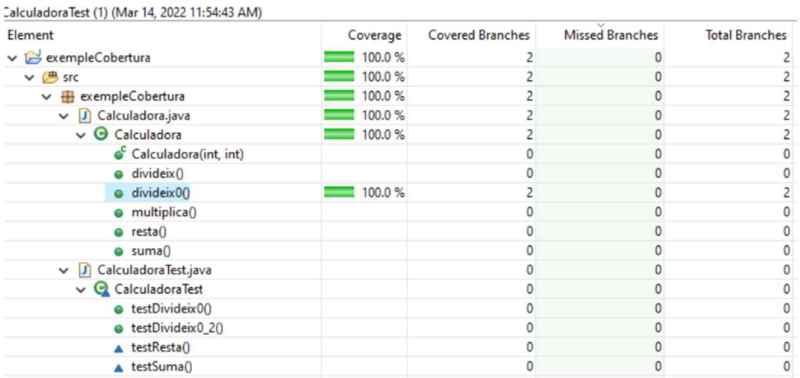
En la classe *Calculadora* es mostra la línia de l'if en color groc i amb un diamant a l'esquerra, que significa **cobertura parcial**. És a dir, només s'ha executat una part de la branca de l'if. El percentatge de cobertura en el mètode és del 50%, ja que només s'ha provat una branca.

Per a executar les dues branques hauríem d'afegir un nou cas de prova que provi el mètode *divideix0()* fent que es provi la branca de l'else. Per exemple:

```
	@Test
	public void testDivideix0_2() {
		double valorEsperat = 2;
		Calculadora calcu = new Calculadora(20, 10); double resultat =
calcu.divideix0();
		assertEquals(valorEsperat, resultat, 0);
	}
```

Una vegada afegit el cas de prova executem de nou la cobertura. En aquest cas el mètode *divideix0()* apareix de color verd i la línia de l'if amb un diamant verd a la seva esquerra indicant **cobertura total**. És a dir, totes les branques s'han executat. El percentatge de cobertura és del 100%.

```
28   public int divideix0() {
29       if(num2 == 0)
30          throw new java.lang.ArithmeticException("Divisió per 0");
31       else{
32          int resul= num1 / num2;
33          return resul;
34       }
35    }
```



alculadoraTest (1) (Mar 14, 2022 11:54:43 AM)

Element	Coverage	Covered Branches	Missed Branches	Total Branches
exempleCobertura	100.0 %	2	0	2
src	100.0 %	2	0	2
exempleCobertura	100.0 %	2	0	2
Calculadora.java	100.0 %	2	0	2
Calculadora	100.0 %	2	0	2
Calculadora(int, int)		0	0	0
divideix()		0	0	0
divideix0()	100.0 %	2	0	2
multiplica()		0	0	0
resta()		0	0	0
suma()		0	0	0
CalculadoraTest.java		0	0	0
CalculadoraTest		0	0	0
testDivideix0()		0	0	0
testDivideix0_2()		0	0	0
testResta()		0	0	0
testSuma()		0	0	0