



T5. Diseño y realización de pruebas

ENTORNO DE DESARROLLO

I.E.S. Luis Vives - Desarrollo de Aplicaciones Web

Curso 2023/2024

Introducción

Es prácticamente imposible realizar pruebas exhaustivas a un programa.

Las pruebas generalmente **son** demasiado **costosas**. Salvo que el programa sea tan importante como para realizarlas, lo que se hace es llegar a un punto intermedio en el cual se garantiza que no va a haber defectos importantes o muchos defectos y la **aplicación** está completamente operativa con un **funcionamiento aceptable**.

El objetivo de las pruebas es convencer, tanto a los usuarios como a los propios desarrolladores, de que el software es lo suficientemente robusto como para poder trabajar con él de forma productiva. Cuando un software supera unas pruebas exhaustivas, las probabilidades de que ese software dé problemas en producción se atenúan y, por tanto, su fiabilidad aumenta.

"Las pruebas solo pueden demostrar la presencia de errores, no su ausencia". Edsger Dijkstra.

Un error de software, error o simplemente fallo (también conocido por el inglés, bug) es un **problema en un programa de computador o sistema de software que desencadena un resultado indeseado**.

Testing Sw: <https://profile.es/blog/que-es-el-testing-de-software/>

Coste de los errores: <https://fastercapital.com/es/contenido/Costo-del-software--el-verdadero-costo-de-los-errores-de-software.html>

Tipos de errores:

https://es.wikipedia.org/wiki/Error_de_software#Errores_de_programaci%C3%B3n_comunes

Errores comunes: <https://geekqa.net/8-errores-comunes-que-cometen-los-testers-cuando-empiezan-en-qa/>

Procedimientos de pruebas

Un procedimiento de prueba es la definición del objetivo que desea conseguirse con las pruebas, qué es lo que va a probarse y cómo.

El objetivo de las pruebas **no siempre es detectar errores**. Muchas veces lo que quiere conseguirse es que el sistema ofrezca **un rendimiento** determinado, que la **interfaz** tenga una apariencia y cumpla unas **características** determinadas, etc. Por lo tanto, la ausencia de errores en las pruebas nunca significa que el software las supere, pues hay muchos parámetros en juego.

Cuando se diseñan los procedimientos, se deciden las personas que hacen las pruebas y bajo qué parámetros van a realizarse. **No siempre tienen que ser los programadores** los que hacen las pruebas. No obstante, siempre tiene que haber personal externo al equipo de desarrollo, puesto que los propios programadores solo prueban las cosas que funcionan (si supieran dónde están los errores, los corregirían).

Hay que tener en cuenta que es imposible probar todo, la prueba exhaustiva no existe. Muchos errores del sistema saldrán en producción cuando el software ya esté implantado, pero siempre se intentará que sea el mínimo número de ellos.

En los planes de pruebas (es un documento que detalla en profundidad las pruebas que se vayan a realizar), generalmente, se cubren los siguientes aspectos:

1. **Introducción.** Breve introducción del sistema describiendo objetivos, estrategia, etc.
2. **Módulos** o partes del software por probar. Detallar cada una de estas partes o módulos.
3. **Características** del software por **probar**. Tanto individuales como conjuntos de ellas.
4. **Características** del software que **no** ha de **probarse**.
5. **Enfoque** de las pruebas. En el que se detallan, entre otros, las personas **responsables**, la planificación, la **duración**, etc.
6. **Criterios** de **validez** o invalidez del software. En este apartado, se registra cuando el software puede darse como válido o como inválido especificando claramente los criterios.
7. **Proceso** de pruebas. Se especificará el proceso y los procedimientos de las pruebas por ejecutar.
8. **Requerimientos** del entorno. Incluyendo niveles de seguridad, comunicaciones, necesidades hardware y software, herramientas, etc.
9. **Homologación** o aprobación del plan. Este plan deberá estar firmado por los interesados o sus responsables.

Las demás fases del proceso de pruebas, como puede entenderse, son el mero desarrollo del plan de pruebas anterior.

Casos de pruebas

En la fase de pruebas, se diseñan y preparan los casos de prueba, que se crean con el objetivo de encontrar fallos. Por experiencia, no hay que probar los programas de forma redundante. Si se prueba un software y funciona, la mayoría de las veces no hace falta probar lo mismo. Hay que crear otro tipo de pruebas, no repetirlas. Lo que no implica que ante un **cambio de software se ejecuten todas las pruebas otra vez**, para verificar que no hay ningún problema con los cambios introducidos.

Que las pruebas tengan que hacerse en la fase de pruebas no implica que tengan que hacerse después de la fase de desarrollo. Conceptos como [TDD-Test Driven Development](#), están cambiando la concepción de cuando se inicia la fase de pruebas. Hay lenguajes, como [Gherkin](#), pensados específicamente para captar las necesidades de usuario

Hay que tener en cuenta que la prueba no debe ser **muy sencilla ni muy compleja**. Si es muy sencilla, no va a aportar nada y, si es muy compleja, quizá, sea difícil encontrar el origen de los errores.

Como se ha observado, las pruebas solo encuentran o tratan de encontrar aquellos errores que van buscando, luego, es muy importante realizar un buen diseño de las pruebas con buenos casos de prueba, puesto que se aumenta de esta manera la probabilidad de encontrar fallos.

| | | | |
|---|---|---|-----------------|
| Nombre del proyecto: Acceso a web | | ID caso de prueba: 001 | |
| Autor: José Manuel | | Fecha: 09/02/2023 | |
| Propósito: | | | |
| Verificar el proceso de <u>login</u> correcto en la web | | | |
| Acciones | | | |
| # | Acciones | Salida Esperada | Salida Obtenida |
| 1 | Acceder al formulario de <u>login</u> e introducir los datos de usuario y contraseña registrados en la BBDD | Los datos se envían al servidor | |
| 2 | Servidor verifica los datos de usuario y contraseña y devuelve ok | El servidor procesa la petición y devuelve ok | |
| 3 | La web redirige al usuario a la página principal | El usuario accede a la sesión en la web | |
| | | | |
| | | | |
| | | | |
| | | | |
| Resultados obtenidos | | | |
| Resultado: Correcto | | | |
| Seguimiento: | | Severidad: | |
| Evidencia: | | | |
| | | | |

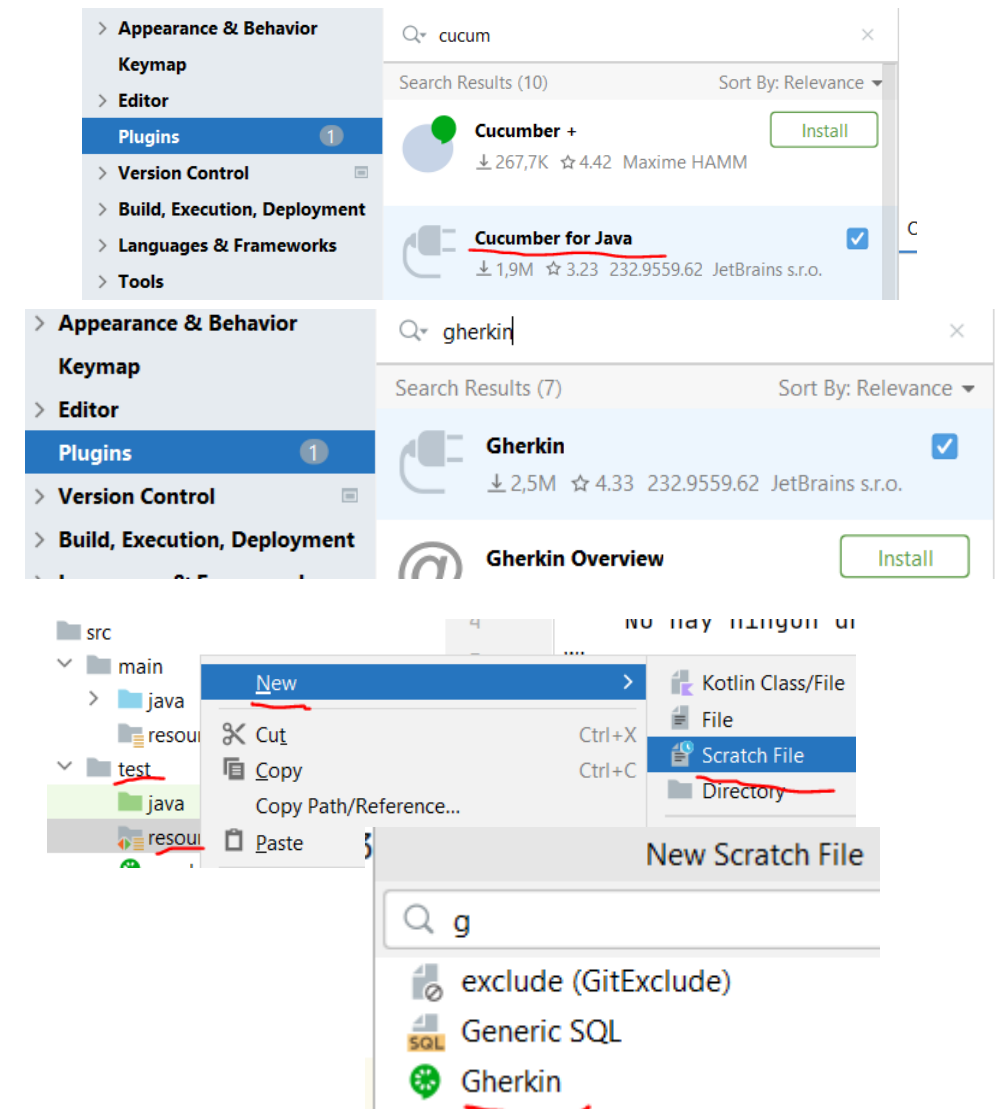
En general la estructura del diseño de una prueba tiene 3 elementos

- **Given:** datos de entrada . Viene del UI
- **Then:** evento, acción. Viene de la capa de servicios
- **When:** salida. Es el resultado, lo que devuelvo

Puedes instalar el plugin de [Cucumber for java](#) y [Gherkin](#) y Test Automaten intelliJ y crear los ficheros (con extensión **.feature** en test/resources, ej. login.feature) con la descripción de las pruebas en **test/resources/features**. Con Gherkin defines los tests como se indica a la dcha. y Cucumber permitiría generar el código java para los test.

También puedes [crear escenarios para varios casos](#)

Es posible generar las clases java automáticamente a partir de los ficheros creados. <https://www.youtube.com/watch?v=7VzRyJVKFXc>



Ejemplos

Scenario: El precio ha sido mal introducido

Given lista de productos

#{té, 25\$, 150}

#{azúcar, 4\$, 500}

When introduzco nuevo producto {sal, siete\$,750}

Then me da un error que el precio no está bien escrito

And te sale un mensaje así

#Lo sentimos el precio está escrito en un formato inválido, por favor inténtelo de nuevo

Puedes escribir en otros idiomas :
<https://cucumber.io/docs/gherkin/languages/>

Curso de cucumber:
<https://www.youtube.com/watch?v=G1DOhBMIFkI&list=PLHBdINTbF1h5XwqAc18Z5xcJpbO6rq6mm>

```
i.feature x  cambiarApellidoClienteAdministrador.feature  agregarNuevoProductoAdministrador.feature
Feature: Como administrador quiero agregar nuevo producto

Scenario: Agrego el producto con éxito
  Given lista de productos
    #{té, 25$, 150}
    #{azúcar, 4$, 500}

  When introduzco nuevo producto {sal, 4$,750}
  Then me sale mensaje que ha sido un éxito
  And sale lista de productos así
    #{té, 25$, 150}
    #{azúcar, 4$, 500}
    #{sal, 4$,750}

Scenario: El producto ya existe
  Given lista de productos
    #{té, 25$, 150}
    #{azúcar, 4$, 500}

  When introduzco nuevo producto {té, 20$, 750}
  Then me da un error que el producto ya existe
  And te sale un mensaje así
    #Lo sentimos no se pudo agregar el producto por que ya existe.

Scenario: El precio ha sido mal introducido
```

Actividad 1. Crea la plantilla para el caso de prueba de dar de alta, vía web, un dni en el servidor, previa verificación de datos básicos del mismo. Se dará un mensaje de ok o error al usuario.

Realiza los casos de prueba utilizando Gherkin /Cucumber

Codificación y ejecución de las pruebas

Una vez **diseñados los casos de prueba**, hay que **generar las condiciones** necesarias para poder ejecutar dichos casos de prueba. Habrá que **codificarlos** en muchos casos **generando set o conjuntos de datos**. En estos set de datos hay que incluir tanto datos válidos e inválidos como algunos datos fuera de rango o disparatados.

También habrá que **preparar las máquinas** sobre las que van a hacerse las pruebas instalando el software necesario, los usuarios de sistema, realizar carga del sistema, etc.

Una vez definidos los casos de prueba y establecido el entorno de las pruebas, es el momento de su ejecución. Irán **ejecutándose los casos de prueba uno a uno** y, cuando se detecte algún **error**, hay que aislarlo y **anotar la acción** que estaba probándose, el caso, el módulo, la fecha, la hora, los datos utilizados, etc. De esa manera, intentará **documentarse** lo más **detalladamente** posible el error. En el caso de que se produzcan errores aleatorios, también hay que registrarlos anotando este hecho.

Tipos de pruebas

FUNCIONALES, ESTRUCTURALES Y REGRESIÓN.

Ya conocemos qué son las pruebas y qué objetivos tienen. En cuanto al tipo de pruebas por realizar, existen muchas categorías. A continuación, se repasan las más frecuentes.

En primer lugar, existen las **pruebas funcionales**, que, como su nombre indica, buscan que los **componentes software** diseñados **cumplan** con **la función** con la que fueron diseñados y desarrollados. Estas pruebas buscan lo que el sistema hace, más que cómo lo hace. Todos los sistemas tienen una serie de funcionalidades o características y esas son las que van a testearse.

Las **pruebas no funcionales** son aquellas pruebas más técnicas que se realizan al sistema. Suelen ser pruebas no funcionales las pruebas de **carga y estrés**, pruebas de **seguridad**, pruebas de **rendimiento**, pruebas de **fiabilidad**, etc.

Entre las pruebas que examinan de forma más detallada la arquitectura de la aplicación, están las **pruebas estructurales**, puesto que, en algún momento, se utilizan técnicas de análisis del código. Generalmente, para este tipo de pruebas, se utilizan herramientas especializadas.

Otro tipo de pruebas son las pruebas de regresión o **pruebas repetidas debido a un cambio**. No suele probarse lo que ya se ha probado, pero, en el caso de que el software haya sido modificado, generalmente, se realiza este tipo de pruebas. Estas pruebas intentan descubrir si existe algún error tras las modificaciones o si se encuentra algún tipo de problema que no se había descubierto previamente. Solamente se realizarán estas pruebas en el caso de que haya una modificación de software. Este tipo de pruebas de regresión suelen automatizarse y se agrupan en conjuntos llamados *regression test suites* (conjuntos de pruebas de regresión).

Tipos de pruebas

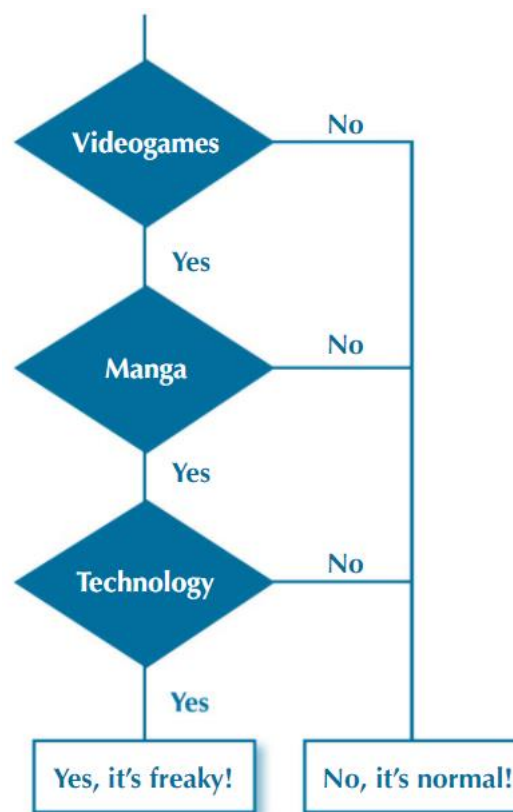
CAJA BLANCA Y CAJA NEGRA.

Pruebas de caja blanca En las pruebas de caja blanca, se conoce o se tiene en cuenta el código que quiere probarse. Se denomina también *clear box testing* porque la persona que realiza las pruebas está en contacto con el código fuente. Su objetivo es probar el código, cada uno de sus elementos. Existen algunas clases de pruebas de este tipo como, por ejemplo:

- Pruebas de cubrimiento.
- Pruebas de condiciones.
- Pruebas de bucles.

Pruebas de cubrimiento. En este tipo de pruebas, el objetivo es ejecutar, **al menos una vez, todas las sentencias** o líneas del programa. Para realizar las pruebas, habrá que generar el suficiente número de casos de prueba para poder cubrir los distintos caminos independientes del código. En cada condición, deberá cumplirse en un caso y en otro no.

```
int isFreaky(int videogames,
             int manga, int technology){
    if(videogames>0){
        if(manga>0){
            if(technology>0){
                return 1;
            }
            else{
                return 0;
            }
        }
        else{
            return 0;
        }
    }
    else{
        return 0;
    }
}
```



Pruebas de condiciones. En este caso, se necesitarán varios casos de prueba. En una condición, puede haber varias condiciones simples y habrá que generar **un caso de pruebas por cada operando lógico o comparación**. La idea es que, en cada expresión, se cumpla en un caso y en otro no.

```
if (videogames=1 && manga=1 && technology=1){ freaky = 1}
```

En el caso anterior, deberán comprobarse todas y **cada una de las combinaciones** de las tres variables anteriores. En esta ocasión, son variables, pero podrían ser otro tipo de expresiones más complejas. De esa manera, habrá que cerciorarse de que cualquiera de las combinaciones de valores de la condición funcionará tal y como el programa fue concebido.

Pruebas de bucles. Los bucles son estructuras que se basan en la repetición, por lo tanto, la prueba de bucles se basará en la repetición de un número especial de veces. En el caso de un bucle simple, los casos de prueba deberían contemplar lo siguiente:

- a) **Repetir el máximo**, máximo - 1 y máximo + 1 veces el bucle para ver si el resultado del código es el esperado.
- b) Repetir el bucle **cero y una vez**.
- c) Repetir el bucle un **número determinado** de veces.

En el caso de bucles anidados, existirán bucles internos y externos. Sería bueno realizar la prueba de bucles simple para los bucles internos ejecutando el bucle externo un número determinado de veces y, luego, realizar la prueba contraria. El bucle interno se ejecuta un número determinado de veces y el externo se prueba con las pruebas anteriores de bucle simple.

Pruebas de caja negra Entre las pruebas de caja negra (aquellas que simplemente prueban la interfaz sin tener en cuenta el código), pueden citarse las siguientes:

- Pruebas de **clases de equivalencia** de datos.
- Pruebas de **valores límites**.
- Pruebas de **interfaces**.
- Ref: <https://elminimoviable.es/ejemplos-de-pruebas-de-caja-negra/>
- Videos ilustrativos
 - <https://www.youtube.com/watch?v=PmdFMDZVmmM>
 - <https://www.youtube.com/watch?v=mlj2HDcnLBM>

Pruebas de clases de equivalencia de datos. Imagínese que está probándose una interfaz y debe generarse un código de usuario y una clave.

Ejem: El sistema dice que el código de usuario tendrá que tener mayúsculas **y** minúsculas, no puede tener caracteres que no sean alfabéticos y ha de tener, al menos, 6 letras (máximo 12). Las contraseñas tendrán, entre 8 y 10 caracteres y contendrán letras **y** números.

Para testear esta interfaz, lo que debe hacerse es establecer clases de equivalencia para cada uno de los campos. Tendrán que crearse **clases válidas y clases inválidas** por cada uno de los campos. Por ejemplo:

1. Usuario:
 - Clases válidas: cvu1:"Pelico" y cvu2:"RocinantesAb".
 - Clases inválidas: ciu1: "marruller044", ciu2: " nene", ciu3: "Portaavionesgigante", ciu4: "Z&aratustra" y cvi5: "Ventajos012"
2. Contraseña:
 - Clases válidas: cvc1:"5Entrevias" y cvc2:"s8brinoS".
 - Clases inválidas: cic1: "corta3" , cic2: "muyperoque3muylarguisima", cic3: "oletugarb" y cic4: "999999999".

El objetivo de esta prueba es comprobar todas las clases válidas y las inválidas al menos una vez.

Cada vez que se diseña un **caso de prueba con datos inválidos**, se introducirá solamente **una clase inválida**. De esa manera, se conocerá si el programa está funcionando correctamente.

Muchas veces, al utilizar varias clases inválidas, los errores se enmascaran y no puede conocerse si todas las clases funcionan.

| Caso de Prueba | Clase válida | clase inválida | salida esperada |
|--|--------------|----------------|------------------|
| ("Pelico", "5Entrevias") | cvu1, cvc1 | - | OK |
| ("RocinantesAb", "s8brinoS") | cvu2, cvc2 | - | OK |
| ("Pelico", "corta3") | cvu1, | cic1 | ERROR Contraseña |
| ("Pelico", "muyperoque3muylarguisima") | cvu1, | cic2 | ERROR Contraseña |
| ("Pelico", "Portaavionesgigante ") | cvu1, | cic3 | ERROR Contraseña |
| ("RocinantesAb", "999999999 ") | cvu2, | cic3 | ERRORContraseña |
| ("marruller044", "5Entrevias ") | ciu1, | cvc1 | ERROR Usuario |
| ("nene", "s8brinoS") | ciu2, | cvc2 | ERROR Usuario |
| | | | ... |

Pruebas de valores límites. Este tipo de pruebas son complementarias a las pruebas de particiones/equivalencia, buscando el error con unos pocos valores en los límites de los rangos. Para ello se utilizan valores que puedan probar si la interfaz y el programa funcionan correctamente en los límites de los valores válidos puesto que muchos errores se producen en dichos límites (al poner < en lugar de <=, por ejemplo)

Imagínese que se accede a la página web de un banco para testearla y la interfaz, cuando va a transferirse una cantidad, comunica: "La cifra máxima que usted puede transferir hoy es de 10.000 euros" . Si quiere probarse dicha interfaz, el tester probaría, por ejemplo, valores fuera de rango como -100 o 20.000; también valores en los límites como 0, 1, 9.999, 10.000 y 10.001, o valores típicos e intermedios como 9.000 o 2.500. El objeto de esta prueba se halla en que, muchas veces, los programadores se equivocan al establecer los límites en la frontera.

Si tomamos que la contraseña debe tener entre 6 y 10 caracteres probaríamos con contraseñas de 5,6,10 y 11 caracteres.

Pruebas de interfaces. Una interfaz de usuario o GUI (.), generalmente, se testea con una técnica que se denomina prueba de interfaces. Generalmente, una interfaz es una serie de objetos con una serie de propiedades. Toda esta serie de objetos con sus propiedades en su conjunto formarán la interfaz. Esos objetos van tomando valores durante la ejecución del programa. En esa ejecución, el usuario va introduciendo valores en la interfaz y haciendo clic sobre algunos objetos. Dependiendo de las entradas, la interfaz proporcionará una salida determinada. Esa salida debería ser la esperada. Dentro de las pruebas de interfaces se debería de verificar:

1. Testear la **funcionalidad de la interfaz**
2. Testear la **usabilidad**
3. Testear la **accesibilidad**

Testear la funcionalidad de la interfaz. Una **primera prueba** puede consistir en seguir el manual de usuario. El tester deberá introducir datos (mejor **datos reales** que inventados) como si se tratase del propio usuario y comprobar que las salidas proporcionadas son las esperadas. Si el software pasa esta prueba, entonces, podrá pasar a sufrir un testeo más serio utilizando casos de prueba.

La idea final de este tipo de pruebas es que no haya comportamientos indeseados al insertar datos o realizar operaciones, o combinación de estas, desde la interfaz de nuestra aplicación.

Pruebas de interfaz de aplicación web: <https://playwright.dev/> Se abre un navegador para hacer la prueba. Se puede ver o no. Puedes simular los clicks, la introducción de texto. Puedes hacer el acceso y que se vayan guardando todos los clicks que vas poniendo para que se cree el fichero de pruebas solo. [Playwright y Cypress](#) son los entornos que más se usan para este tipo de pruebas.

```
public static void primeraPrueba() {  
    try (Playwright playwright = Playwright.create()) {  
        Browser browser = playwright.chromium().launch();  
        Page page = browser.newPage();  
        page.navigate(url: "http://playwright.dev");  
  
        // Expect a title "to contain" a substring.  
        assertThat(page).hasTitle(Pattern.compile(regex: "Playw
```

```
@Test  
void shouldCheckTheBox() {  
    page.setContent("<input id='checkbox' type='checkbox'></input>");  
    page.locator(selector: "input").check();  
    assertTrue((Boolean) page.evaluate(expression: "() => window['checkbox'].checked"));  
}  
  
@Test  
void shouldSearchWiki() {  
    page.navigate(url: "https://www.wikipedia.org/");  
    page.locator(selector: "input[name='search']").click();  
    page.locator(selector: "input[name='search']").fill(value: "playwright");  
    page.locator(selector: "input[name='search']").press(key: "Enter");  
    assertEquals(expected: "https://en.wikipedia.org/wiki/Playwright", page.url());  
}
```


Testear la usabilidad. Tiene por objeto evaluar si el producto generado va a resultar lo esperado por el usuario. Hay que ver y trabajar con la interfaz desde el punto de vista del usuario. Además, en su testeo, deberían utilizarse datos reales. Solamente, al observar cómo el usuario interactúa con el software y escuchando su feedback, pueden detectarse aquellas características de este que lo hacen difícil y tedioso de utilizar.

Una vez detectadas esas disfunciones, se realizarán los cambios pertinentes, de tal manera que el software sea fácil de usar y eficiente. Es importante **escuchar la opinión del cliente** porque, a la postre, es la persona que va a trabajar de forma sistemática con el software. Los test de usabilidad deben estar integrados en el ciclo de vida de desarrollo del software. La usabilidad ha de tenerse en cuenta no solamente en el momento de realizar las pruebas, sino también en el momento de diseñar la interfaz. A la hora de diseñar la interfaz, habría que hacerse las siguientes preguntas:

- ¿Los **usuarios comprenderán cómo funciona la interfaz** de una manera sencilla? ¿Mensajes claros?
- ¿Es la interfaz lo suficientemente **rápido y eficiente** para el usuario?

Téngase en cuenta que, muchas veces, en las interfaces, se echan de menos teclas rápidas o combinaciones de teclas, valores por defecto, autocompletar, etc.

Testear la accesibilidad. En general, un software es accesible cuando el programa o aplicación se adecua a los **usuarios con discapacidad**, pueden hacer su trabajo de forma efectiva y la satisfacción con él es buena. Además, hay que tener en cuenta que, en ocasiones, hay estándares y requerimientos preestablecidos de accesibilidad que el software ha de cumplir.-

La accesibilidad no solamente es que el software esté diseñado para usuarios con discapacidad, sino que también **sea accesible por frameworks de test automatizados.**

Planificación de pruebas

La **planificación** de las pruebas es un punto importante en la toma de decisiones de un proyecto. Qué tipo de pruebas y cuándo van a realizarse son preguntas que hay que tener en cuenta desde el principio. Los tipos que veremos a continuación son:

- Pruebas **unitarias**
- Pruebas de **integración**
- Pruebas de **aceptación** o validación

Hay que tener en cuenta que muchas veces será necesario automatizar las pruebas o repetir las mismas pruebas tras realizar mantenimientos, modificaciones o correcciones del software. Es siempre bueno conservar los datos, set de pruebas, programas y módulos de prueba, puesto que no se sabe si van a ser necesarios en un futuro. En el caso de que se hayan utilizado herramientas u otro sistema, se documentará y se almacenarán en un repositorio para su **ejecución automática posterior**.

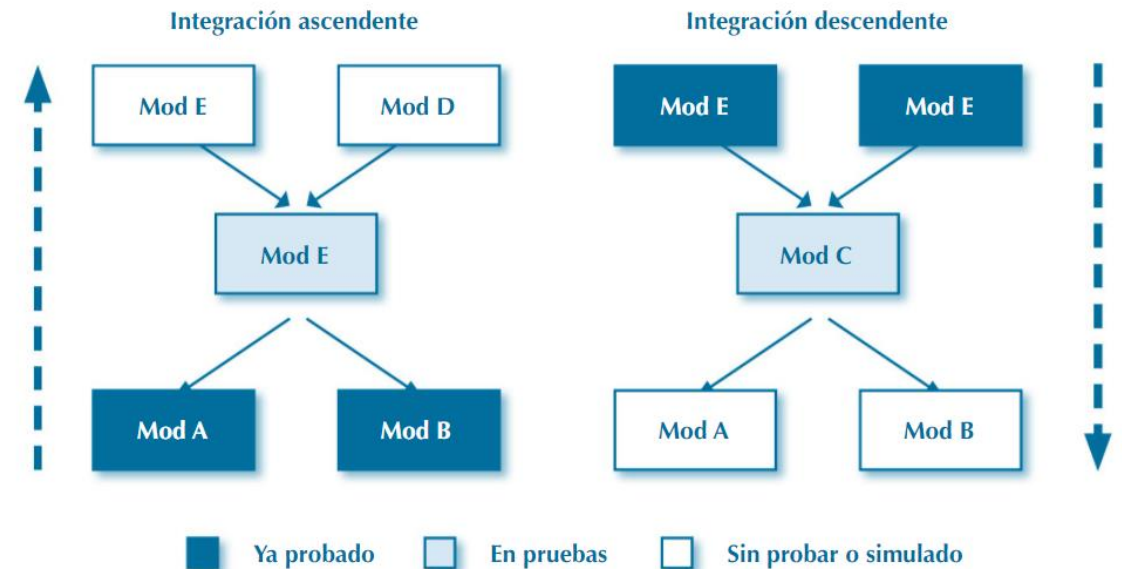
Suelen realizarse durante las primeras fases de diseño y desarrollo. Obviamente, no hay que demorar mucho la realización de dichas pruebas, puesto que luego hay que integrar todo el software (las distintas unidades) y los fallos van acumulándose y la localización y diagnóstico se complican. En el caso de la POO, las pruebas unitarias deberían probar **unitariamente cada método sin intervención de otras clases**, para ello se pueden usar frameworks de creación de **mocks** como son [mockito](https://mockito.org/) o Jmockit.

En la Programación Orientada a Objetos (POO) se llaman Mock a los objetos que imitan el comportamiento de objetos reales de una forma controlada.

<https://blog.softtek.com/es/testing-unitario>

Una vez que los componentes individuales se han probado, es momento de ir **integrando módulos**. Las pruebas de integración tendrán que hacerse al final de la fase de diseño (se realizarán pruebas para corroborar que el diseño es factible y eficiente) y también al final de la fase de codificación (una vez realizadas todas las pruebas individuales, se integran componentes y se prueban en conjunto verificando que funcionan correctamente de forma conjunta). Existen **pruebas de integración ascendentes y descendentes**. Pueden probarse los módulos más generales y, luego, ir a los más específicos o al contrario.

En las descendentes, generalmente, hay que hacer módulos de pruebas o programas de pruebas para probar unitariamente los módulos individuales, mientras que, en las ascendentes, muchas veces, hay que crear módulos, objetos y clases ficticias para probar partes más generales del programa. Habrá que crear [Mocks](#) (para comprobar el flujo) o [stubs](#) (para comprobar funcionalidad)



Este tipo de pruebas tratan de **probar el sistema completo**. Además de probar que los **requisitos** del programa se cumplen uno por uno, el equipo de pruebas mirará también si técnicamente el programa es **estable** y no tiene ningún fallo.

Además, habrá que probar el **rendimiento** del sistema modificando la carga y observando su evolución. También se harán **pruebas de estrés** para cerciorarse de que el sistema va a responder eficientemente ante cualquier eventualidad.

Existen en este estadio las :

- Pruebas Alfa: Se realizan en un **entorno controlado** y bajo unas especificaciones concretas. Por ejemplo realizadas por el usuario en el lugar del desarrollo, como si se tratara de un funcionamiento normal. El desarrollador irá registrando errores y problemas de uso.
- Pruebas Beta: Los **usuarios prueban** el sistema en un entorno no controlado por los desarrolladores. Por ejemplo, llevadas a cabo por el usuario en su lugar de trabajo, el desarrollador no esté presente. El usuario registra los problemas encontrados y se informan al desarrollador que tras las modificaciones prepara una nueva versión.

Está formada por un conjunto de pruebas con el objetivo de ejercitar profundamente el software:

★Prueba de **recuperación**: Se fuerza el fallo del software y se verifica que la recuperación se lleva a cabo apropiadamente.

★Prueba de **seguridad**: Intenta verificar que el sistema está protegido contra accesos ilegales.

★Prueba de **resistencia** (Stress): Enfrenta al sistema con situaciones de gran demanda de recursos: máximo de memoria, gran frecuencia de entrada de datos, problemas del sistema operativo virtual, etc.

PRUEBAS DE UNIDAD

I.E.S. Luis Vives - Desarrollo de Aplicaciones Web

Herramientas de testing

JUNIT [HTTPS://JUNIT.ORG/JUNIT5/](https://junit.org/junit5/)

[HTTPS://WWW.BAELDUNG.COM/JUNIT](https://www.baeldung.com/junit)

[JUNIT5 CHEATSHEET](#)

Definiciones pruebas de unidad

Documentación para las pruebas

Pruebas de código

Prueba del camino básico

Pruebas unitarias con Junit

- Creación de una clase de prueba
- Preparación y ejecución de las pruebas
- Tipos de anotaciones
- [Pruebas parametrizadas](#)
- Suite de pruebas

Pruebas unitarias con DBUnit

PRUEBAS DE UNIDAD

Pruebas orientadas a comprobar el funcionamiento de las unidades más pequeñas de un programa informático -> JAVA -> Clases (métodos).

Se prueba cada módulo para eliminar errores en la interfaz o en la lógica interna. Con pruebas de **caja blanca** y **caja negra se prueba:**

- ★ La interfaz del módulo, para asegurar el flujo correcto de información
- ★ Las estructuras de datos locales.
- ★ Las condiciones límite.
- ★ Todos los caminos independientes de la estructura de control.
- ★ Todos los caminos de manejo de errores.

DOCUMENTACIÓN PARA LAS PRUEBAS

El estándar **IEEE 829-1998** describe el conjunto de documentos que pueden producirse durante el proceso de prueba:

★ **Plan de Pruebas:** Alcance, enfoque, recursos y calendario de prueba.

★ **Especificaciones de prueba:**

- ✧ Diseño de la prueba: requisitos y criterios de **pasa-no pasa**.

- ✧ Casos de prueba: valores de entrada y resultados previstos.

- ✧ Procedimientos de prueba: pasos necesarios a realizar.

★ **Informes de pruebas:**

- ✧ Informe de elementos probados.

- ✧ Registro de la prueba (resultado de la ejecución de la prueba).

- ✧ Informe de incidencias de prueba (sucesos para ser investigados).

- ✧ Informe resumen de las actividades de prueba.

PRUEBAS de código (I)

Consisten en la ejecución del programa o parte de él con el objetivo de encontrar errores. Se parte de un conjunto de entradas y una serie de condiciones de ejecución; se observan y registran los resultados y se comparan con los resultados esperados.

Se observará si el comportamiento del programa es el previsto o no y por qué.

- ★ Pruebas de caja blanca: Pruebas del camino básico.

- ★ Pruebas de caja negra:

 - ✧ Clases de equivalencia.

 - ✧ Análisis del valor límite.

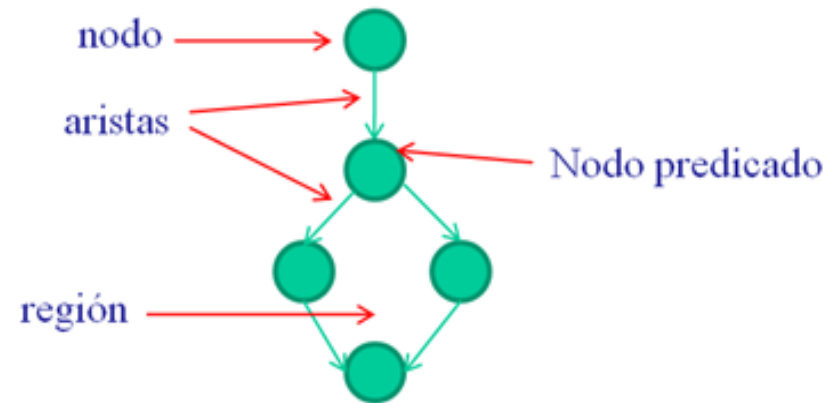
PRUEBA DEL CAMINO BÁSICO (i)

Es una técnica de prueba de caja blanca que permite al diseñador de casos de prueba obtener una medida de la complejidad de un diseño procedimental (**complejidad ciclomática**) y usar esa medida como guía para definir un conjunto de **caminos de ejecución**.

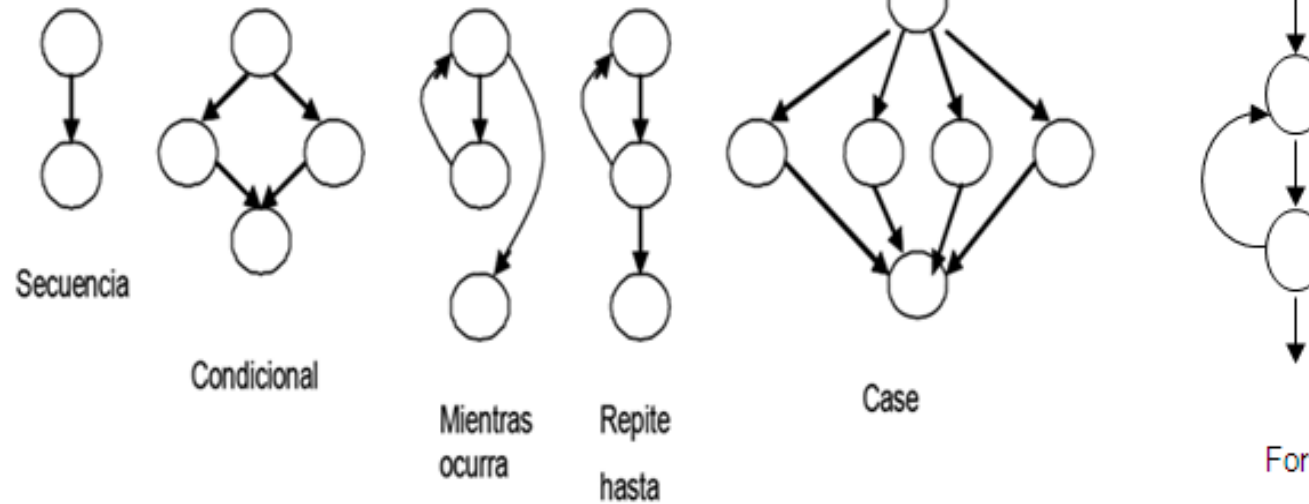
Los casos de prueba obtenidos garantizan que durante la prueba se ejecuta por lo menos una vez cada sentencia del programa.

Para estudiar la complejidad ciclomática se utilizan los **grafos de flujo** o **grafos de programa**.

PRUEBA DEL CAMINO BÁSICO (ii)



Estructuras en forma de grafo de flujo:



PRUEBA DEL CAMINO BÁSICO (III)

- ★ Dado un diagrama de flujo se numeran cada uno de los símbolos y los finales de las estructuras de control.
- ★ Cada **nodo** representa una o más sentencias.
- ★ Las **aristas** o **enlaces** representan el flujo de control. Una arista termina en un nodo.
- ★ Las áreas delimitadas por aristas y nodos se llaman **regiones**. El área exterior del grafo constituye otra región adicional.
- ★ El nodo que contiene una condición se llama nodo predicado y se caracteriza porque de él salen dos o más aristas.

PRUEBA DEL CAMINO BÁSICO (IV)

La complejidad ciclomática, $V(G)$, es una métrica del software que proporciona una medida cuantitativa de la complejidad lógica de un programa.

Establece el número de caminos independientes en la ejecución del programa y, por lo tanto, el número de casos de prueba que se deben ejecutar para asegurar que cada sentencia se ejecuta al menos una vez.

PRUEBA DEL CAMINO BÁSICO (V)

La complejidad ciclomática, $V(G)$, puede calcularse de tres formas:

- ★ $V(G)$ = número de regiones del grafo.
- ★ $V(G)$ = Aristas - Nodos + 2.
- ★ $V(G)$ = Nodos predicado + 1.

| Complejidad Ciclométrica | Riesgo |
|--------------------------|---|
| 1-10 | Programa simple, sin mucho riesgo |
| 11-20 | Programa medianamente complejo, riesgo moderado |
| 21-50 | Programa complejo, alto riesgo |
| > 50 | Programa no testeable, riesgo muy alto |

PRUEBA DEL CAMINO BÁSICO (VI)

El valor de $V(G)$ nos indica el número de caminos independientes del programa. Un **camino independiente** es cualquier camino del programa que introduce un nuevo conjunto de sentencias o una condición de proceso.

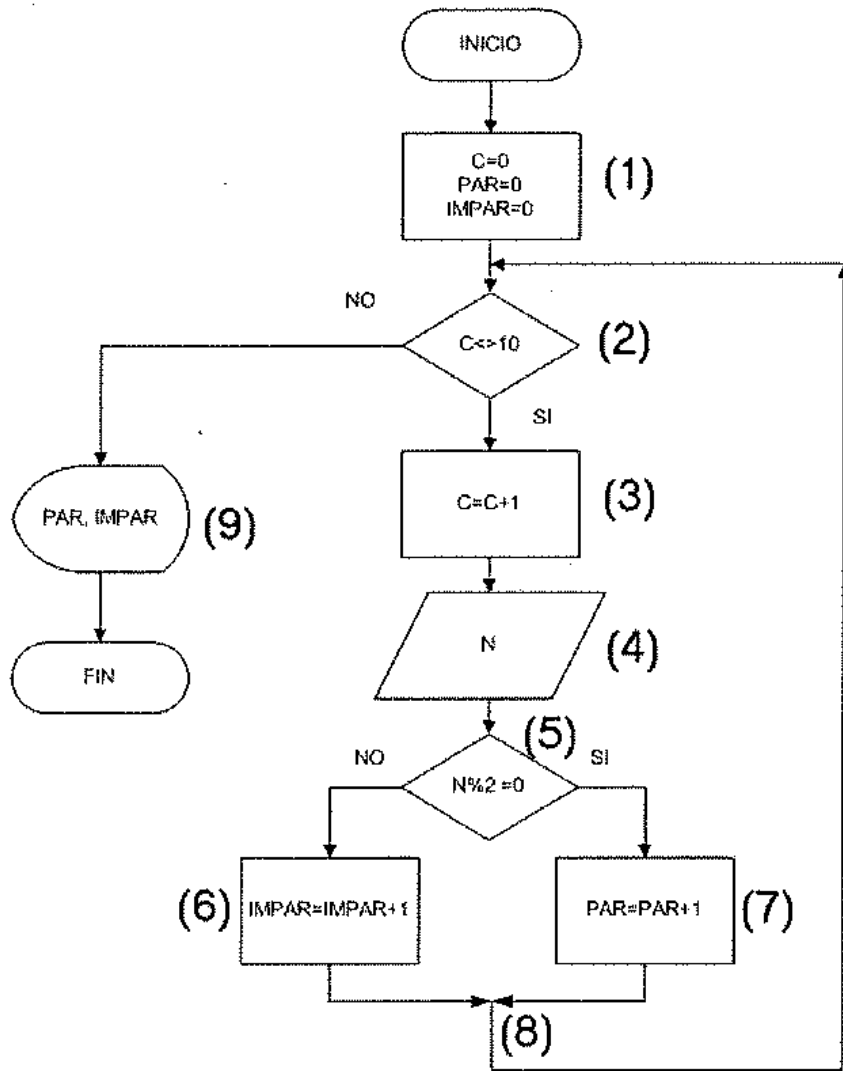
Una vez obtenidos los caminos independientes es preciso construir los casos de prueba que fuerzan la ejecución de cada camino.

PRUEBA DEL CAMINO BÁSICO (VII)

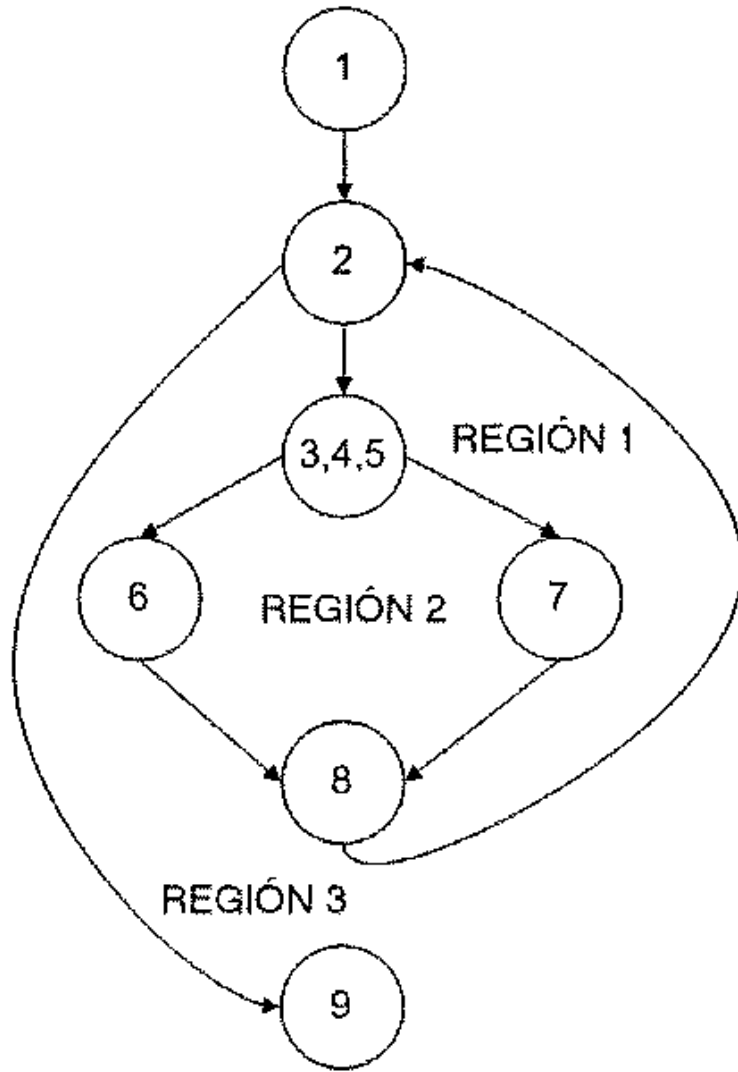
NOTAS:

- Una vez realizado el grafo, comenzar siempre por el camino más fácil. Si hay 2 caminos que atraviesan el mismo número de nodos, comenzar con el que menos líneas de código necesite para terminar.
- Una vez creado el primer camino, continuar creando caminos buscando **siempre añadir el menor número de aristas posible** en cada camino nuevo que se crea.
- Con cada camino nuevo, seleccionar los valores de entrada necesarios para probarlo, antes de continuar buscando el siguiente y comprobando que tiene sentido, que no rompa la lógica del programa.

PRUEBA DEL CAMINO BÁSICO. Ejemplo



PRUEBA DEL CAMINO BÁSICO. Ejemplo



PRUEBA DEL CAMINO BÁSICO. Ejemplo

Inicio

Abrir Fichero Alumnos
Leer Alumnos (Curso, Nombre, Sexo, Nota) (1)

Mientras Haya registros **Hacer** (2)

(3) NH = 0, NM = 0
Visualizar Curso (4) (5)

Mientras Haya registros y Mismo curso **Hacer**

Si sexo = "H" (6) Entonces

NH = NH + 1 (8)

Sino

NM = NM + 1 (7)

Fin si

Leer Alumnos (Curso, Nombre, Sexo, Nota) (9)

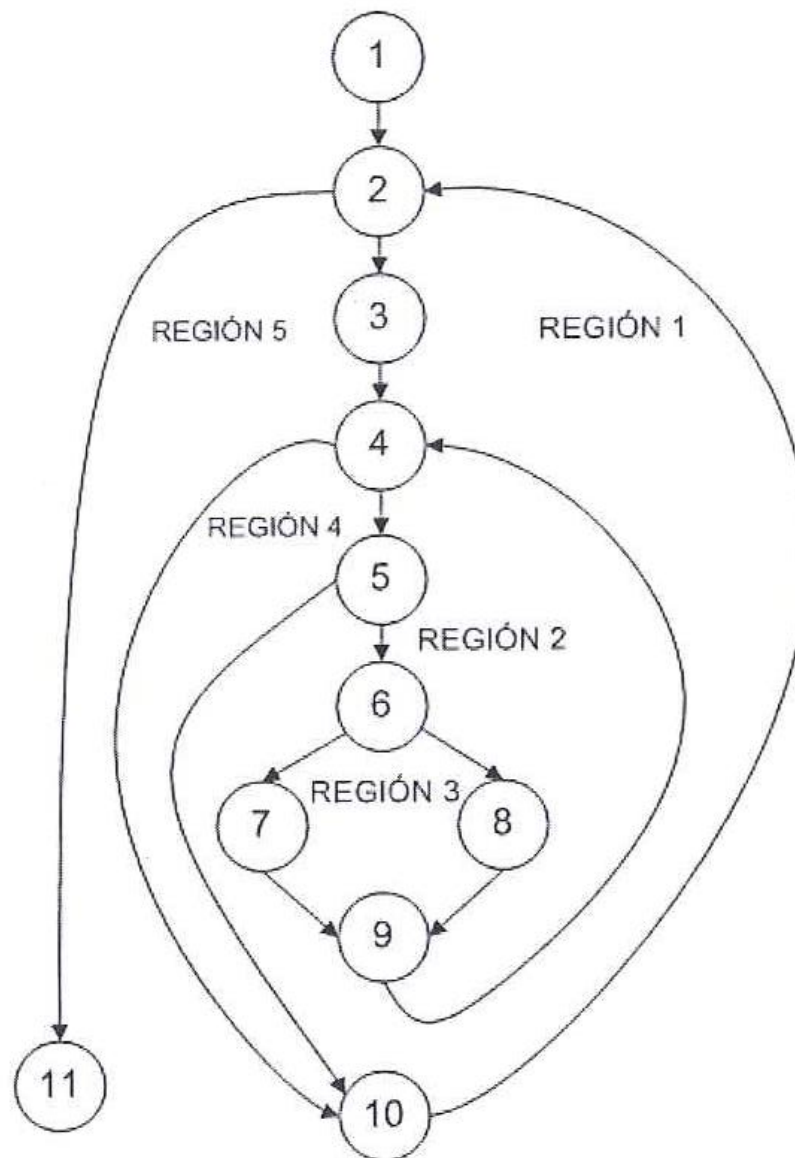
Fin Mientras

Visualizar NH, NM (10)

Fin Mientras

Abrir Fichero Alumnos (11)

Fin



PRUEBA DEL CAMINO BÁSICO. EJERCICIO

¿Regiones?

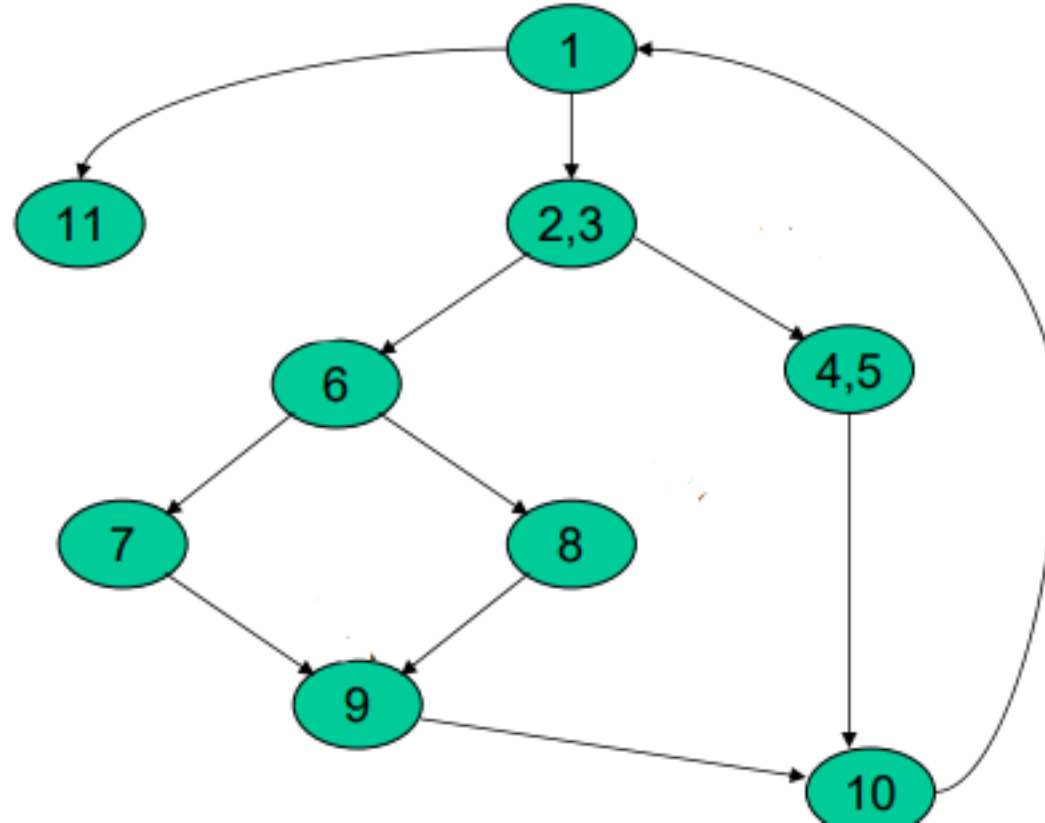
¿Complejidad ciclomática?

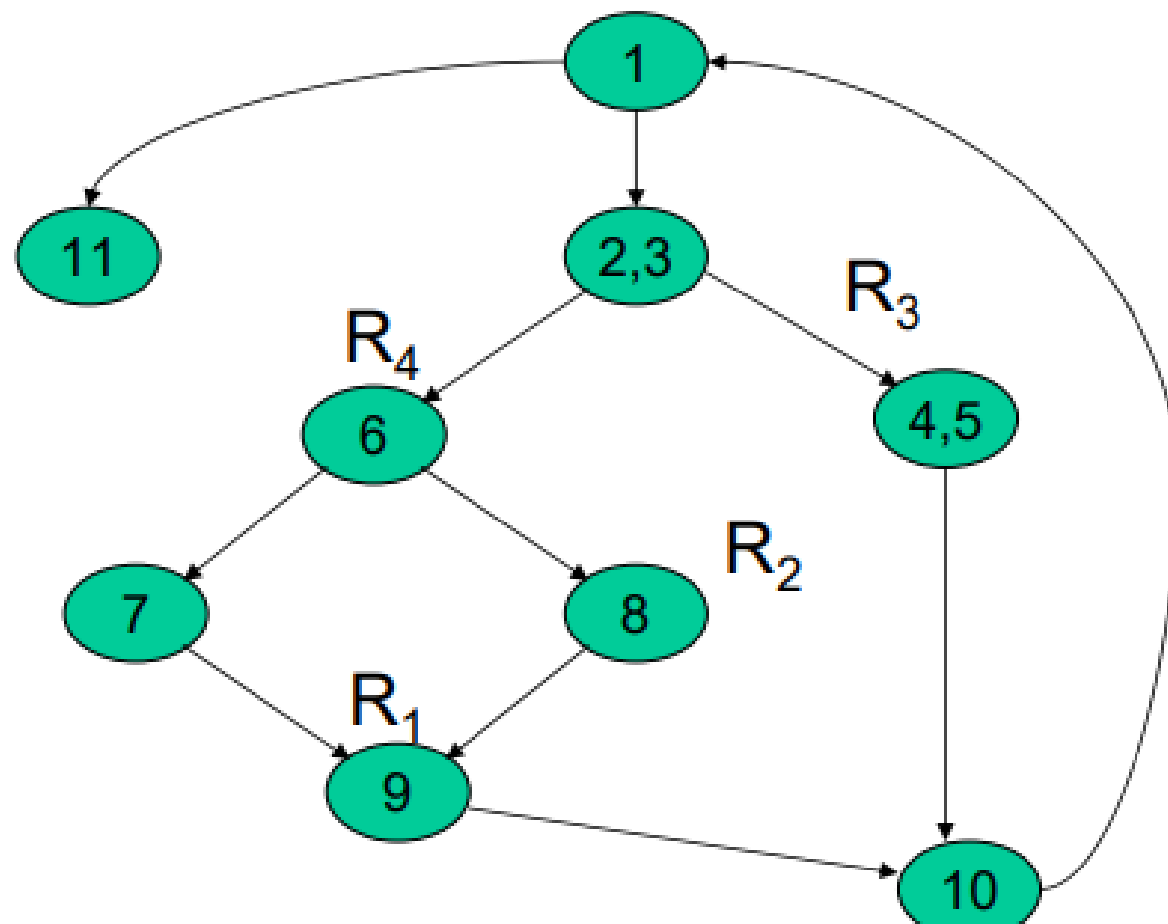
➤ N° de regiones

➤ Aristas – Nodos + 2

➤ Nodos predicados + 1

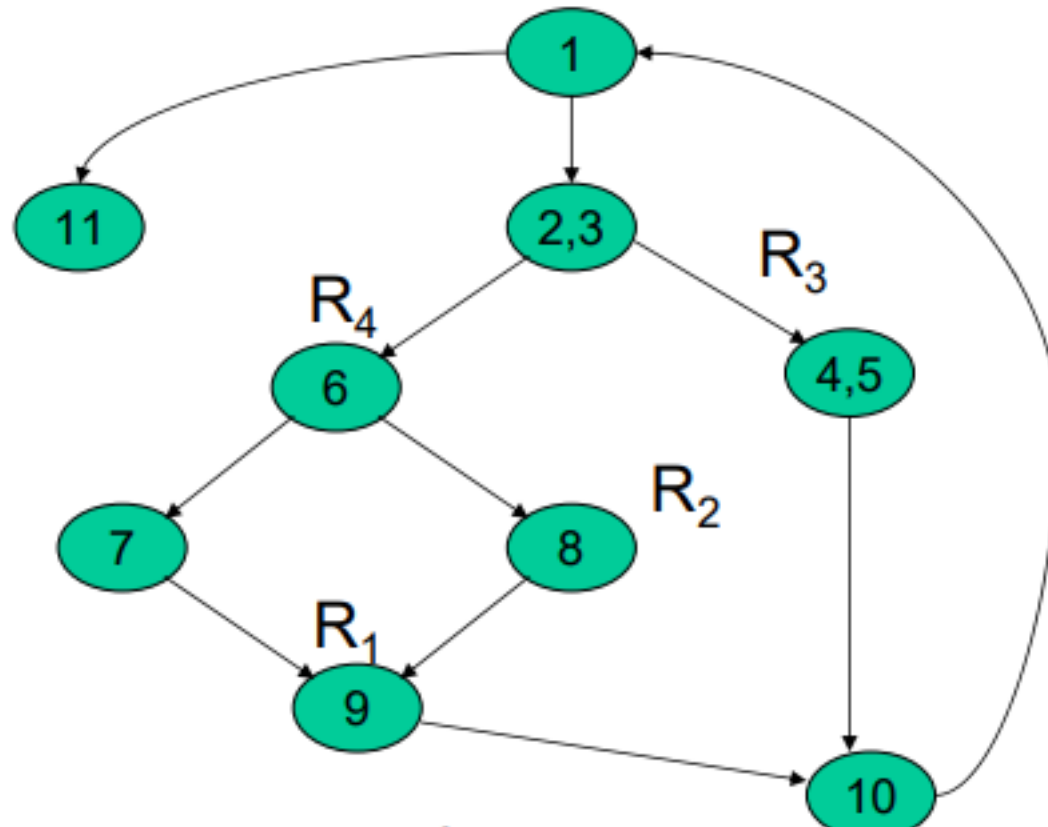
¿Caminos básicos?





- $V(G)=4$ Regiones
- $V(G)= 11A-9N+2=4$
- $V(G)=3NP+1=4$

PRUEBA DEL CAMINO BÁSICO. EJERCICIO

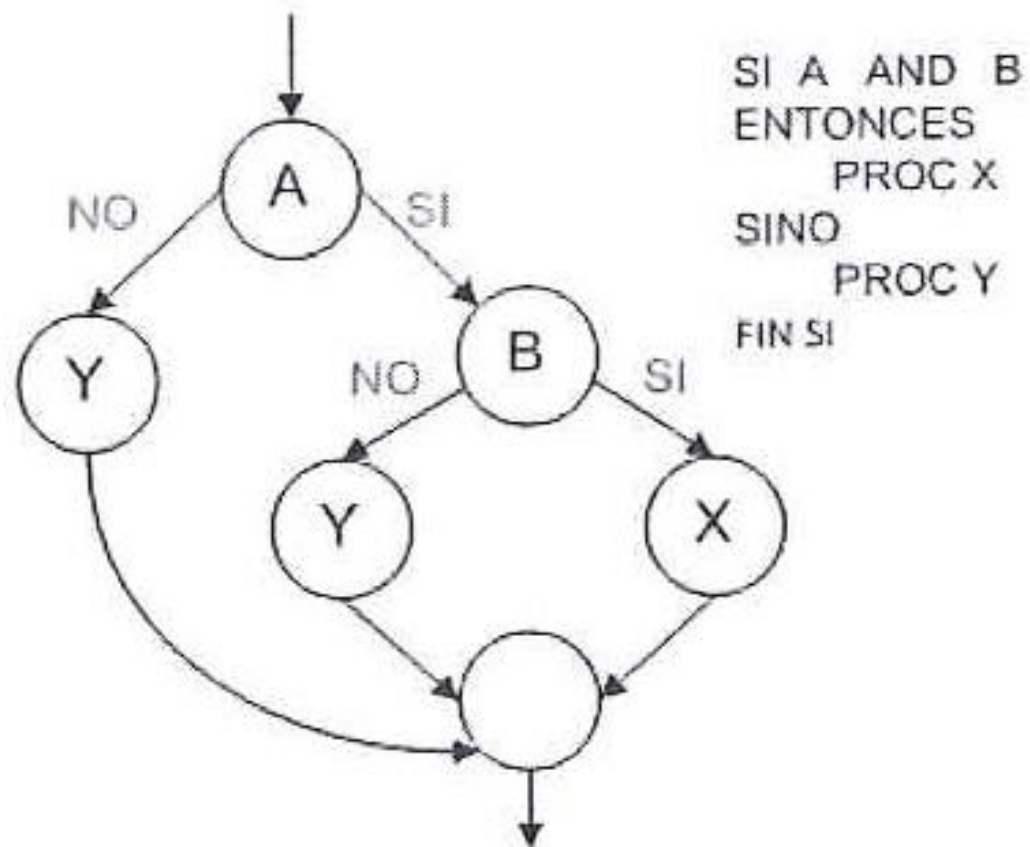
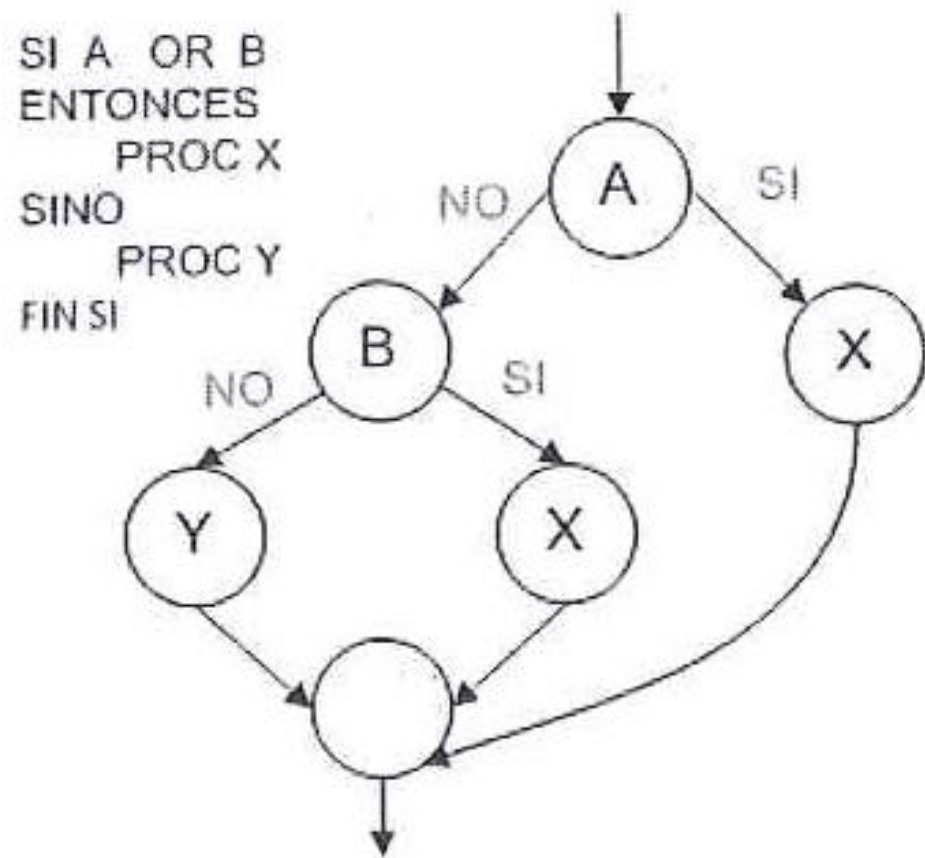


- $V(G)=4$ Regiones
- $V(G)= 11A-9N+2=4$
- $V(G)=3NP+1=4$

CAMINOS BÁSICOS:

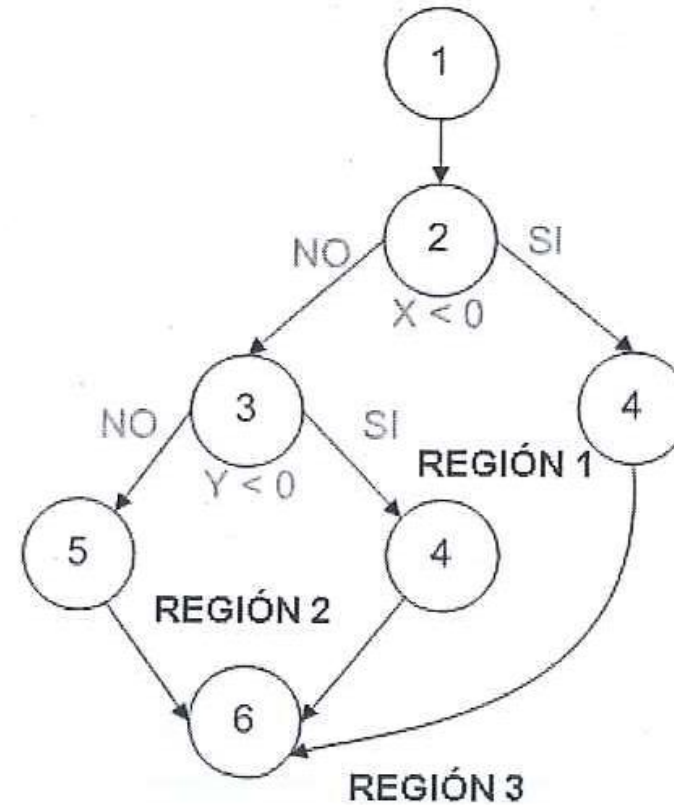
- Camino 1: 1-11
- Camino 2: 1-2-3-4-5-10-1-11
- Camino 3: 1-2-3-6-8-9-10-1-11
- Camino 4: 1-2-3-6-7-9-10-1-11

Camino básico. Lógica compuesta



PRUEBA DEL CAMINO BÁSICO. Ejemplo

```
static void visualizarMedia(float x, float y) {  
    float resultado = 0; (1)  
    if (x < 0 || y < 0) (3)  
    (2) System.out.println("X e Y deben ser positivos"); (4)  
    else {  
        resultado = (x + y) / 2;  
        System.out.println("La media es: " + resultado); (5)  
    }  
    (6)  
}
```



PRUEBA DEL CAMINO BÁSICO. Ejemplo

| Camino | Caso de prueba | Resultado esperado |
|------------------------|--|--------------------------------------|
| Camino 1: 1-2-3-5-6 | Escoger algún X e Y tal que NO se cumpla la condición $X < 0 \vee Y < 0$ $X=4, Y=5$ visualizarMedia(4,5) | Visualiza: La media es:4.5 |
| Camino 2: 1-2-4-6 | Escoger algún X e Y tal que SI se cumpla la condición $X < 0$ $X=-4, Y=5$ visualizarMedia(-4,5) | Visualiza: X e Y deben ser positivos |
| Camino 3: 1-2-3-4-6 | Escoger algún X e Y tal que SI se cumpla la condición $Y < 0$ $X=4, Y=-5$ visualizarMedia(4,-5) | Visualiza: X e Y deben ser positivos |

Pruebas unitarias con junit (I)

- ★ JUnit es una herramienta para realizar pruebas unitarias automatizadas.
- ★ Está integrada en Eclipse, por lo que no hay que instalar ningún plugin.
- ★ Las pruebas unitarias se realizan sobre una clase para probar su comportamiento de modo aislado del resto de clases de la aplicación.

Pruebas unitarias con junit (II)

Procedimiento:

- ★ Creación de una clase de prueba.
- ★ Preparación y ejecución de las pruebas.
- ★ Elección de los tipos de anotaciones.
- ★ Pruebas parametrizadas.
- ★ Suite de pruebas.

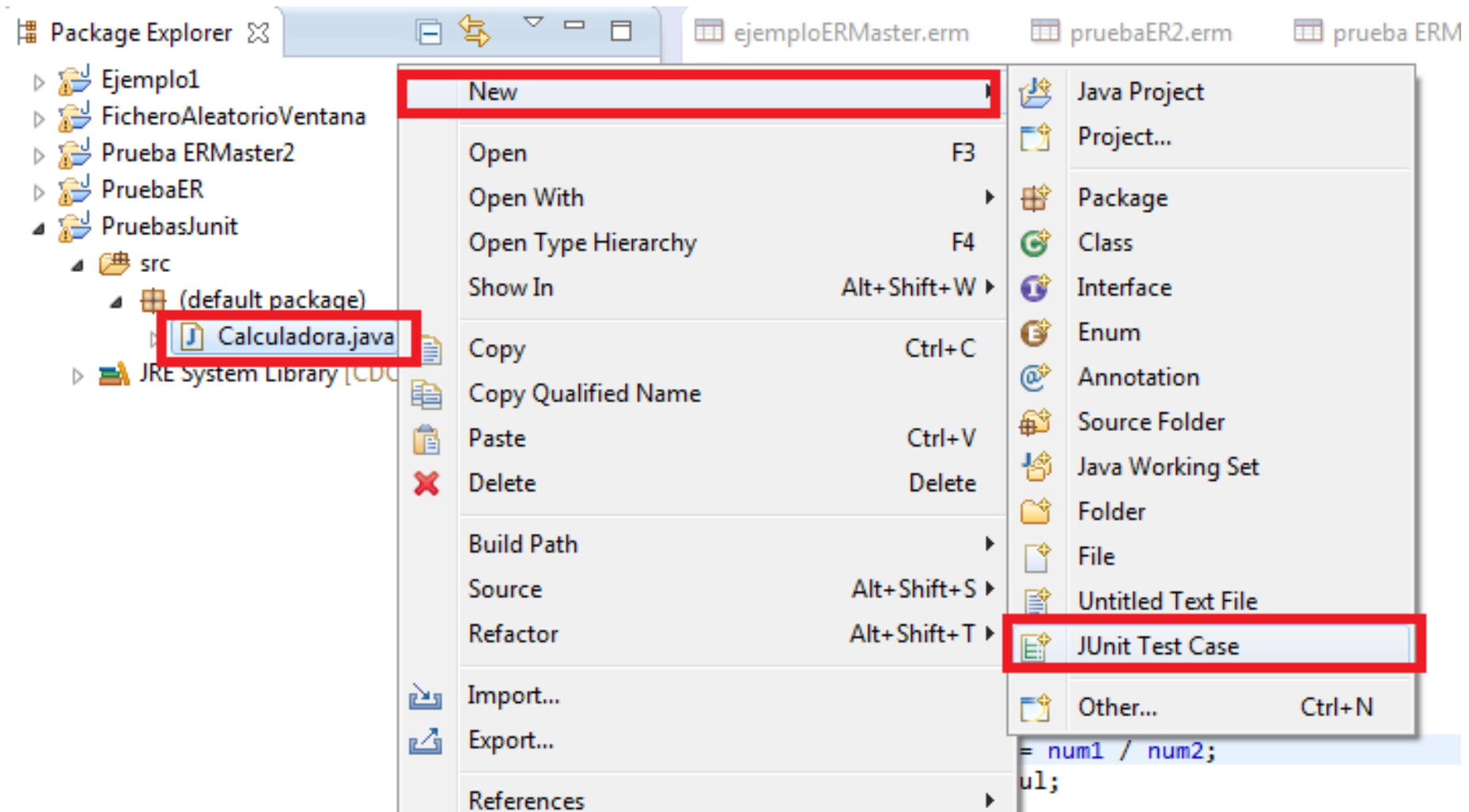
Pruebas unitarias con junit (III)

```
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 divide(){  
        int resul = num1 / num2;  
        return resul;  
    }  
}
```

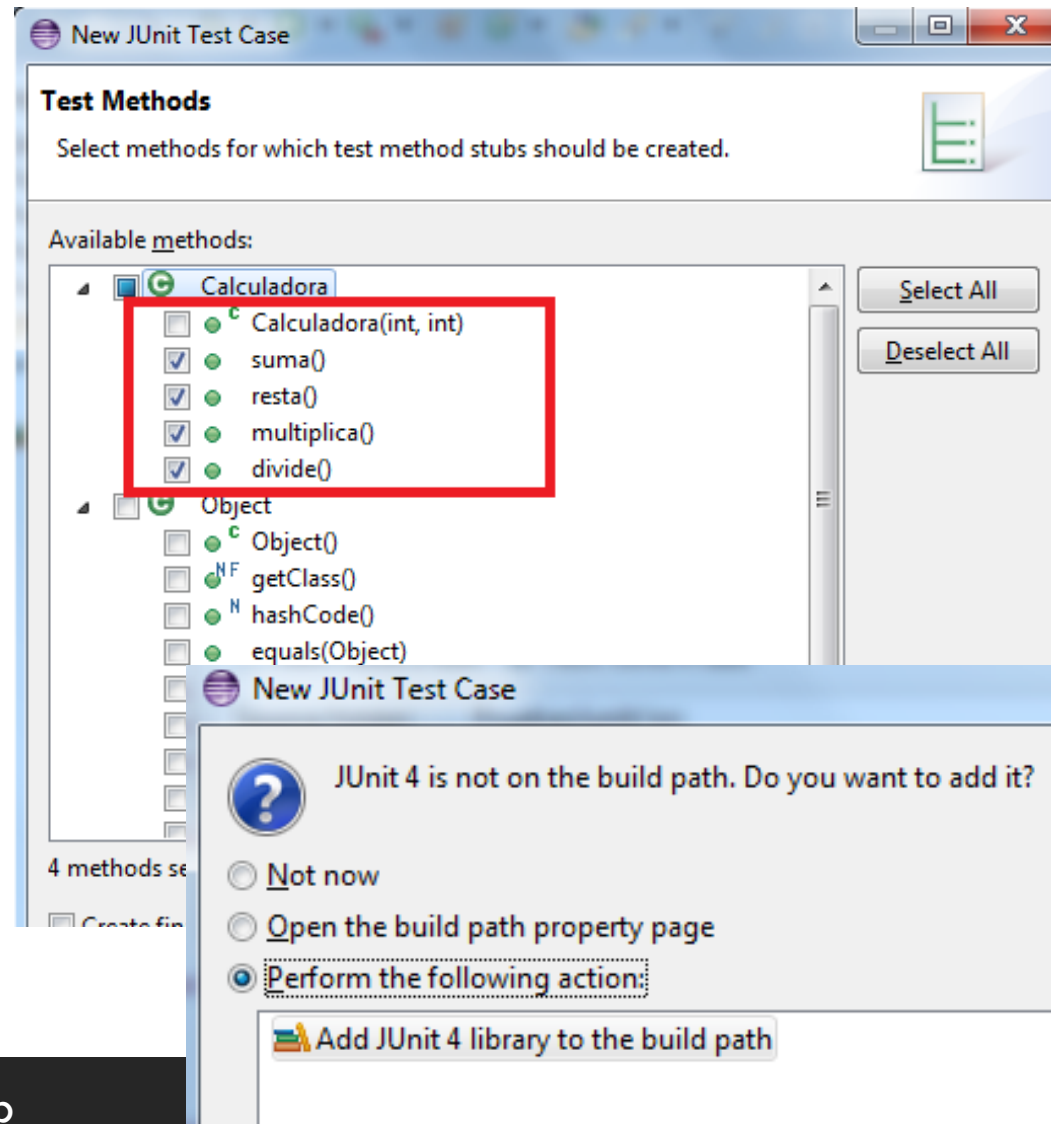
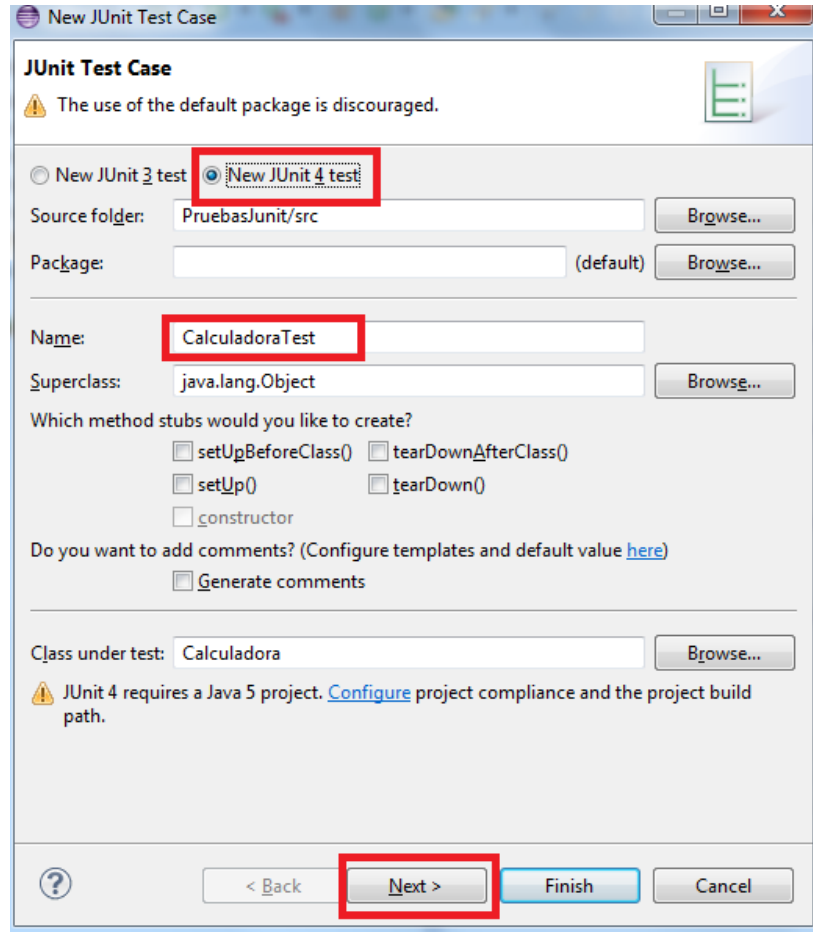
creación de una clase de prueba (I)

- ★ Dada la clase que queremos probar, la seleccionamos y con el botón derecho *New->JUnit Test Case* o *File->New->JUnit Test Case*.
- ★ Seleccionar **New JUnit 4 test** y dar un nombre.
- ★ Seleccionar los métodos que se quieren probar.
- ★ Incluir la clase de prueba en el proyecto.
 - a. Se crea un método de prueba para cada método a probar.
 - b. Los métodos son públicos no reciben argumentos ni devuelven valores.
 - c. El nombre del método se inicia con la palabra **test**.
 - d. Sobre cada método aparece la anotación **@Test**.
 - e. Cada método de prueba tiene una llamada al método **fail()** con un mensaje indicando que no está implementado.

creación de una clase de prueba (II)



creación de una clase de prueba (III)



creación de una clase de prueba (IV)

```
import static org.junit.Assert.*;
import org.junit.Test;
public class CalculadoraTest {
    @Test
    public void testSuma() {
        fail("Not yet implemented");
    }
    @Test
    public void testResta() {
        fail("Not yet implemented");
    }
    @Test
    public void testMultiplica() {
        fail("Not yet implemented");
    }
    @Test
    public void testDivide() {
        fail("Not yet implemented");
    }
}
```

Preparación y ejecución de las pruebas(I)

★ Métodos útiles para hacer comprobaciones:

- a. `assertTrue(java.lang.String message, boolean condition)` y `assertTrue(boolean condition)`: Comprueba que la expresión se evalúa a Verdadero.
- b. `assertFalse(java.lang.String message, boolean condition)` y `assertFalse(boolean condition)`: Comprueba que la expresión se evalúa a Falso.
- c. `assertEquals(java.lang.String message, valorEsperado, valorReal)` y `assertEquals(valorEsperado, valorReal)`: Comprueba que el valorEsperado sea igual al valorReal.
- d. `assertNull(java.lang.String message, Object objeto)` y `assertNull(Object objeto)`: Comprueba que el objeto sea Null.
- e. `assertNotNull(java.lang.String message, Object objeto)` y `assertNotNull(Object objeto)`: Comprueba que el objeto no sea Null.
- f. `assertSame(java.lang.String message, Object objetoEsperado, Object objetoReal)` y `assertSame(Object objetoEsperado, Object objetoReal)`: Comprueba que el objetoEsperado sea igual al objetoReal.
- g. `assertNotSame(java.lang.String message, Object objetoEsperado, Object objetoReal)` y `assertNotSame(Object objetoEsperado, Object objetoReal)`: Comprueba que el objetoEsperado no sea igual al objetoReal.
- h. `fail(java.lang.String message)` y `fail()`: hace que la prueba falle.

Para saber más: <http://junit.sourceforge.net/javadoc/index.html?org/junit/>

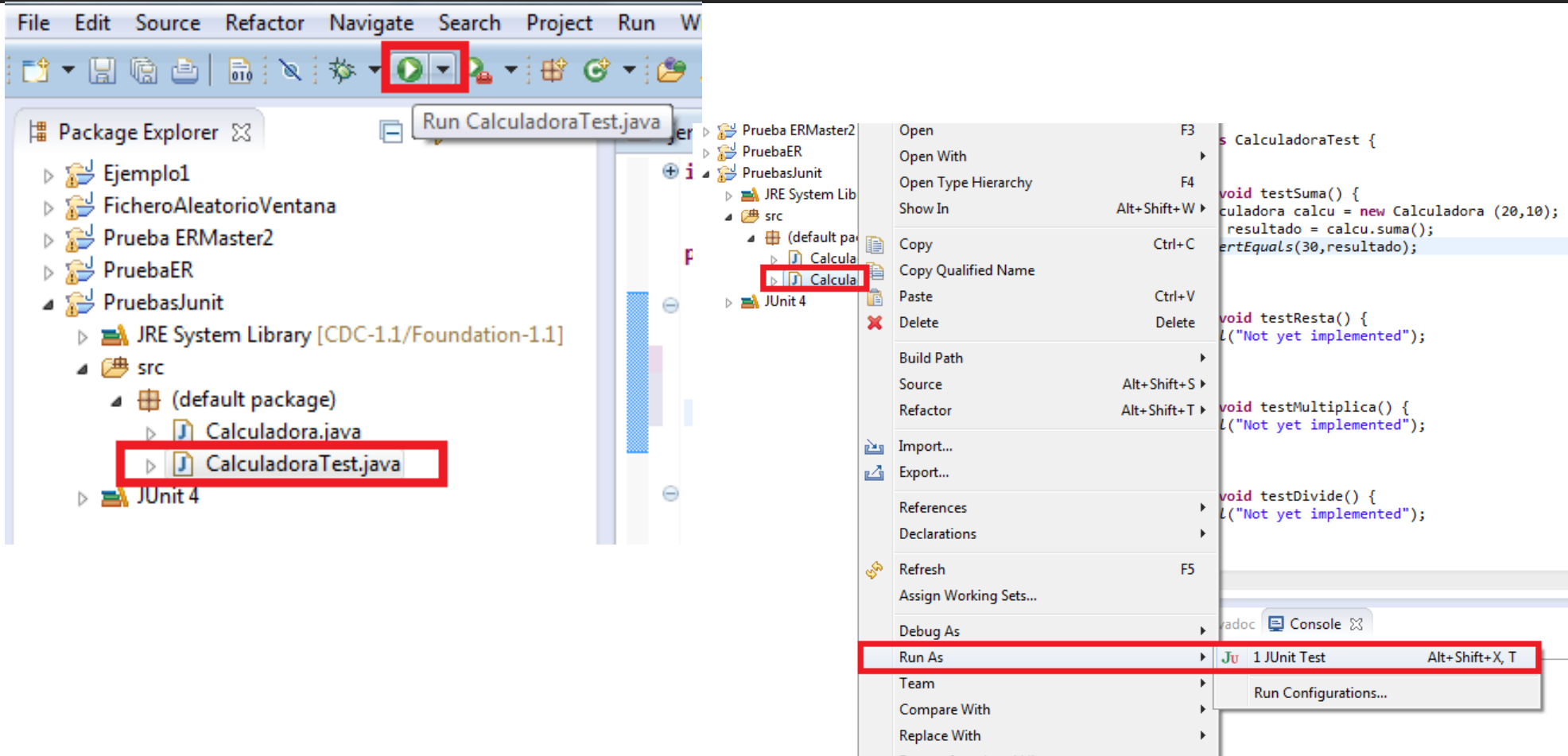
Preparación y ejecución de las pruebas(II)

- ★ Creamos el código de pruebas para el método `testSuma()` que probará el método `suma()`.
- ★ Creamos una instancia de la clase `Calculadora`.
- ★ Llamamos al método `suma()` con los valores a sumar.
- ★ Comprobamos los valores con el método `assertEquals()`. En el primer parámetro de este método escribimos el resultado esperado y como segundo parámetro el resultado de la llamada a `suma()`.

@Test

```
public void testSuma() {  
    Calculadora calculo = new Calculadora (20,10);  
    int resultado = calculo.suma();  
    assertEquals(30,resultado);  
}
```


Preparación y ejecución de las pruebas(III)



The screenshot illustrates the process of running a JUnit test in an IDE. The Package Explorer on the left shows the project structure, with 'CalculadoraTest.java' highlighted under the 'PruebasJUnit' package. The Run menu is open, showing the 'Run As' option selected. The code editor on the right shows the implementation of 'CalculadoraTest'.

Package Explorer Structure:

- Ejemplo1
- FicheroAleatorioVentana
- Prueba ERMaster2
- PruebaER
- PruebasJUnit
 - JRE System Library [CDC-1.1/Foundation-1.1]
 - src
 - (default package)
 - Calculadora.java
 - CalculadoraTest.java**
 - JUnit 4

Run Menu Options:

- Open (F3)
- Open With
- Open Type Hierarchy (F4)
- Show In (Alt+Shift+W)
- Copy (Ctrl+C)
- Copy Qualified Name
- Paste (Ctrl+V)
- Delete (Delete)
- Build Path
- Source (Alt+Shift+S)
- Refactor (Alt+Shift+T)
- Import...
- Export...
- References
- Declarations
- Refresh (F5)
- Assign Working Sets...
- Debug As
- Run As** (Alt+Shift+X, T)
- Team
- Compare With
- Replace With

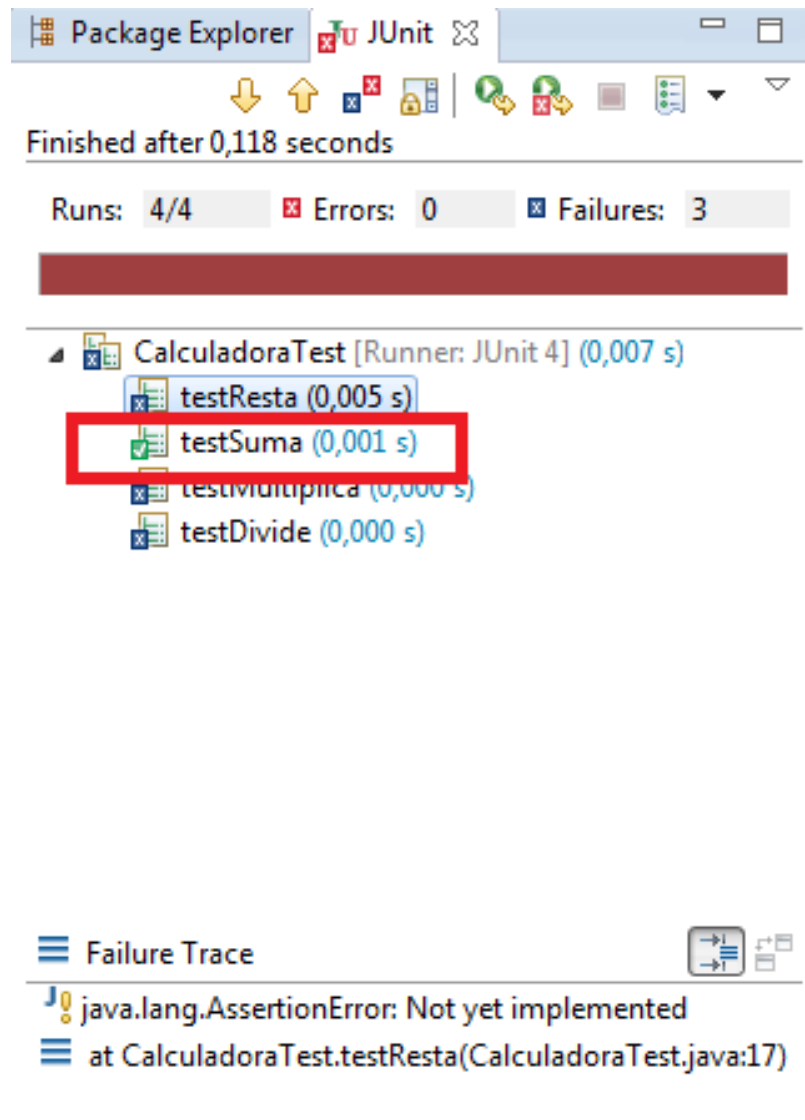
Code Editor Content:

```
CalculadoraTest {  
    void testSuma() {  
        calculadora calculo = new Calculadora (20,10);  
        resultado = calculo.suma();  
        assertEquals(30,resultado);  
    }  
    void testResta() {  
        //("Not yet implemented");  
    }  
    void testMultiplica() {  
        //("Not yet implemented");  
    }  
    void testDivide() {  
        //("Not yet implemented");  
    }  
}
```

Preparación y ejecución de las pruebas(IV)

- ★ Se abre la pestaña de JUnit donde se muestran los resultados de ejecución de las pruebas.
- ★ Al lado de cada prueba aparece un icono con una marca:
 - prueba exitosa: marca de verificación verde.
 - fallo (comprobación que no se cumple): aspa azul.
 - error (excepción durante la ejecución del código): aspa roja.
- ★ El resultado de la ejecución de la prueba muestra:
Runs : 4/4 Errors:0 Failures:3

Preparación y ejecución de las pruebas(V)



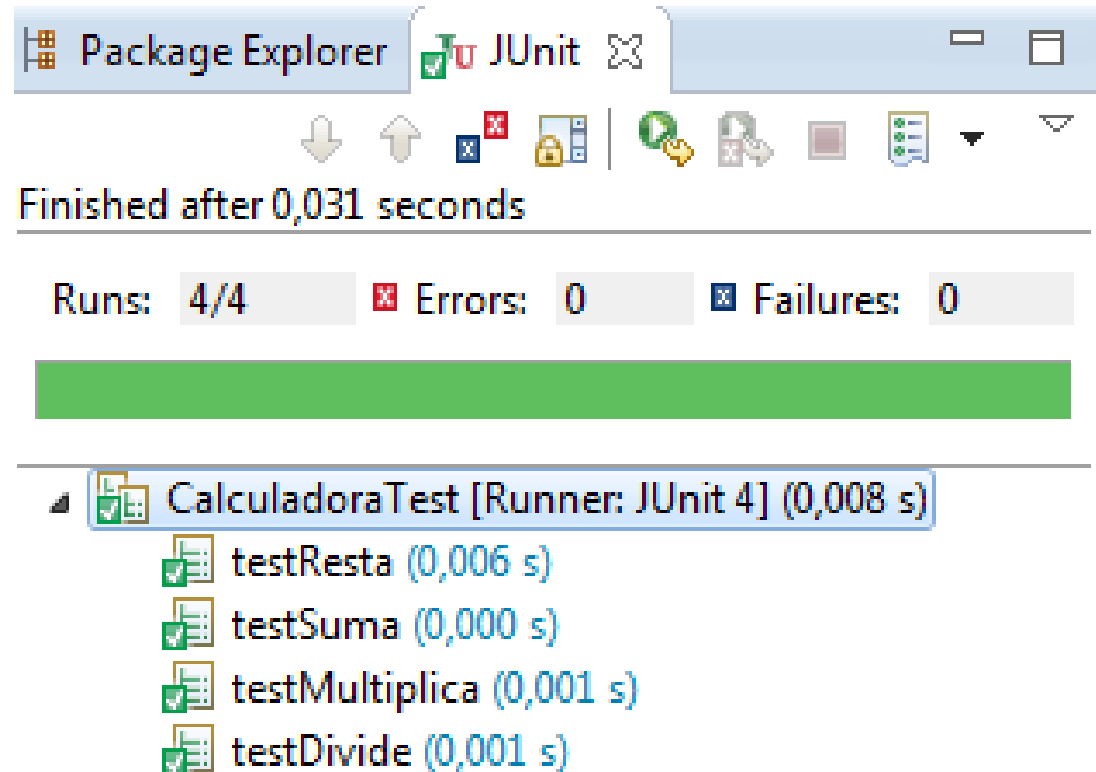
Preparación y ejecución de las pruebas(VI)

```
@Test
public void testResta() {
    Calculadora calculo = new Calculadora (20,10);
    int resultado = calculo.resta();
    assertEquals(10,resultado);
}
```

```
@Test
public void testMultiplica() {
    Calculadora calculo = new Calculadora (20,10);
    int resultado = calculo.multiplica();
    assertEquals(200,resultado);
}
```

```
@Test
public void testDivide() {
    Calculadora calculo = new Calculadora (20,10);
    int resultado = calculo.divide();
    assertEquals(2,resultado);
}
```

Preparación y ejecución de las pruebas(VII)



Preparación y ejecución de las pruebas(VIII)

The image displays two side-by-side screenshots of the Eclipse IDE's JUnit test runner interface, illustrating different types of test failures.

Left Screenshot (Assertion Error):

- Package Explorer: JUnit
- Status: Finished after 0,063 seconds
- Summary: Runs: 4/4, Errors: 1, Failures: 1
- Test Results:
 - CalculadoraTest [Runner: JUnit 4] (0,017 s)
 - testResta (0,000 s) [Success]
 - testSuma (0,000 s) [Success]
 - testMultiplica (0,002 s) [Failure]
 - testDivide (0,015 s) [Success]
- Failure Trace:
 - java.lang.AssertionError: expected:<20> but was:<200>
 - at CalculadoraTest.testMultiplica(CalculadoraTest.java:26)

Right Screenshot (Arithmetic Exception):

- Package Explorer: JUnit
- Status: Finished after 0,063 seconds
- Summary: Runs: 4/4, Errors: 1, Failures: 1
- Test Results:
 - CalculadoraTest [Runner: JUnit 4] (0,017 s)
 - testResta (0,000 s) [Success]
 - testSuma (0,000 s) [Success]
 - testMultiplica (0,002 s) [Success]
 - testDivide (0,015 s) [Failure]
- Failure Trace:
 - java.lang.ArithmeticException: / by zero
 - at Calculadora.divide(Calculadora.java:23)
 - at CalculadoraTest.testDivide(CalculadoraTest.java:33)

Preparación y ejecución de las pruebas(IX)

Para probar un método que puede lanzar excepciones se utiliza el parámetro **expected** con la anotación **@Test**. Y en el método la instrucción **throw**.

La prueba fallará si no se produce la excepción.

```
public int divide0(){
    if (num2==0) {
        throw new
java.lang.ArithmeticException("División por 0");
    }else{
        int resul = num1 / num2;
        return resul;
    }
}

@Test (expected = java.lang.ArithmeticException.class)
public void testDivide0() {
    Calculadora calculo = new Calculadora (20,0);
    Integer resultado = calculo.divide0();
}
```

Preparación y ejecución de las pruebas(X)

★ En la vista de JUnit se muestran varios botones:

★ **Next Failed Test:** navega a la siguiente prueba que ha producido fallo o error.

★ **Previous Failed Test:** navega a la anterior prueba que ha producido fallo o error.

★ **Show Failures Only:** muestra sólo las pruebas que han producido fallo o error.

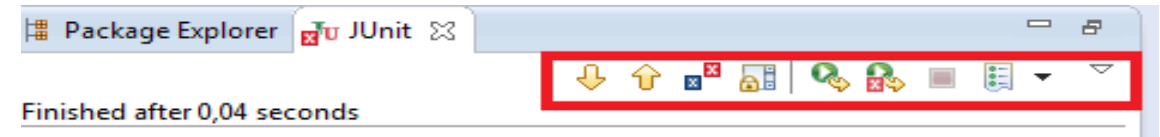
★ **Scroll Lock:** activa o desactiva el scroll lock.

★ **Rerun Test:** vuelve a ejecutar las pruebas.

★ **Rerun Test- Failures First:** vuelve a ejecutar las pruebas, ejecutando en primer lugar los fallos o errores.

★ **Stop JUnit Test Run:** detiene la ejecución de las pruebas.

★ **Test Run History:** muestra el historial de las pruebas realizadas.

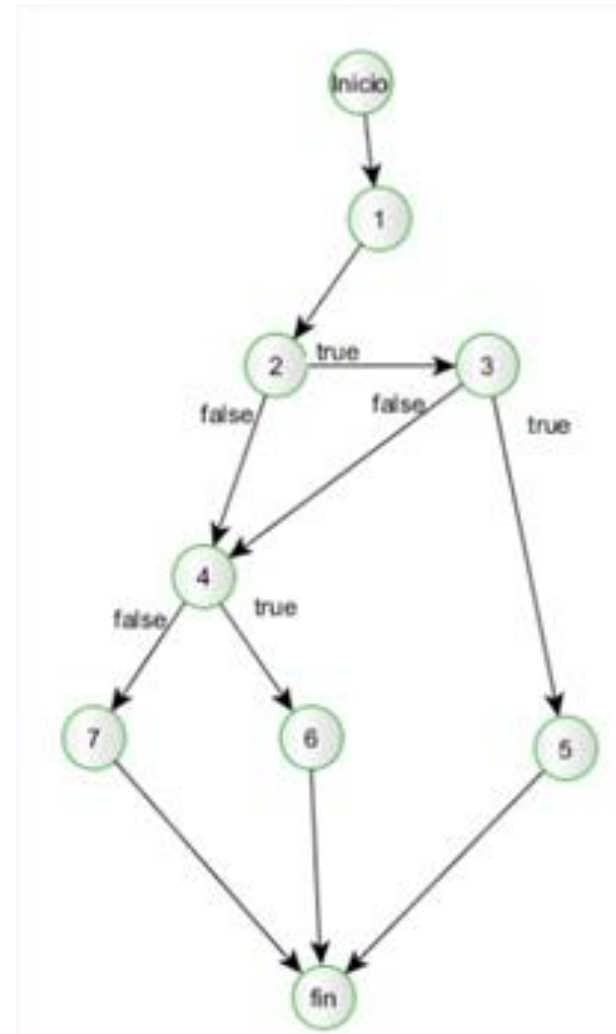
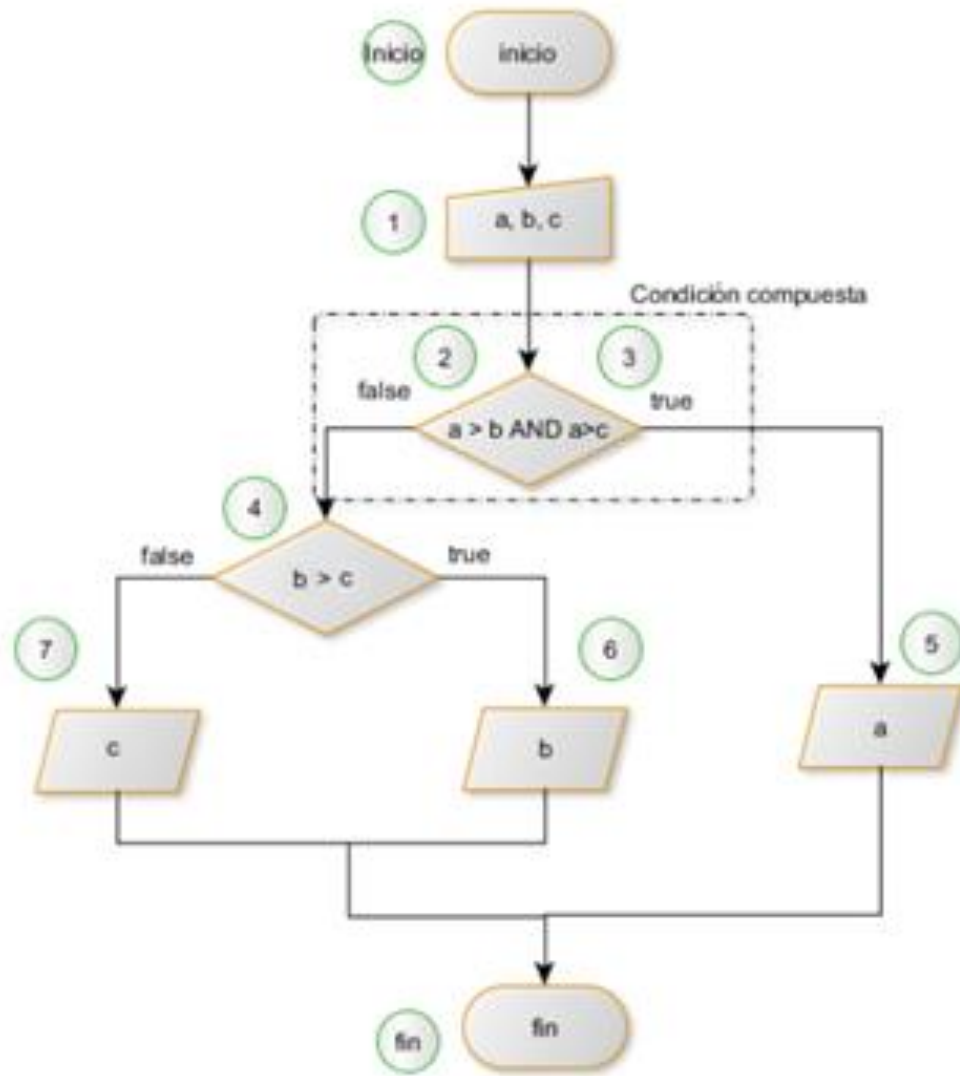


Preparación y ejecución de las pruebas: Ejercicio 1

```
public class numMayor
{
    /* Método que compara 3 números y devuelve el mayor*/
    public int numero_mayor(int a, int b, int c)
    {
        if (a > b && a > c)
        {
            return a;
        } else if (b > c)
        {
            return b;
        } else
        {
            return c;
        }
    }
}
```

```
1
2 public class numMayor
3 {
4     /* Método que compara 3 números y devuelve el mayor*/
5     public int numero_mayor(int a, int b, int c)
6     {
7         if (a > b && a > c)
8         {
9             return a;
10        } else if (b > c)
11        {
12            return b;
13        } else
14        {
15            return c;
16        }
17    }
18
19 }
```

preparación y ejecución de las pruebas: Ejercicio 1



Preparación y ejecución de las pruebas: Ejercicio 1

```
1 import static org.junit.Assert.*;
2
3 import org.junit.Test;
4
5 public class numMayorTest
6 {
7
8     @Test
9     public void testNumero_mayor1()
10    {
11        int a = 3;
12        int b = 2;
13        int c = 1;
14        numMayor num = new numMayor();
15        int result = num.num_mayor(a, b, c);
16        assertEquals(3, result);
17    }
18
19     @Test
20     public void testNumero_mayor2()
21    {
22        int a = 5;
23        int b = 3;
24        int c = 9;
25        numMayor num = new numMayor();
26        int result = num.num_mayor(a, b, c);
27        assertEquals(9, result);
28    }
29 }
```

```
30 @Test
31 public void testNumero_mayor3()
32 {
33     int a = 1;
34     int b = 3;
35     int c = 5;
36     numMayor num = new numMayor();
37     int result = num.num_mayor(a, b, c);
38     assertEquals(5, result);
39 }
40
41 @Test
42 public void testNumero_mayor4()
43 {
44     int a = 1;
45     int b = 3;
46     int c = 2;
47     numMayor num = new numMayor();
48     int result = num.num_mayor(a, b, c);
49     assertEquals(3, result);
50 }
51
52 }
```

| CAMINO | ENTRADA | PRUEBA | SALIDA |
|-------------|--------------------------------------|---------------|--------|
| 1,2,3,5,F | a>b = True; a>c = True | a=3; b=2; c=1 | a |
| 1,2,3,4,7,F | a>b = True; a>c = False; b>c = False | a=5; b=3; c=9 | c |
| 1,2,4,7,F | a>b = False; b>c = False | a=1; b=3; c=5 | c |
| 1,2,4,6,F | a>b = False; b>c = True | a=1; b=3; c=2 | b |

Preparación y ejecución de las pruebas: Ejercicio 2

★ Crea un vector si la dimensión del vector es igual a n

```
1 public class VectorJUnit {
2
3   int[] vector_nuevo (int n, int[] valores)
4   {
5       int[] vec = new int[n];
6       if (n == valores.length)
7       {
8           for (int i=0;i<vec.length;i++)
9           {
10              vec[i] = valores [i];
11          }
12      }else
13          throw new java.lang.NullPointerException ("Introduzca un número de valores igual a la dimensión");
14      return vec;
15  }
16
17 }
```

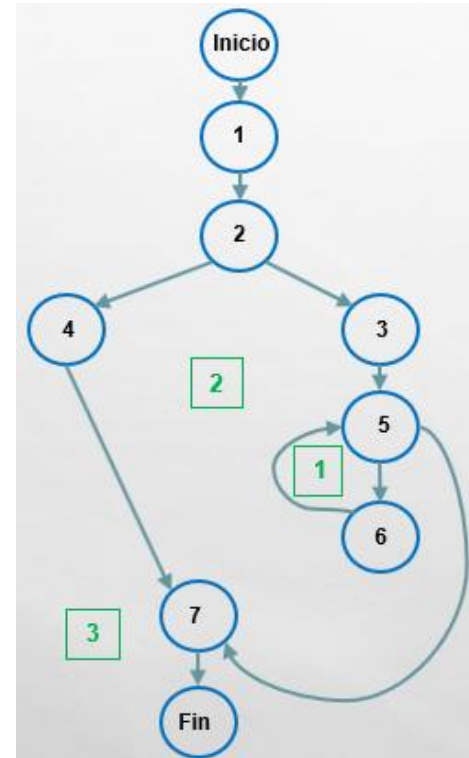
```
public class VectorJUnit {
int[] vector_nuevo (int n, int[] valores)
{
int[] vec = new int[n];
if (n == valores.length)
{
for (int i=0;i<vec.length;i++)
{
vec[i] = valores [i];
}
}else
throw new java.lang.NullPointerException ("Introduzca un número de valores igual a la
dimensión");
return vec;
}
}
```

Preparación y ejecución de las pruebas: Ejercicio 2

```

1 import static org.junit.Assert.*;
2
3 import org.junit.Test;
4
5 public class VectorJUnitTest {
6
7     @Test (expected = java.lang.NullPointerException.class)
8     public void testVectorJUnit1() {
9         int[] array = {4,7};
10        int n = 1;
11        VectorJUnit v1 = new VectorJUnit();
12        int[] result = v1.vector_nuevo(n,array);
13        assertEquals(array, result);
14    }
15
16    @Test
17    public void testVectorJUnit2() {
18        int[] array = {9,4,7};
19        int n = 3;
20        VectorJUnit v1 = new VectorJUnit();
21        int[] result = v1.vector_nuevo(n,array);
22        assertEquals(array, result);
23    }
24
25    @Test
26    public void testVectorJUnit3() {
27        int[] array = {};
28        int n = 0;
29        VectorJUnit v1 = new VectorJUnit();
30        int[] result = v1.vector_nuevo(n,array);
31        assertEquals(array, result);
32    }
33
34 }
35

```



| CAMINO | PRUEBA | SALIDA |
|---------------|--------------------|--------------------|
| 1-2-4-7 | n=1; array = {4,7} | Expected Exception |
| 1-2-3-5-7 | n=0; array = {} | Array Null |
| 1-2-3-5-6-5-7 | n=3; array={9,4,7} | {{9,4,7},{9,4,7}} |

Preparación y ejecución de las pruebas: Ejercicio 3

★ Método que devuelve TRUE si el número num está en el array

```
4- boolean chequeaNum (int[] array, int num)
5 {
6     boolean b = false;
7     int[] a = array;
8
9     for (int i=0;i<a.length;i++)
10     {
11         if (a[i] == num)
12         {
13             b = true;
14         }
15     }
16     return b;
17 }
```

```
boolean chequeaNum (int[] array, int num)
{
    boolean b = false;
    int[] a = array;
    for (int i=0;i<a.length;i++)
    {
        if (a[i] == num)
        {
            b = true;
        }
    }
    return b;
}
```

Preparación y ejecución de las pruebas: Ejercicio 3

```
7- @Test
8 public void testChequeaNum_1() {
9     int[] array = {};
10    int n = 2;
11    diffArray v1 = new diffArray();
12    boolean result = v1.chequeaNum(array, n);
13    assertFalse(result);
14 }
15
16- @Test
17 public void testChequeaNum_2() {
18     int[] array = {2,3,7};
19     int n = 5;
20     diffArray v1 = new diffArray();
21     boolean result = v1.chequeaNum(array, n);
22     assertFalse(result);
23 }
24
25- @Test
26 public void testChequeaNum_3() {
27     int[] array = {2,3,7};
28     int n = 7;
29     diffArray v1 = new diffArray();
30     boolean result = v1.chequeaNum(array, n);
31     assertTrue(result);
32 }
33
```

| CAMINO | PRUEBA | SALIDA |
|---------------|----------------------|--------|
| 1,2,7 | n=2; array = {} | False |
| 1,2,3,5,2,7 | n=5; array = {2,3,7} | False |
| 1,2,3,4,5,2,7 | n=7; array={2,3,7} | True |

Tipos de anotaciones (I)

- ★ JUnit dispone de una serie de anotaciones que permiten ejecutar código antes y después de las pruebas:
 - a. **@Before**: el código se ejecuta antes de cualquier método de prueba. Se usa para inicializar datos.
 - b. **@After**: el código será ejecutado después de la ejecución de todos los métodos de prueba. Se usa para limpiar datos.
 - c. **@BeforeClass**: el código es invocado una vez al principio del lanzamiento de todas las pruebas. Se usa para inicializar atributos comunes a todas las pruebas. Sólo puede haber un método de este tipo.
 - d. **@AfterClass**: el código es invocado una vez al finalizar todas las pruebas. Sólo puede haber un método de este tipo.

Tipos de anotaciones (II)

```
import static org.junit.Assert.*;
import org.junit.After;
import org.junit.Before;
import org.junit.Test;

public class CalculadoraTest2 {
    private Calculadora calculo;
    private int resultado;

    @Before
    public void creaCalculadora(){
        calculo = new Calculadora (20,10);
    }

    @After
    public void borraCalculadora(){
        calculo = null;
    }

    @Test
    public void testSuma() {
        resultado = calculo.suma();
        assertEquals(30,resultado);
    }
}
```

Tipos de anotaciones (III)

```
import static org.junit.Assert.*;
import org.junit.AfterClass;
import org.junit.BeforeClass;
import org.junit.Test;

public class CalculadoraTest3 {
    private static Calculadora calcul;
    private int resultado;

    @BeforeClass
    public static void creaCalculadora(){
        calcul = new Calculadora (20,10);
    }

    @AfterClass
    public static void borraCalculadora(){
        calcul = null;
    }
}
```

Pruebas parametrizadas (I)

- ★ **JUnit** permite generar parámetros para lanzar varias veces una prueba con distintos valores.
 - a. Se añade la etiqueta **@RunWith(Parameterized.class)** a la clase de prueba.
 - b. En la clase se crea un **atributo** para cada parámetro y un constructor con tantos argumentos como parámetros.
 - c. Se define un método con la etiqueta **@Parameters**, que devolverá la lista de valores a probar.
 - d. En **@Parameters** se definen filas de valores, una para cada caso de prueba.

Pruebas parametrizadas (II)

```
import static org.junit.Assert.*;
import java.util.*;
import org.junit.Test;
import org.junit.runner.RunWith;
import org.junit.runners.Parameterized;
import org.junit.runners.Parameterized.Parameters;
@RunWith(Parameterized.class)
public class CalculadoraTest4 {
    private int nume1;
    private int nume2;
    private int resul;

    public CalculadoraTest4(int nume1, int nume2, int resul){
        this.nume1 = nume1;
        this.nume2 = nume2;
        this.resul = resul;
    }

    @Parameters
    public static Collection<Object[]> numeros(){
        return Arrays.asList((new Object[][]{
            {20,10,2},{30,-2,-15}, {5,2,3}
        }));
    }

    @Test
    public void testDivide(){
        Calculadora calculo = new Calculadora(nume1, nume2);
        int resultado = calculo.divide();
        assertEquals(resul, resultado);
    }
}
```

Pruebas parametrizadas (III)

File Edit Source Refactor Navigate Search Project Run Window Help

Package Explorer JUnit

Finished after 0,121 seconds

Runs: 21/21 Errors: 18 Failures: 3

CalculadoraTest4 [Runner: JUnit 4] (0,037 s)

- [0] (0,007 s)
 - testResta[0] (0,003 s)
 - testSuma[0] (0,001 s)
 - testMultiplica[0] (0,001 s)
 - testDivide0[0] (0,001 s)
 - testDivide2[0] (0,000 s)
 - testDivide0[0] (0,001 s)
 - testResta2[0] (0,000 s)
- [1] (0,009 s)
 - testResta[1] (0,000 s)
 - testSuma[1] (0,000 s)
 - testMultiplica[1] (0,001 s)

Failure Trace

java.lang.AssertionError: expected:<-15> but was:<28>
at CalculadoraTest4.testSuma(CalculadoraTest4.java:31)

```
@RunWith(Parameterized.class)
public class CalculadoraTest4 {
    private int nume1;
    private int nume2;
    private int resul;

    public CalculadoraTest4(int nume1, int nume2, int resul){
        this.nume1 = nume1;
        this.nume2 = nume2;
        this.resul = resul;
    }

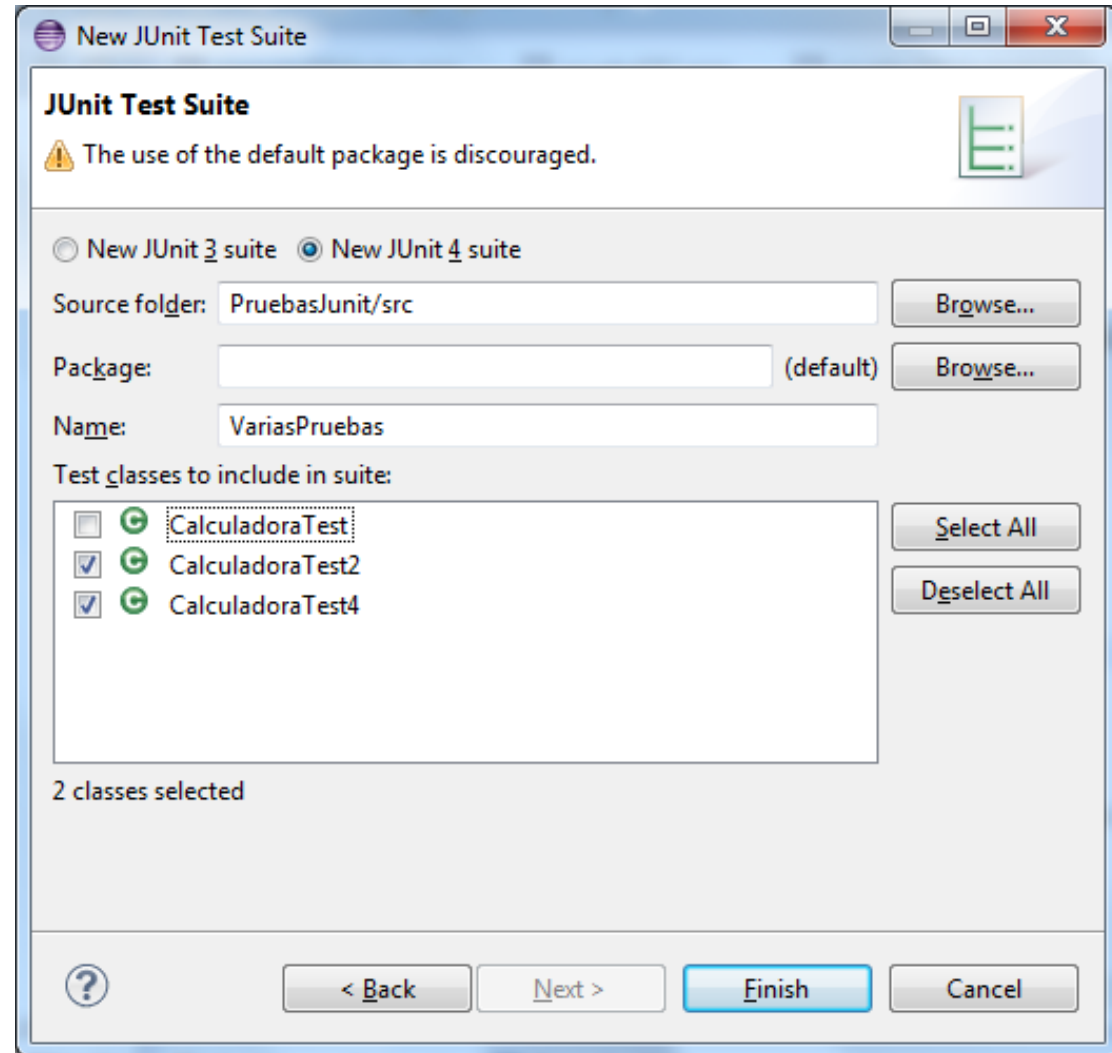
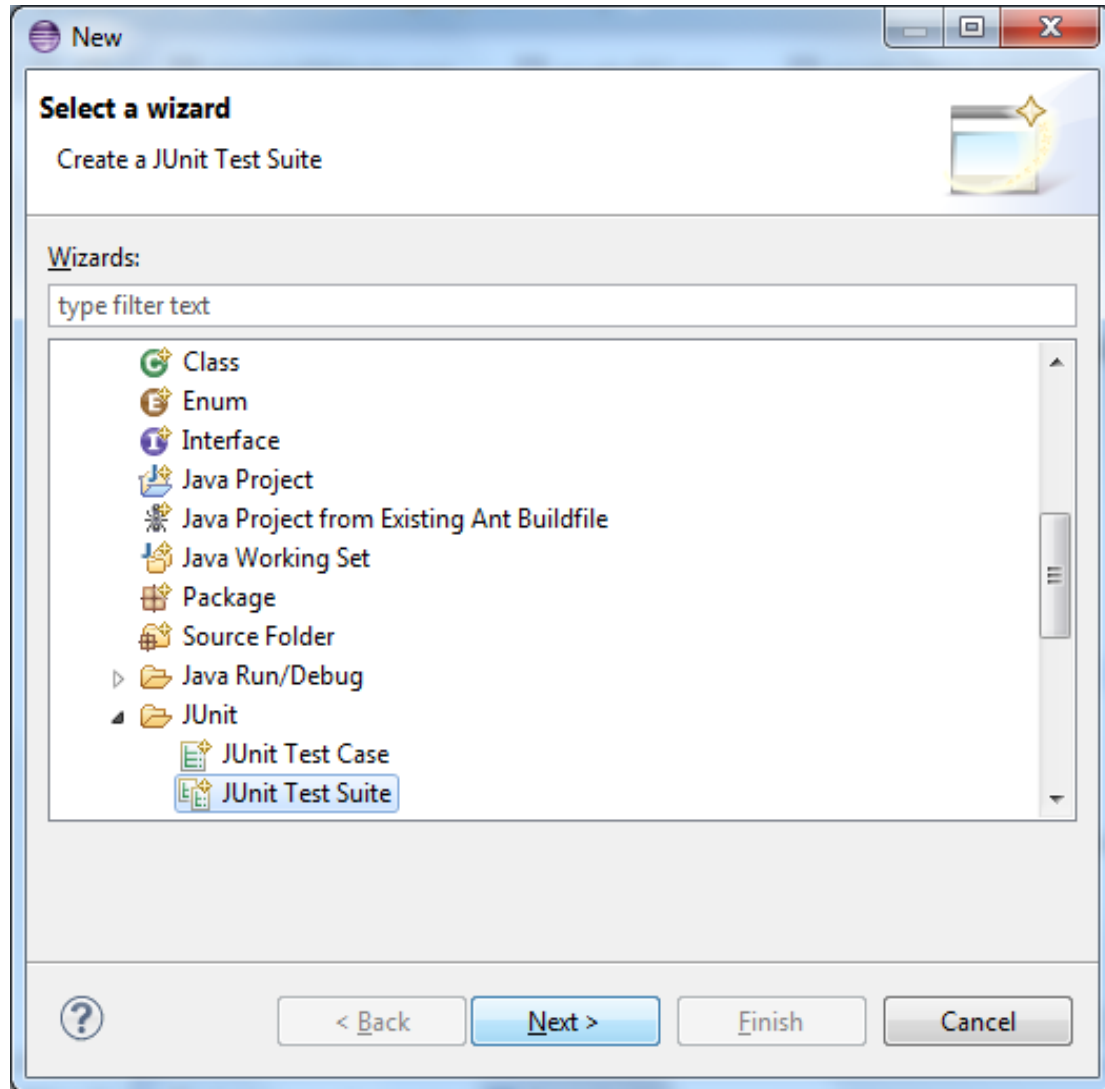
    @Parameters
    public static Collection<Object[]> numeros(){
        return Arrays.asList((new Object[][]{
            {20,10,2},{30,-2,-15}, {5,2,3}
        }));
    }

    @Test
    public void testSuma() {
        Calculadora calcul = new Calculadora (nume1, nume2);
        int resultado = calcul.suma();
        assertEquals(resul,resultado);
    }
}
```

Suite de pruebas (I)

- ★ **Test Suites** es un mecanismo de Unit que permite agrupar varias clases de prueba para que se ejecuten una tras otra.
- ★ Se crean desde *File->New->Other->Java->JUnit->JUnit Test Suite*.
- ★ Bajo un mismo nombre se integran las diferentes clases de prueba que se quieren agrupar.
- ★ La clase no contienen ninguna línea de código.
- ★ Anotaciones:
 - `@RunWith(Suite.class)`: indica que es una suite de pruebas.
 - `@SuiteClasses()`: se indican las clases que se van a ejecutar.

suite de pruebas (II)



Suite de pruebas (III)

```
import org.junit.runner.RunWith;
```

```
import org.junit.runners.Suite;
```

```
import org.junit.runners.Suite.SuiteClasses;
```

```
@RunWith(Suite.class)
```

```
@SuiteClasses({ CalculadoraTest2.class, CalculadoraTest4.class })
```

```
public class VariasPruebas {
```

```
}
```


Pruebas unitarias con DBUnit

- ★ **DBUnit** es una extensión de JUnit que permite realizar pruebas unitarias de clases que interactúan con bases de datos.
- ★ Utiliza ficheros XML para cargar los datos de prueba en la base de datos.
- ★ Al finalizar la prueba restaura el contenido de las tablas a su estado original.
- ★ Componentes:
 - **IDatabase**: conexión DBUnit-Base de datos.
 - **IDateSet**: colección de tablas.
 - **DatabaseOperation**: operación que se realiza sobre la BD antes y después de cada prueba.

Herramientas de testing

JUNIT [HTTPS://WWW.BAELDUNG.COM/JUNIT](https://www.baeldung.com/junit)

JUnit es un **framework** que permite realizar test repetibles (pruebas de regresión), es decir, que puede diseñarse un test para un programa o clase concreta y ejecutarlo tantas veces como sea necesario.

<https://junit.org/junit5/docs/5.0.3/api/overview-summary.html>



La ventaja es que puede (o mejor, debe) ejecutarse el test cada vez que se modifique o cambie algo del código y verificar si el programa sigue funcionando correctamente tras los cambios.

Para usar JUnit será necesario **descargar sus librerías**. Podemos hacer uso de Maven para ello. La última versión de JUnit es **JUnit5** (Jupiter) para hacer uso de esta versión desde un proyecto Maven, habrá que añadir al pom.xml las dependencias de junit-jupiter-engine y junit-jupiter-properties

```
<dependency>
  <groupId>org.junit.jupiter</groupId>
  <artifactId>junit-jupiter-engine</artifactId>
  <version>5.5.2</version>
  <scope>test</scope>
</dependency>
```



Plugins: <https://maven.apache.org/plugins/>

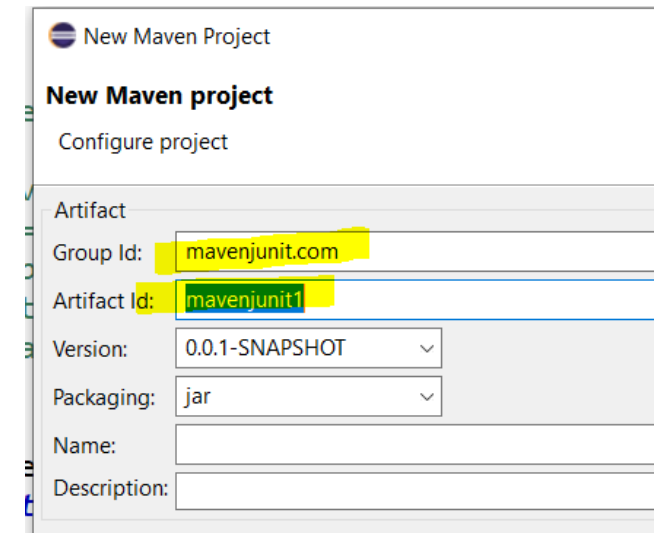
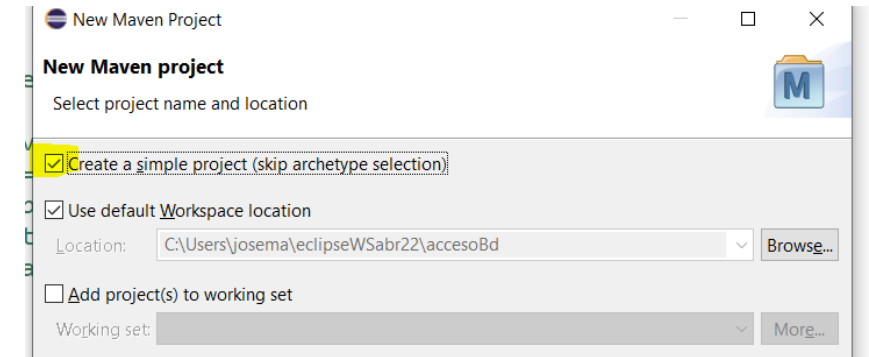
Dependencias: <https://mvnrepository.com/>

Para usar las librerías de forma sencilla vamos a hacer uso de Maven para ello creando un proyecto Maven en eclipse (new->Other->Maven->Maven Project y seleccionamos simple Project. Pulsamos Next, después añadimos el groupId y el artifactId y pulsamos Finish

La última versión de JUnit es JUnit5 (Jupiter) para hacer uso de esta versión desde un proyecto Maven, habrá que añadir al pom.xml

```
<dependency>
  <groupId>org.junit.jupiter</groupId>
  <artifactId>junit-jupiter-engine</artifactId>
  <version>5.5.2</version>
  <scope>test</scope>
</dependency>
```

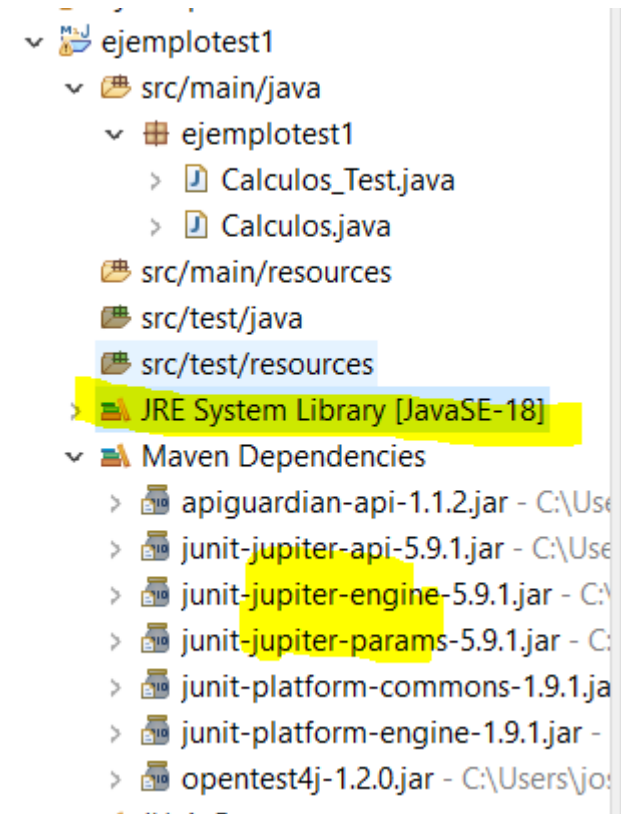
Videos configuración: [V1](#), [V2](#), [V3](#),



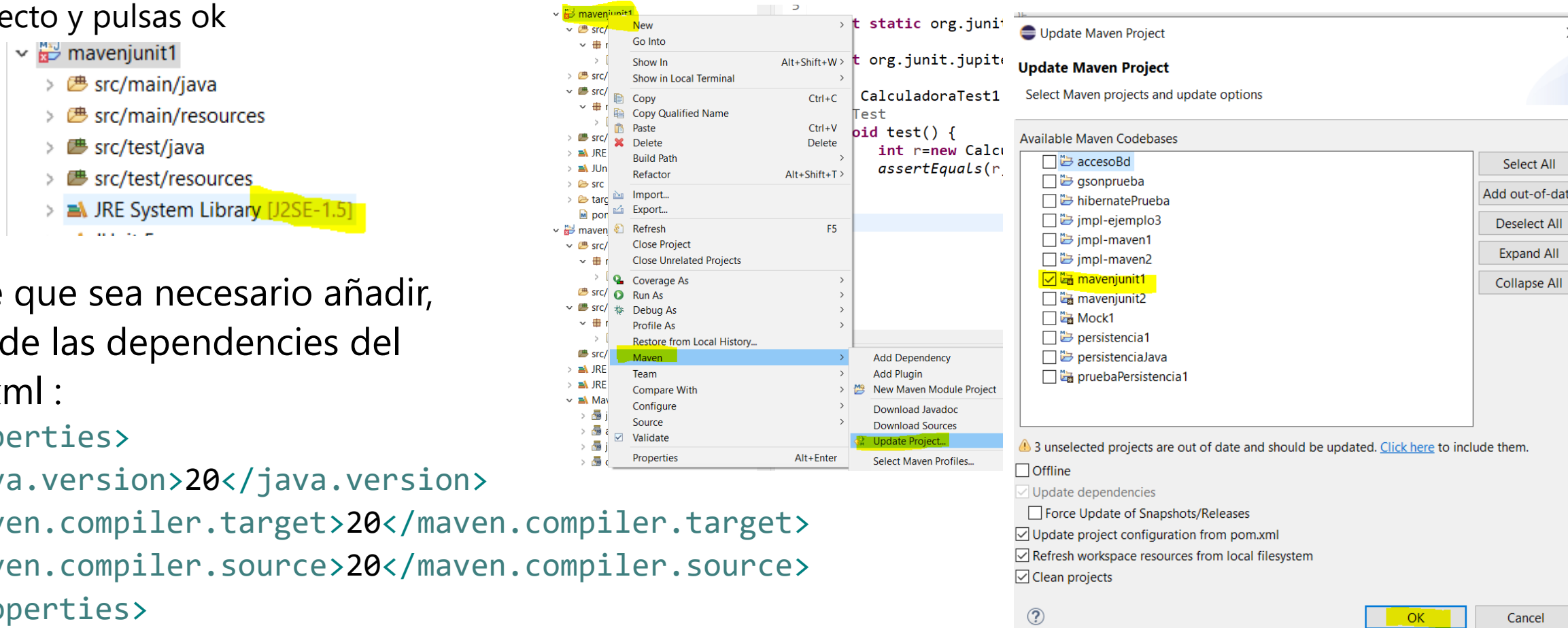
Para poder ejecutar los test con Maven hará falta usar el Maven-surefire-plugin como mínimo a la versión 2.22.0 y la versión de java mínima soportada será la 8. Hay que añadirla con properties y cambiar la que utiliza eclipse en el proyecto. Añadimos al pom.xml después de `</versión>`

```
<!-- Este POM no está completo, solo se muestra los ficheros de propiedades y la configuración necesaria para añadir JUnit -->
```

```
<properties>
  <java.version>1.8</java.version>
  <maven.compiler.target>1.8</maven.compiler.target>
  <maven.compiler.source>1.8</maven.compiler.source>
  <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
</properties>
<dependencies>
  <dependency>
    <groupId>org.junit.jupiter</groupId>
    <artifactId>junit-jupiter-engine</artifactId>
    <version>5.5.2</version>
    <scope>test</scope>
  </dependency>
  <dependency>
    <groupId>org.junit.jupiter</groupId>
    <artifactId>junit-jupiter-params</artifactId>
    <version>5.5.2</version>
    <scope>test</scope>
  </dependency>
</dependencies>
<build>
  <plugins>
    <plugin><!-- Need at least 2.22.0 to support JUnit 5 -->
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-surefire-plugin</artifactId>
      <version>3.0.0-M3</version>
    </plugin>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-compiler-plugin</artifactId>
      <version>3.8.1</version>
    </plugin>
  </plugins>
</build>
```



Si no tenemos la versión 1.8 o superior en nuestro proyecto podemos actualizarla pulsando con el botón derecho sobre la versión que tenemos y eligiendo Maven->Update Project, aunque no es imprescindible actualizarlo. Eliges el proyecto y pulsas ok



Puede que sea necesario añadir, antes de las dependencies del pom.xml :

```
<properties>
<java.version>20</java.version>
<maven.compiler.target>20</maven.compiler.target>
<maven.compiler.source>20</maven.compiler.source>
</properties>
```

Actividad 2. Crea un proyecto Maven con JUnit 5.

JUnit sirve para crear test unitarios. Los tests unitarios prueban las funcionalidades implementadas en el SUT (System Under Test). Si somos desarrolladores Java, para nosotros el SUT será la clase Java.

Los tests unitarios deben cumplir las siguientes características:

- Principio **FIRST**
 - **Fast**: Rápida ejecución.
 - **Isolated**: Independiente de otros test.
 - **Repeatable**: Se puede repetir en el tiempo.
 - **Self-Validating**: Cada test debe poder validar si es correcto o no a sí mismo.
 - **Timely**: ¿Cuándo se deben desarrollar los test? ¿Antes o después de que esté todo implementado? Sabemos que cuesta hacer primero los test y después la implementación (**TDD: Test-driven development**), pero es lo suyo para centrarnos en lo que realmente se desea implementar.

- Además, podemos añadir estos tres puntos más:
 - Sólo **probar los métodos públicos** de cada clase.
 - No se debe hacer uso de las dependencias de la clase a probar. Esto quizás es discutible porque en algunos casos donde las dependencias son clases de utilidades y se puede ser menos estricto. Se recomienda siempre aplicar el sentido común.
 - Un **test no debe implementar ninguna lógica de negocio** (nada de if...else...for...etc)
- Los tests unitarios tienen la siguiente estructura:
 - **Preparación** de **datos** de entrada.
 - **Ejecución** del test.
 - **Comprobación** del test (**assert**). No debería haber más de 1 assert en cada test.
- En cada test unitario deberías utilizar la estructura Given-Then-When

```
@Test
void sumar() {
    //Given
    int n1=3, n2=4;
    //When
    ServicioCalculadora instance = new
ServicioCalculadora();
    int expResult = 7;
    int result = instance.sumar(n1, n2);
    //Then
    assertEquals(expResult, result);
}
```

JUnit es un framework Java para implementar test en Java. Se basa en anotaciones:

- **@Test**: indica que el método que la contiene es un test

```
@Test
void testDevuelveTrue() {
    System.out.println("Llamando a testDevuelveTrue");
}
```

- **@RepeatedTest**: Indica que el siguiente método será llamado las veces que la etiqueta recibe como parámetro. En el caso del ejemplo 2.

```
@RepeatedTest(2)
void testRepiteTest() {
    System.out.println("Llamando a testRepiteTest");
}
```

- **@BeforeEach**: Indica que el siguiente método es llamado antes de ejecutar cada una de las pruebas etiquetadas con @Test, @Repeatedtest o @ParameterizedTest.

```
@BeforeEach
void MetodoBeforeEach() {
    System.out.println("Llamando a MetodoBeforeEach");
}
```

- **@AfterEach**: Indica que el siguiente método es llamado después de ejecutar cada una de las pruebas etiquetadas con @Test, @Repeatedtest o @ParameterizedTest

```
@AfterEach
void MetodoAfterEach() {
    System.out.println("Llamando a MetodoAfterEach");
}
```

- **@BeforeAll**: Indica que el siguiente método (que tiene que ser **static**) es llamado una sola vez en toda la ejecución del programa. A continuación son llamadas las pruebas etiquetadas con @Test, @Repeatedtest o @ParameterizedTest

```
@BeforeAll
static void MetodoBeforeAll() {
    System.out.println("Llamando a MetodoBeforeAll");
}
```

- **@AfterAll**: Indica que el siguiente método es llamado una sola vez en toda la ejecución del programa. Antes habrán sido llamadas todas las pruebas etiquetadas con @Test, @Repeatedtest o @ParameterizedTest.

```
@AfterAll
static void MetodoAfterAll() {
    System.out.println("Llamando a MetodoAfterAll");
}
```

- **@Disabled**: evita la ejecución del test. No es muy recomendable su uso porque puede ocultar test fallidos. Si dudamos si el test debe estar o no, quizás borrarlo es la mejor de las decisiones.

```
@Test
@Disabled
void testDesactivado() {
    System.out.println("Desactivada la llamada a testDesactivado");
}
```

- **@ParameterizedTest** y **@ValueSource**. Permite pasar al método de prueba parámetros a utilizar en la ejecución. Se lanza una vez por parámetro pasado.

```
@ParameterizedTest
@ValueSource(strings = {"HOLA", "ADIOS"})
void testParameterizedTest(String sMiInput) {
    System.out.println("Llamando a testParameterizedTest " +
        sMiInput );
}
```

- **@ParameterizedTest** y **@CsvSource**. @ValueSource sólo permite el paso de un parámetro al método de prueba. Si necesitamos pasar varios se utiliza la etiqueta CsvSource. Se lanza una vez por cada tupla de parámetros pasada.

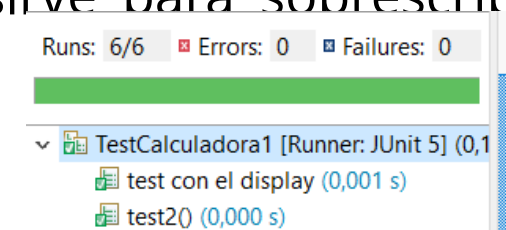
```
@ParameterizedTest
@CsvSource({"HOLA,1", "ADIOS,2"})
void testParameterizedIntTest(String a, int b) {
    System.out.println("Llamando a testParameterizedIntTest : "
        + a + " --- " + b);
}
```

Las condiciones de aceptación del test se implementa con los **asserts**. Se encuentran en la clase Assertions y los más comunes son los siguientes (<https://junit.org/junit5/docs/current/user-guide/#appendix>):

- **assertTrue/assertFalse** (condición a testear): Comprueba que la condición es cierta o falsa.
- **assertEquals/assertNotEquals** (valor esperado, valor obtenido). Es importante el orden de los valores esperado y obtenido.
- **assertNull/assertNotNull** (object): Comprueba que el objeto obtenido es nulo o no.
- **assertSame/assertNotSame**(object1, object2): Comprueba si dos objetos son iguales o no.
- **fail()**; indica que el test ha fallado

En Junit 5 (jupiter) además se ha añadido una nueva anotación que sirve para sobrescribir el nombre del test en la suite de JUnit **@DisplayName("")**.

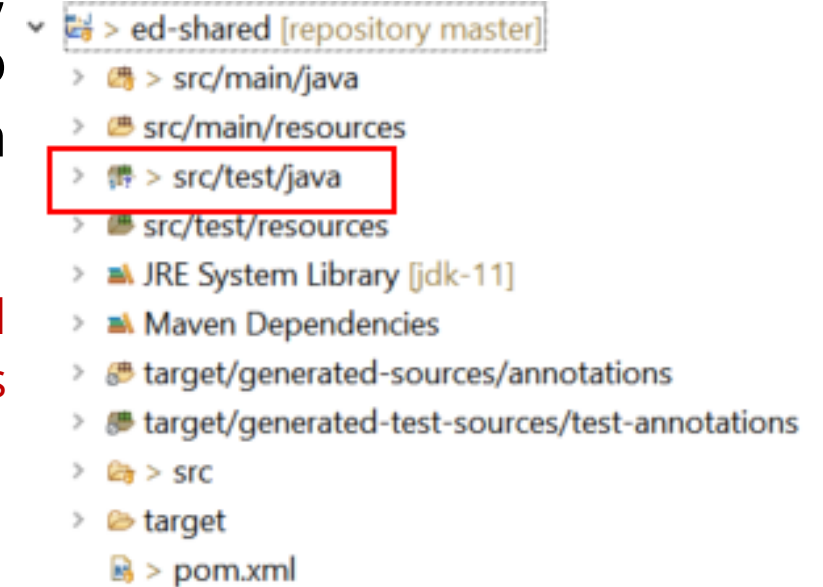
```
@Test
@DisplayName("test con el display")
void test1() {
    int r=new Calculadora().suma(3,4)
```



Hay muchas más anotaciones y posibilidades pero para una iniciación en JUnit con estas nos basta.

Por tanto, para empezar, creando una clase de test, deberemos crear una estructura para añadir los test, como estamos usando Maven, los proyectos por defecto con maven te generan un paquete para incorporar los tests

Antes de hacer los test tienes que haber añadido las dependencias y el plugin , en el pom.xml y luego, sobre el fichero pom.xml puedes seleccionar Maven->update project

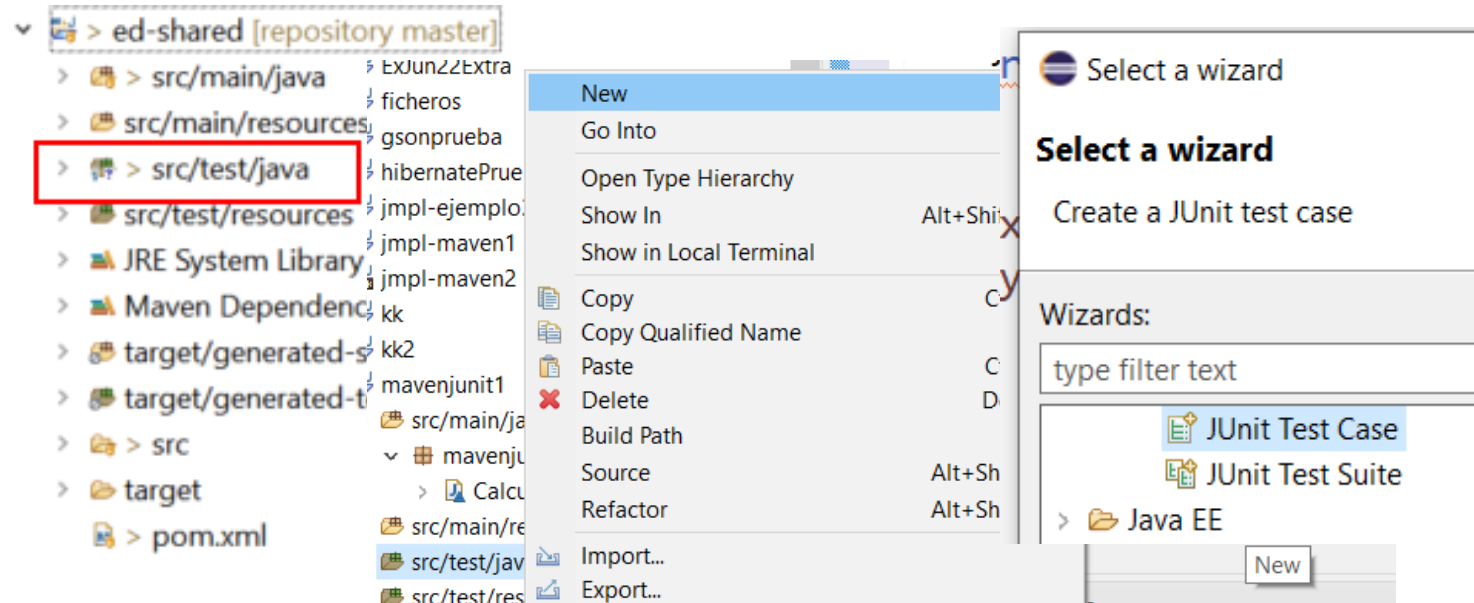


Primero creamos un fichero fuente Calculadora.java en src/main/java para realizar las pruebas →,

Por tanto para empezar creando una clase de test, deberemos crear una estructura para añadir los test, como estamos usando Maven, los proyectos por defecto con maven te generan un paquete para incorporar los tests.

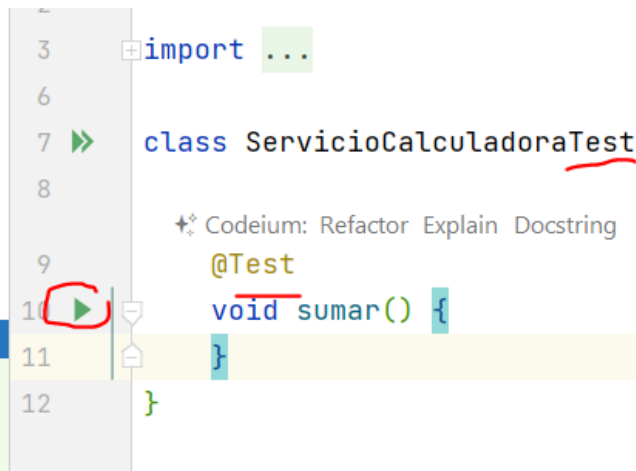
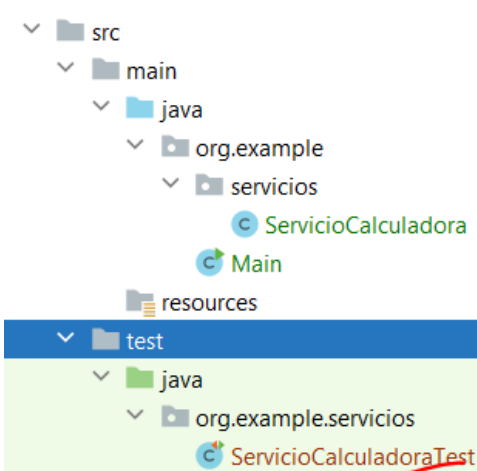
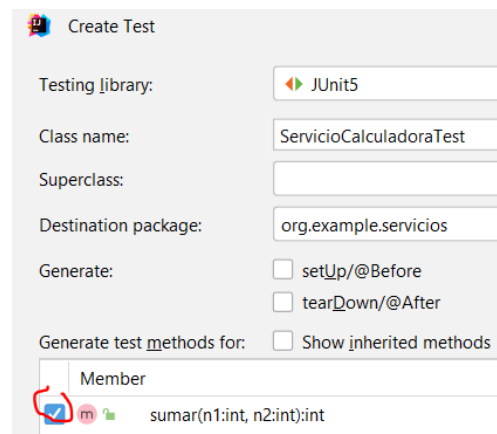
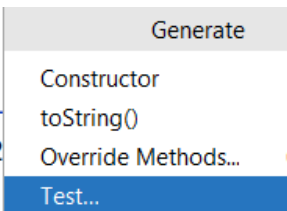
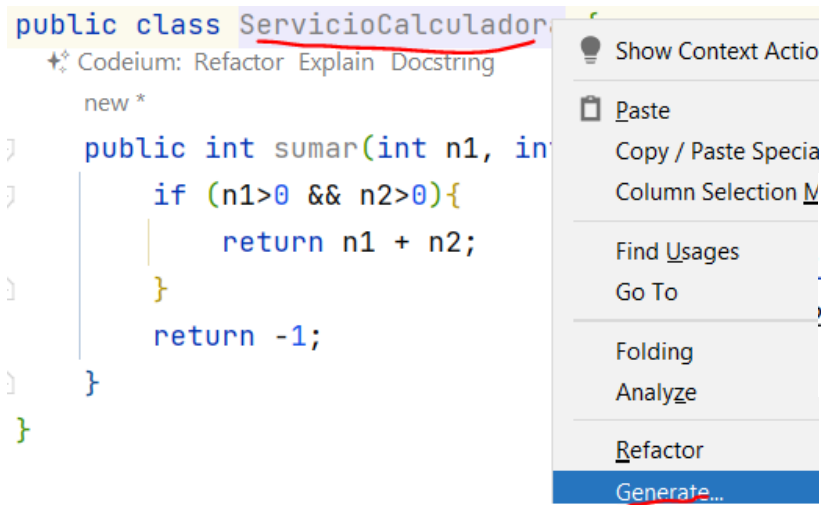
```
package mavenjunit1;  
public class Calculadora {  
    int sumar(int x, int y) {  
        return x+y;  
    }  
}
```

Pulsamos con el botón derecho sobre la carpeta de test (src/test/java) y elegimos New>Other->JUnit Test Case.



En **intelliJ** puedes generar test clase en la clase java que deseas probar) y eligiendo Generate > Test → elige los métodos y pulsa OK. Para ejecutarlos dale al play de cada test

También puedes pulsar **Alt+Ins** en la clase java que deseas probar y elegir el tipo de prueba



public class Ca

Generate

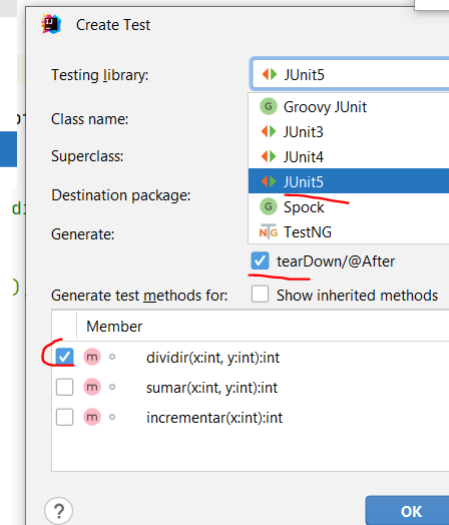
Constructor

toString()

Override Methods...

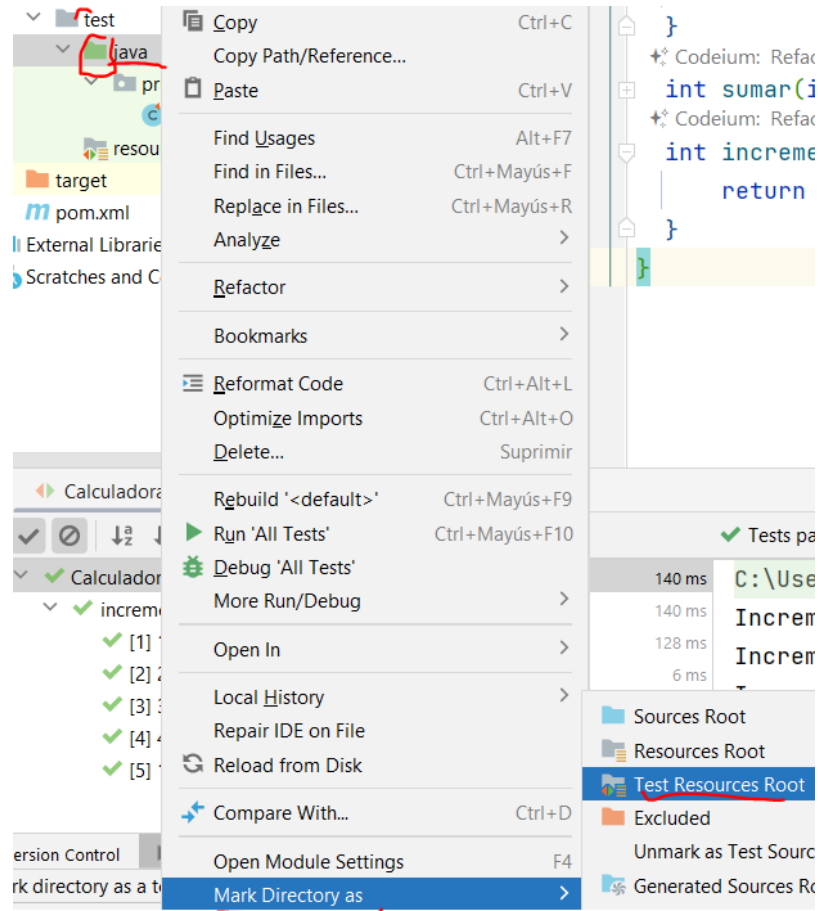
Test...

Copyright



<https://www.jetbrains.com/help/idea/junit.html>
<https://www.jetbrains.com/help/idea/testng.html>
<https://www.jetbrains.com/help/idea/spock.html>
[#create_project_spock](#)

Los Test deben estar en la carpeta src/test/java con una estructura similar a la del src/main/java. Puede que tengas que pulsar sobre el directorio java de Test para marcarlo como directorio de pruebas si el icono de la carpeta java no aparece en verde.



Puedes ver la cobertura de los test . A la derecha aparece la ventana con la cobertura y en el código probado podrás ver las líneas que se han probado (en verde) y las que no (en rojo). También puedes depurar los test

The screenshot illustrates the JUnit test execution environment in IntelliJ IDEA, showing the project structure, test annotations, coverage data, and the source code of the tested class.

Coverage Data:

| Element | Class, % | Method, % | Line, % |
|--------------|------------|------------|-----------|
| prjunit5jmv1 | 100% (1/1) | 100% (3/3) | 83% (5/6) |
| Calculadora | 100% (1/1) | 100% (3/3) | 83% (5/6) |

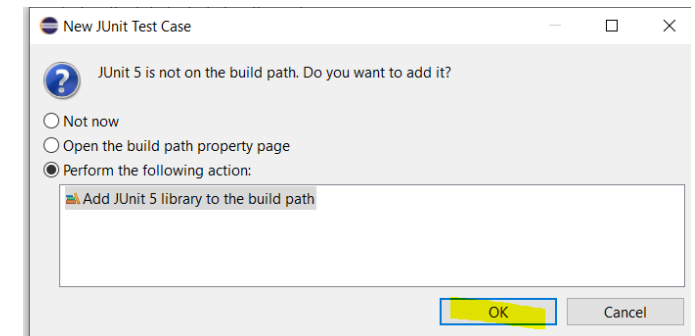
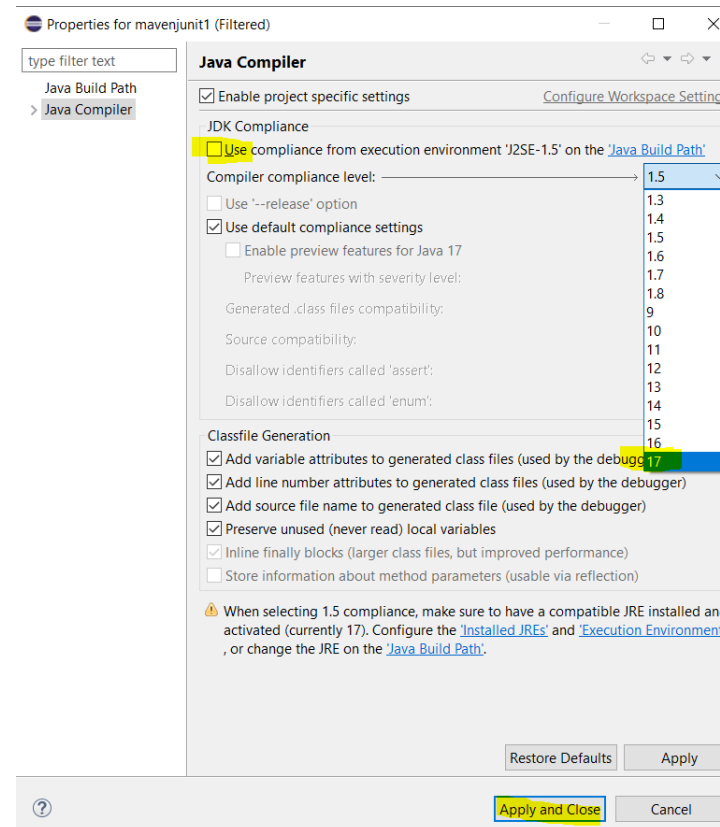
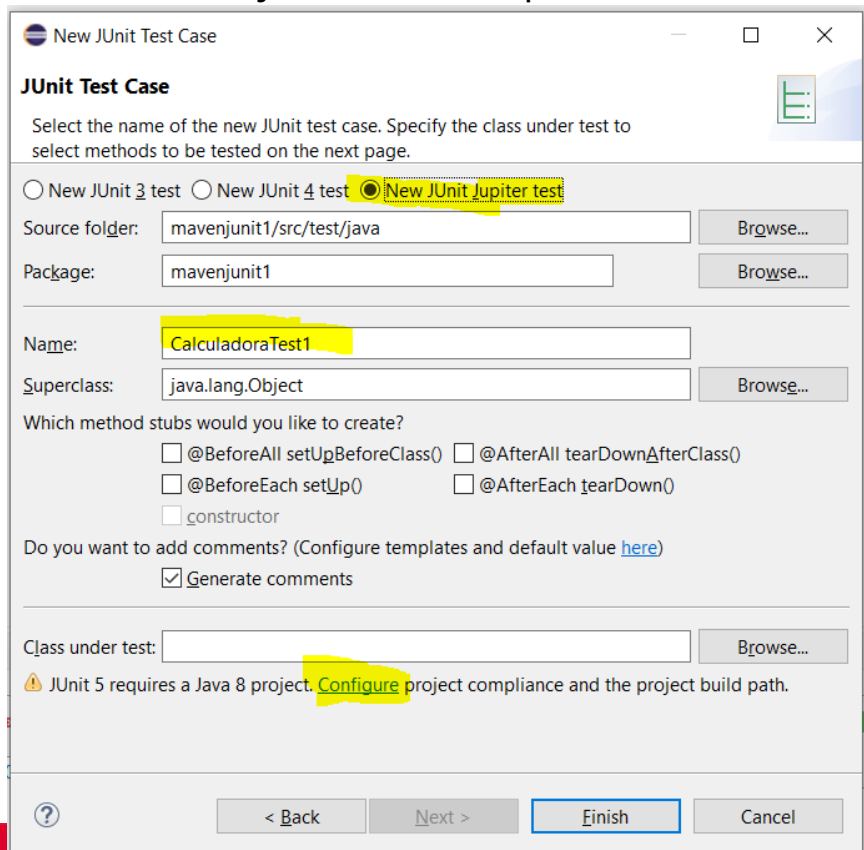
Source Code (Calculadora.java):

```

1 package prjunit5jmv1;
2 public class Calculadora {
3
4     * Codeium: Refactor Explain Docstring
5     int dividir(int x, int y) throws Exception{
6         if (y!=0)
7             return x/y;
8         else throw new Exception("Error, divisi
9
10    * Codeium: Refactor Explain Docstring
11    int sumar(int x, int y){ return (x+y); }
12    * Codeium: Refactor Explain Docstring
13    int incrementar(int x){
14        return (x+1);
15    }

```

Después elegimos new jupiter test y ponemos un nombre al Fichero de test en name (TestCalculadora1 **es importante que empiece por la palabra Test**). Como para utilizar jupiter necesitamos una versión de java igual o superior a la 8 es posible que, debajo de class under test nos aparezca un mensaje indicando que debemos tener una versión de java compatible . Si es el caso pulsaremos sobre el configure de esa línea y, después de desmarcar la opción de Use compliance from.. elegiremos una versión superior a la 8) y pulsamos apply an close. y en la ventana que aparece decimos Yes a recompilar el proyecto. Al volver a la pantalla de Junit Test Case el mensaje de compatibilidad habrá desaparecido y pulsaremos Finish. En la ventana que aparece añadiremos junit5 al build path.

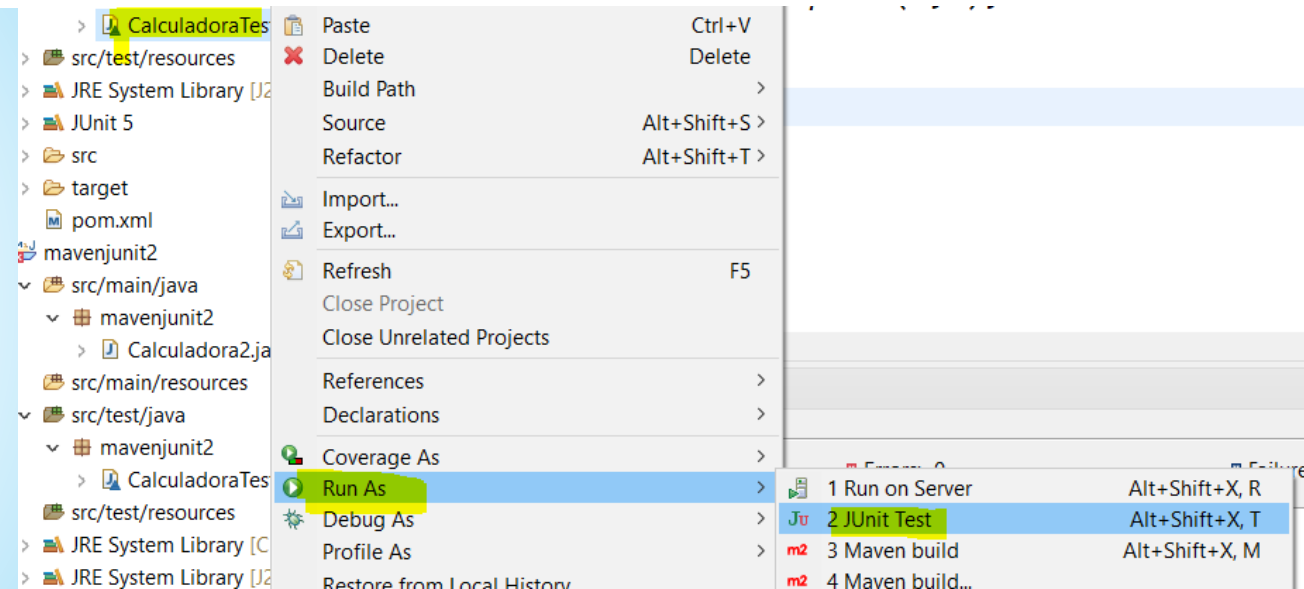


Codificaremos un test en el fichero `src/test/java/mavenjunit1/CalculadoraTest1` (en la misma ruta que la clase que queremos probar pero en la carpeta de test (cambia test por java))

Pulsamos con botón derecho sobre el fichero de test y elegimos Run As -> JUnit Test

```
package mavenjunit1;
import static org.junit.jupiter.api.Assertions.*;
import org.junit.jupiter.api.Test;

class CalculadoraTest1 {
    @Test
    void test() {
        int r=new Calculadora().suma(3,4);
        assertEquals(r,7);
    }
}
```



```
public class TestCalculadora1 {
    static int contPruebas=10;
    @BeforeAll
    static void inicializacion() {
        System.out.println("inicialización");
        contPruebas=0;
    }
    @Test
    @DisplayName("test con el display")
    void test1() {
        int r=new Calculadora().suma(3,4);
        assertEquals(r,7);
    }
    @Test
    void test2() {
        int r=new Calculadora().suma(3,4);
        assertEquals(r,7);
    }
}
```

```
@RepeatedTest(2)
void test3() {
    int r=new Calculadora().suma(3,4);
    System.out.println("Test repetido "+contPruebas++);
    assertEquals(r,7);
}

@ParameterizedTest
@CsvSource({"6,3,9","7,9,16"})
void test4(int a,int b, int c) {
    int r=new Calculadora().suma(a,b);
    System.out.println("Test con params
"+a+"+"+b+"="+c);
    assertEquals(r,c);
}
}
```

La forma de verificar que se **lanza una excepción** ha cambiado con JUnit5, y ahora se usa una lambda para su correcta verificación. <https://howtodoinjava.com/junit5/expected-exception-example/>

Código en la clase Calculadora:

```
int dividir(int x, int y) throws Exception{
    if (y!=0)
        return x/y;
    else throw new ArithmeticException("Division por zero");
}
```

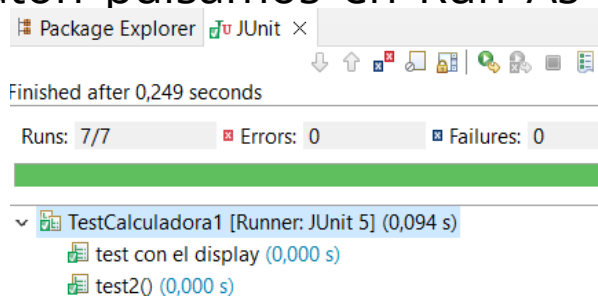
Test :

```
@DisplayName("Test divide por cero")
```

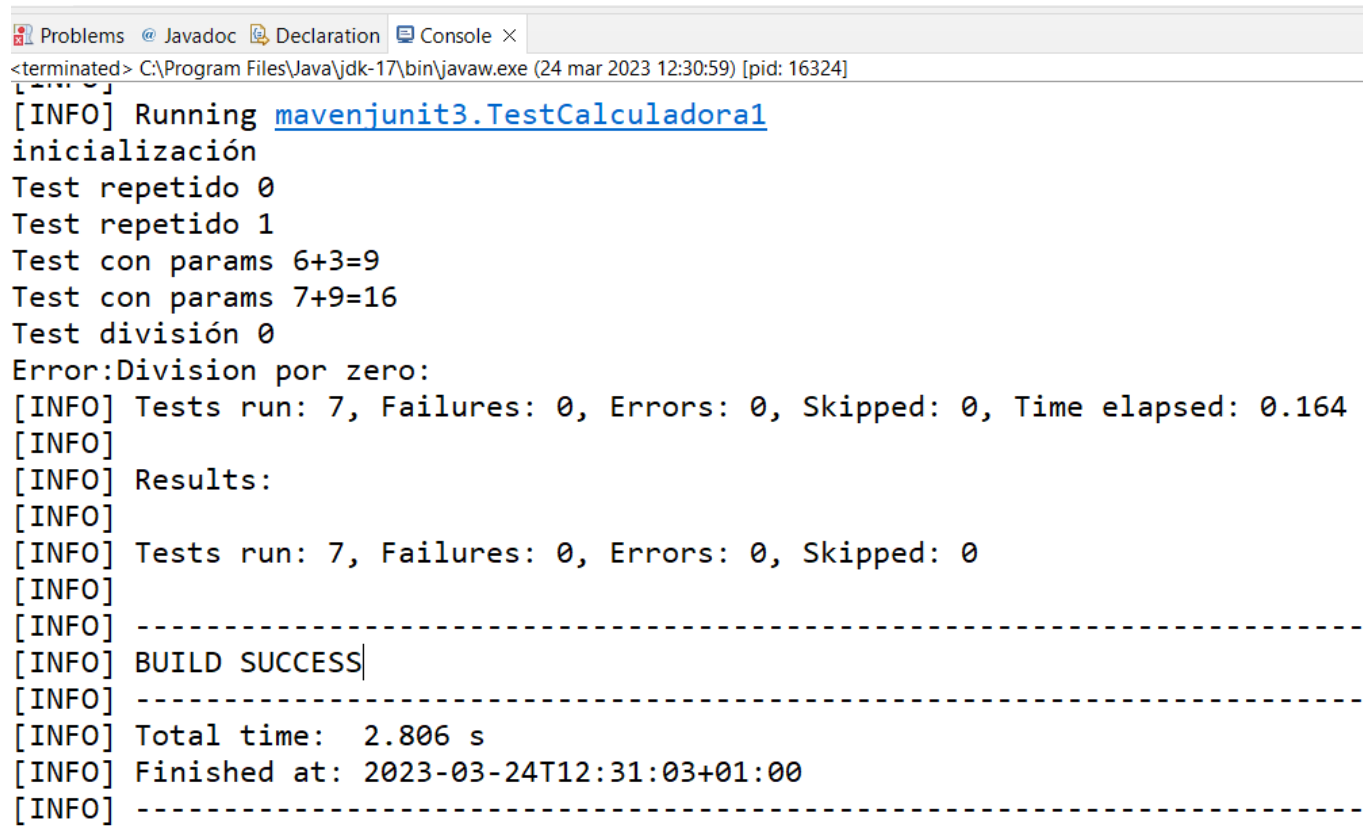
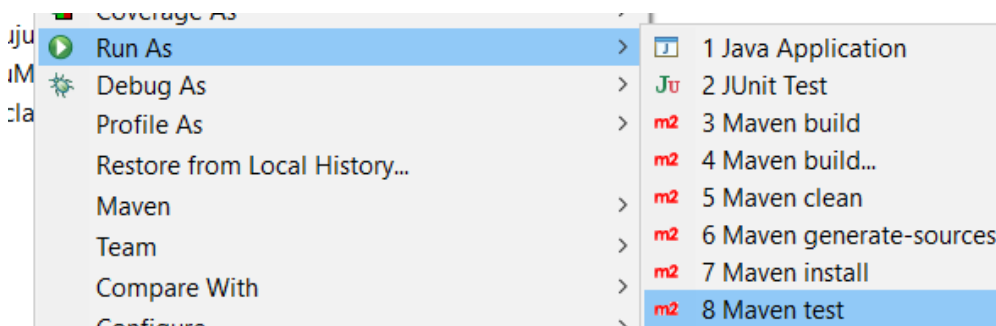
```
@Test
```

```
void divide_by_zero_test() {
    Exception e = Assertions.assertThrows(ArithmeticException.class, () ->
        calculadora.division(2, 0));
    Assertions.assertEquals("Division por zero", e.getMessage());
}
```

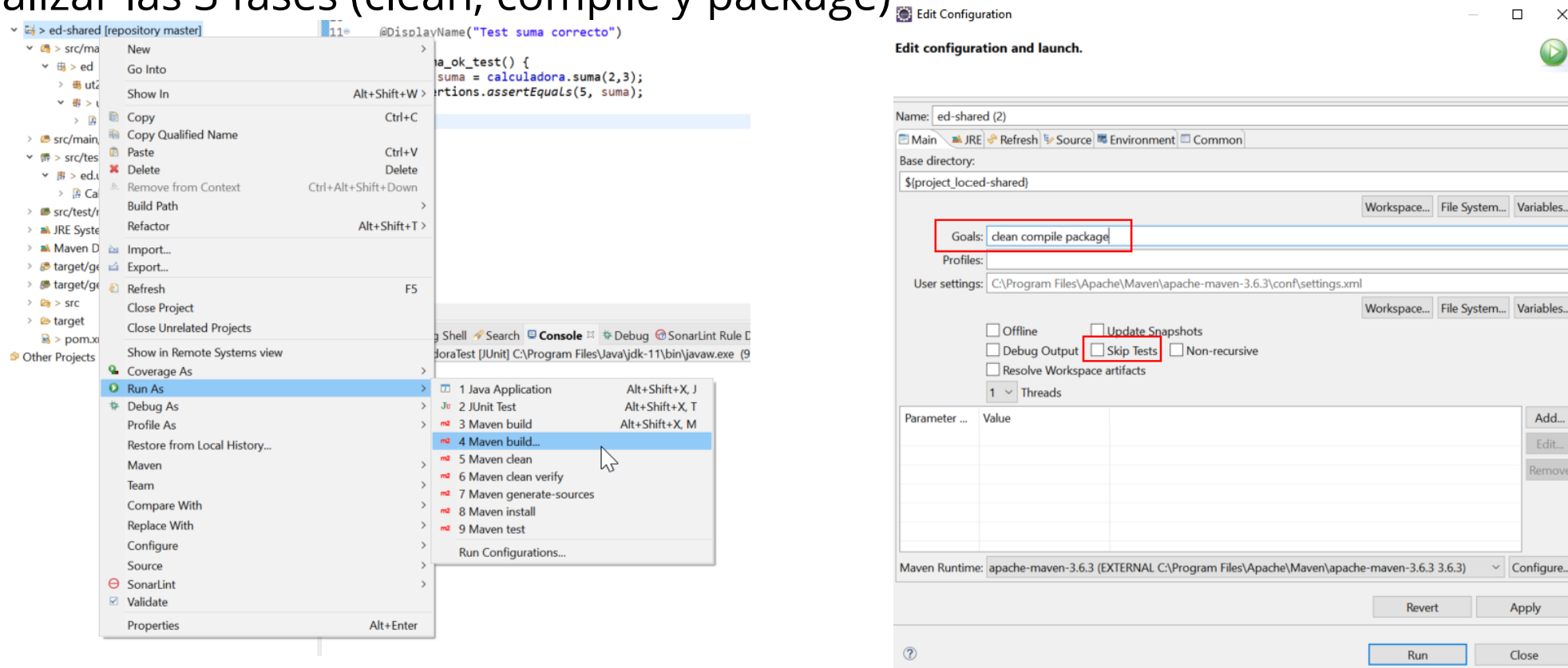
Una vez creado el test, para ejecutarlo podemos usar el IDE eclipse, para ello pulsamos con el botón derecho en la clase y con el ratón pulsamos en Run As > Junit Test. Si queremos hacer un debug, pulsaremos en Debug as > Junit Test



También podemos ejecutar los test desde Maven pulsando sobre el nombre del proyecto y eligiendo Run as->Maven Test . Aparecerá en la consola el resultado:



También podemos ejecutar todos los tests con Maven, simplemente ejecutando un Run as->Maven build (2 opción) indicando como meta (goal) clean compile package para realizar las 3 fases (clean, compile y package).



Obsérvese que Skip Tests está desmarcado. Si quisiéramos saltarnos el paso de los tests, deberíamos marcar esta opción.

Actividad 3. Crea un método que reciba por parámetro 2 números enteros.

- Si son iguales devuelve 0
- Si son diferentes devuelve el mayor.
- Si alguno es 0 devuelve una excepción de tipo Exception.

Crea los test JUnit

Actividad 4. Crea un proyecto de una calculadora con Maven y JUnit. La calculadora deberá hacer las operaciones de sumar, restar, multiplicar y dividir. Si se intenta dividir por cero deberá dar un ArithmeticException.

Actividad 5. Crea un método que reciba por parámetro un valor entero perteneciente a una fecha y te diga si es correcta. Puedes utilizar la clase Fecha vista en clase.
Crea los test JUnit.

Actividad 6. Crea un programa que reciba dos parámetros enteros y devuelva todos los números primos entre esos dos valores. Un número primo es un numero que sólo es divisible por si mismo y por uno. Por definición ni el 0 ni el 1 es un número primo.
Crea los test Junit para el programa.

La **cobertura de código (coverage)** es una medida porcentual en las pruebas de software que mide el grado en que el código fuente de un programa ha sido comprobado. Es comúnmente utilizada en pruebas de caja blanca, como las pruebas unitarias, en las que sí se tiene acceso al código y estructura del software que se está testeando.

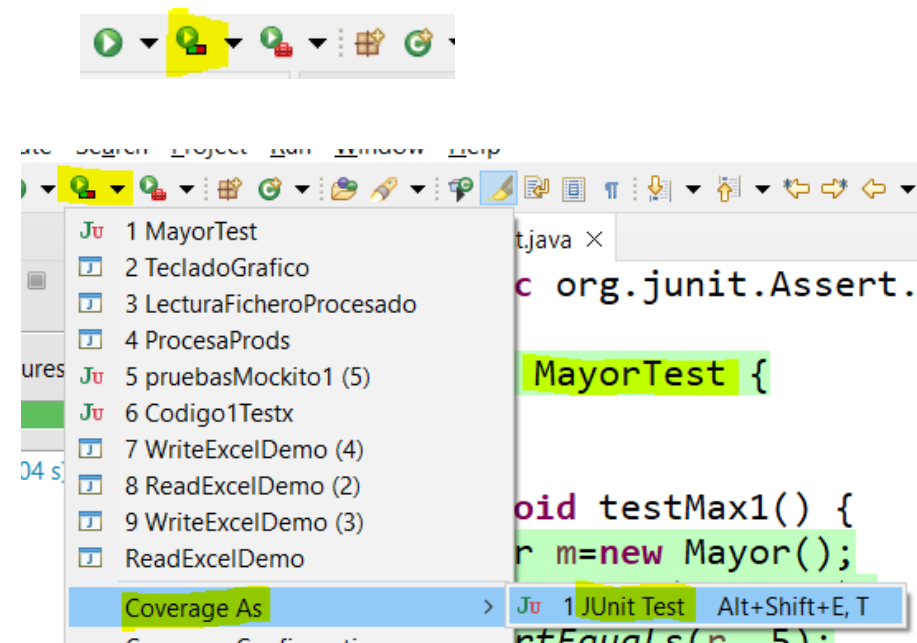
Con Maven tenemos algunos plugin para obtener la cobertura de código y prácticamente con cualquier IDE tienes herramientas que te indican la cobertura, nosotros usaremos la integrada en el propio IDE Eclipse para verificar la cobertura de código.

Para ello haremos lo siguiente, en la clase de Test pulsaremos con el botón derecho y seleccionaremos la opción Coverage As > Junit Test. También podemos pulsar en Run > Coverage As > Junit Test.

Utilizando [EclEmma](#) (herramienta que debemos instalar del Help->Eclipse Marketplace podemos ver la cobertura de las pruebas realizando la ejecución con el icono a la derecha del de ejecución

Con esto podremos ver el porcentaje de cobertura de código que tenemos y visualmente [que código está testeado](#) (verde), código parcialmente cubierto (amarillo) y que código no está testeado (rojo). También aparecen rombos coloreados con el mismo significado para las decisiones

```
1 import java.io.BufferedReader;
4 public class Mayor {
5     public int max (int x, int y, int z){
6         int max = 0;
7         if (x>y && x>z) {
8             max = x;
9         } else {
10            if (z>y) {
11                max = z;
```



| MayorTest (7 mar 2023 8:50:26) | | | | |
|--------------------------------|----------|-------------------|--------------------|-------|
| Element | Coverage | Covered Instru... | Missed Instruct... | Total |
| ▼ ProjectPruebaExam2ev | 89,5 % | 51 | 6 | |
| ▼ src | 89,5 % | 51 | 6 | |
| ▼ (default package) | 89,5 % | 51 | 6 | |
| > Mayor.java | 75,0 % | 18 | 6 | |
| > MayorTest.java | 100,0 % | 33 | 0 | |
| | | | | |
| | | | | |
| | | | | |
| | | | | |
| | | | | |
| | | | | |
| | | | | |
| | | | | |

La **cobertura de código** nos da un índice de que código está probado pero no nos dice si las pruebas realizadas son las correctas o no, para ello habría que profundizar mas en temas como [mutation tests](#) (añadir modificaciones erróneas en el código original y comprobar si los test detectan esos errores) y diseño de pruebas.

Otra framework que sirve para pruebas, en java, en lugar de JUnit es [TestNG](#)

Existen frameworks como [Hamcrest](#) o [AssertJ](#) que proporcionan sistemas de pruebas que amplían las posibilidades de JUnit, facilitando la comprensión de las pruebas, por ejemplo, mejorando la sintaxis del assertThat . Mira algún ejemplo en <https://assertj.github.io/doc/#assertj-guava-quick-start>

Para aplicaciones móviles se suele utilizar [JUnit4 en lugar de JUnit 5](#)

- [Inversión de control](#). Cesión del control del flujo de ejecución. En lugar de realizar llamadas a los diferentes métodos se especifican respuestas deseadas a sucesos o solicitudes de datos concretas y algún framework (ej. Spring) lleva a cabo las acciones.
- [Inversión de dependencias](#). Se deben utilizar abstracciones (ej. interfaces) que eviten las dependencias de los módulos de nivel superior de los de nivel superior, y viceversa. Si cambia la implementación del nivel inferior el superior no se debe ver afectado. Por ej. el constructor de la clase se le pasan las dependencias que se necesitan con interfaces.
- [Inyección de dependencias](#). Suministras los objetos a la clase en lugar de ser la propia clase la que los crea. Por ejem. en la clase DAOProductos tienes un atributo `private List<Producto> listaProd;` y tienes un constructor `DAOProductos(List<Producto> lp) {listaProd=lp}.` De igual forma `class ServicioProductos{ DAOProductos d; ServicioProductos(DAOProductos dp){d=dp}.` En el main habría que hacer muchos `new`s para crear los diferentes objetos por eso los frameworks (Spring, Google Guice, Weld, Dagger Hilt) facilitan la inyección de dependencias creando automáticamente los objetos cuando son necesarios (utilizando anotaciones `@Inject` para indicarlo)

De esta forma cuando realices las pruebas tienes todo el control sobre lo que vas a probar y para probar un método no dependes de otros.

Actividad. Crea las pruebas para añadir productos a la lista de productos de la aplic. del supermercado

- [Weld](#) es la implementación de referencia de CDI, el sistema de inyección de dependencias standard de java.
- Debes incluir, en el pom.xml la dependencia adjunta
- Debes crear el fichero beans.xml en el directorio src/main/resources/META-INF

```
<dependency>
  <groupId>org.jboss.weld.se</groupId>
  <artifactId>weld-se-core</artifactId>
  <version>5.1.2.Final</version>
</dependency>
```

- Además de la anotación **@Inject** delante de los constructores o de los set o de la declaración de atributos para que se realicen los news sin que los haga yo manualmente, se puede usar el **@Singleton** delante de la definición de la clase para que los news en ella se traten con el patrón singleton y solo se cree un objeto de la misma. Si la clase está en mi paquete se utiliza @Inject, pero si está en otro paquete o se crean sin constructor (return xx.getInstance())o necesita inicialización se utiliza **@Produces**
@Inject

```
private final ImageFileEditor imageFileEditor;
```

```
@Inject
```

```
public void setImageFileEditor(ImageFileEditor imageFileEditor) { ... }
```

```
<beans xmlns="http://xmlns.jcp.org/xml/ns/javaee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance"

  xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
http://xmlns.jcp.org/xml/ns/javaee/beans_1_1.xsd"
  bean-discovery-mode="all"
  version="1.1">
</beans>
```

<https://www.baeldung.com/java-ee-cdi>

- Cuando necesite hacer una inyección pongo @Inject antes del constructor (o get) y se creará un objeto como el que recibe el método como parámetro. Si recibe varios parámetros se construirá un objeto de cada uno de ellos.
- En el main, si haces new de los objetos no se utilizará en inyector de dependencias, pero si se los pides al container con el get será Weld, al hacer el get, el encargado de hacer los news de los objetos que se necesiten en la inyección de dependencias.

```
public class ImageFileProcessor {  
    private ImageFileEditor imageFileEditor;  
    @Inject  
    public ImageFileProcessor(ImageFileEditor imageFileEditor) {  
        this.imageFileEditor = imageFileEditor;    }  
    public static void main(String[] args) {  
        //Inicializo el inyector de dependencias  
        Weld weld = new Weld();  
        WeldContainer container = weld.initialize();  
        ImageFileProcessor imageFileProcessor = container.select(ImageFileProcessor.class).get();  
        System.out.println(imageFileProcessor.imageFileEditor.abrirFichero("file1.png"));  
        container.shutdown();  
    }  
}
```

Mockito

<https://www.baeldung.com/mockito-series>

<https://devs4j.com/2018/04/23/pruebas-unitarias-parte-2-junit-y-mockito-primeros-pasos/>

<https://www.baeldung.com/mockito-junit-5-extension>

Está pensado para realizar pruebas unitarias eliminando las referencias a servicios externos a la clase en la que se realiza la prueba.

Se usa cuando se programa con [inyección de dependencias](#) y tenemos una clase que necesita de otra para que realiza una parte del trabajo. Al utilizar inyección de dependencias la clase que da servicio se coloca como atributo en la clase cliente y se añade al cliente en el constructor. [Es diferente a la inversión de dependencias](#)

Mockito permite probar métodos sustituyendo las llamadas a la clase que da servicio por un mock, un sustituto del servicio que devolverá el valor que nosotros decidamos en el test sin necesidad de llamar al servicio real.

Creamos un proyecto Maven (simple Project) para java 8:

```
<version>0.0.1-SNAPSHOT</version>
```

```
<properties>
```

```
<java.version>1.8</java.version>
```

```
<maven.compiler.target>1.8</maven.compiler.target>
```

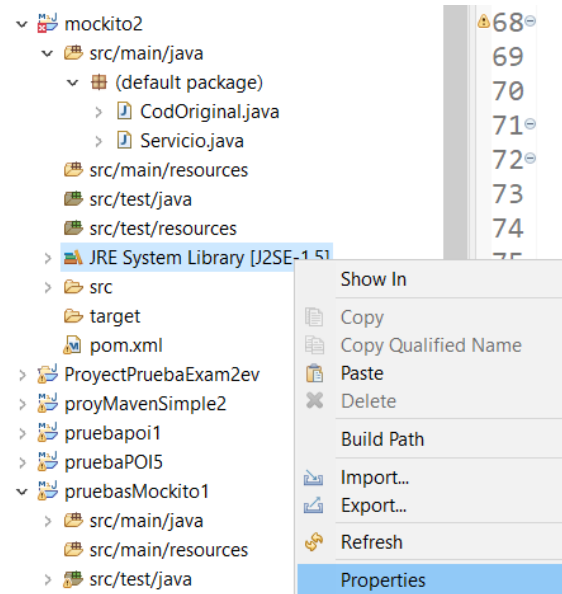
```
<maven.compiler.source>1.8</maven.compiler.source>
```

```
<project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
```

```
</properties>
```

☒ Create a simple project (skip archetype selection)

Cambiarlo desde eclipse:



JRE System Library

Select JRE for the project build path.

System library

- ☒ Execution environment: J2SE-1.5 (jdk-17) ▼
- ☐ Alternate JRE:
- ☐ Workspace default JRE (j
- CDC-1.1/Foundation-1.1 (jdk-17)
 - JRE-1.1 (jdk-17)
 - OSGi/Minimum-1.1 (jdk-17)
 - J2SE-1.2 (jdk-17)
 - OSGi/Minimum-1.2 (jdk-17)
 - J2SE-1.3 (jdk-17)
 - J2SE-1.4 (jdk-17)
 - J2SE-1.5 (jdk-17)
 - JavaSE-1.6 (jdk-17)
 - JavaSE-1.7 (jdk-17)
 - JavaSE-1.8 (jdk-17)
 - JavaSE-9 (jdk-17)
 - JavaSF-10 (jdk-17)

Añadimos las dependencias:

```
<dependencies>
  <dependency>
    <groupId>org.mockito</groupId>
    <artifactId>mockito-core</artifactId>
    <version>5.1.1</version>
    <scope>test</scope>
  </dependency>
  <dependency>
    <groupId>org.mockito</groupId>
    <artifactId>mockito-junit-jupiter</artifactId>
    <version>5.1.1</version>
    <scope>test</scope>
  </dependency>
  <dependency>
    <groupId>org.junit.jupiter</groupId>
    <artifactId>junit-jupiter-engine</artifactId>
    <version>5.9.2</version>
    <scope>test</scope>
  </dependency>
```

```
<dependency>
  <groupId>org.junit.jupiter</groupId>
  <artifactId>junit-jupiter-params</artifactId>
  <version>5.9.2</version>
  <scope>test</scope>
</dependency>
<dependency>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-surefire-plugin</artifactId>
  <version>3.0.0-M9</version>
</dependency>
<dependency>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-compiler-plugin</artifactId>
  <version>3.10.1</version>
</dependency>
</dependencies>
```

Con surefire puedes correr todas las pruebas, pero si el IDE te lo permite no haría falta.

Añadimos los plugins:

```
<build>
  <plugins>
    <plugin> <!-- Need at least 2.22.0 to support JUnit 5 -->
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-surefire-plugin</artifactId>
      <version>3.0.0-M9</version>
    </plugin>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-compiler-plugin</artifactId>
      <version>3.10.1</version>
    </plugin>
  </plugins>
  <plugin>
    <artifactId>maven-surefire-plugin</artifactId>
    <version>2.22.2</version>
    <dependencies>
      <dependency>
        <groupId>org.junit.platform</groupId>
        <artifactId>junit-platform-surefire-provider</artifactId>
        <version>1.3.2</version>
      </dependency>
    </dependencies>
  </plugin>
</build>

<!-- no incluir la etiqueta del fin de Project si ya está en
el fichero xml -->
</project>
```

Codificamos las clases:

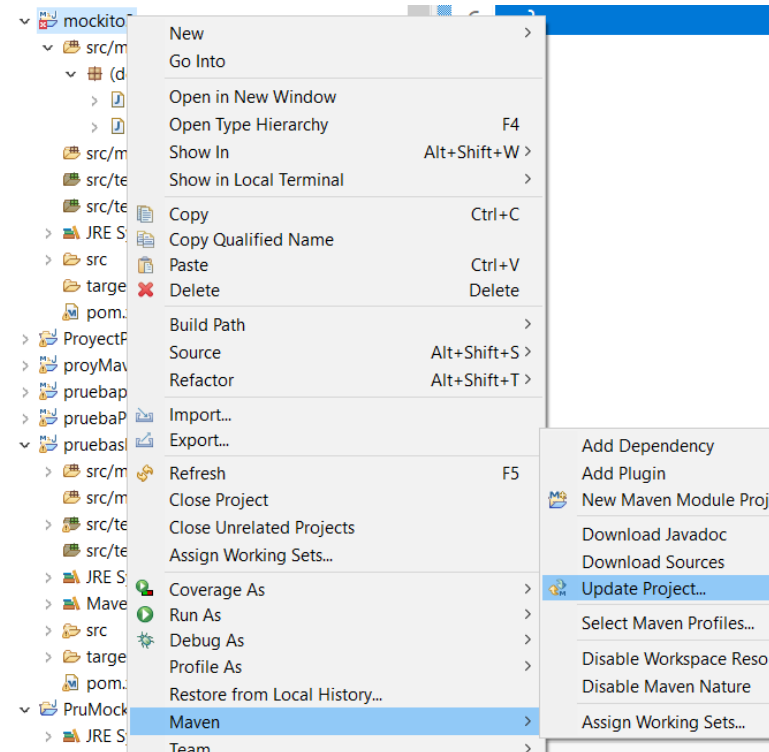
```
public class CodOriginal {
    Servicio serv;

    public CodOriginal(Servicio serv) {
        this.serv = serv;
    }

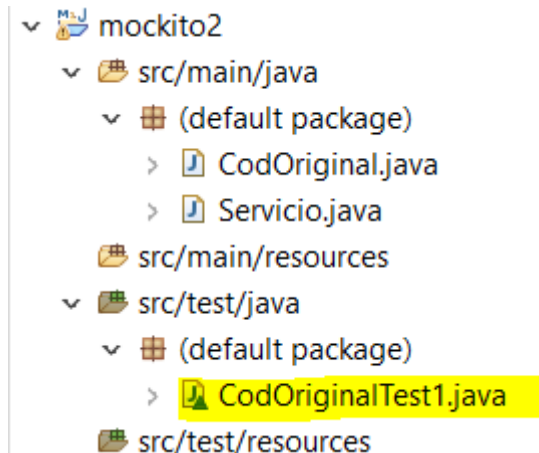
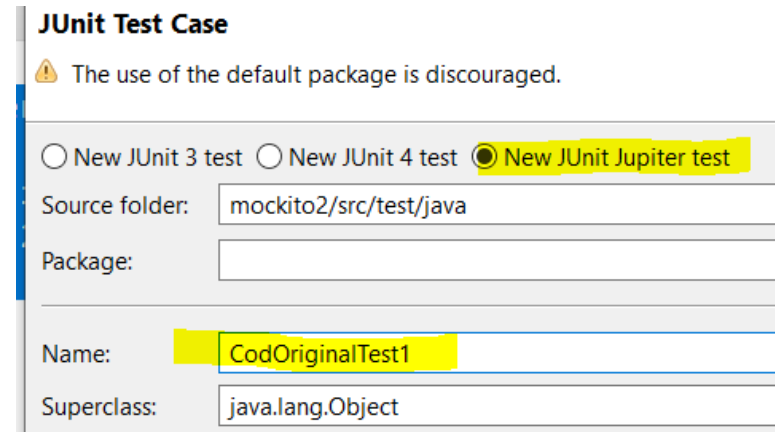
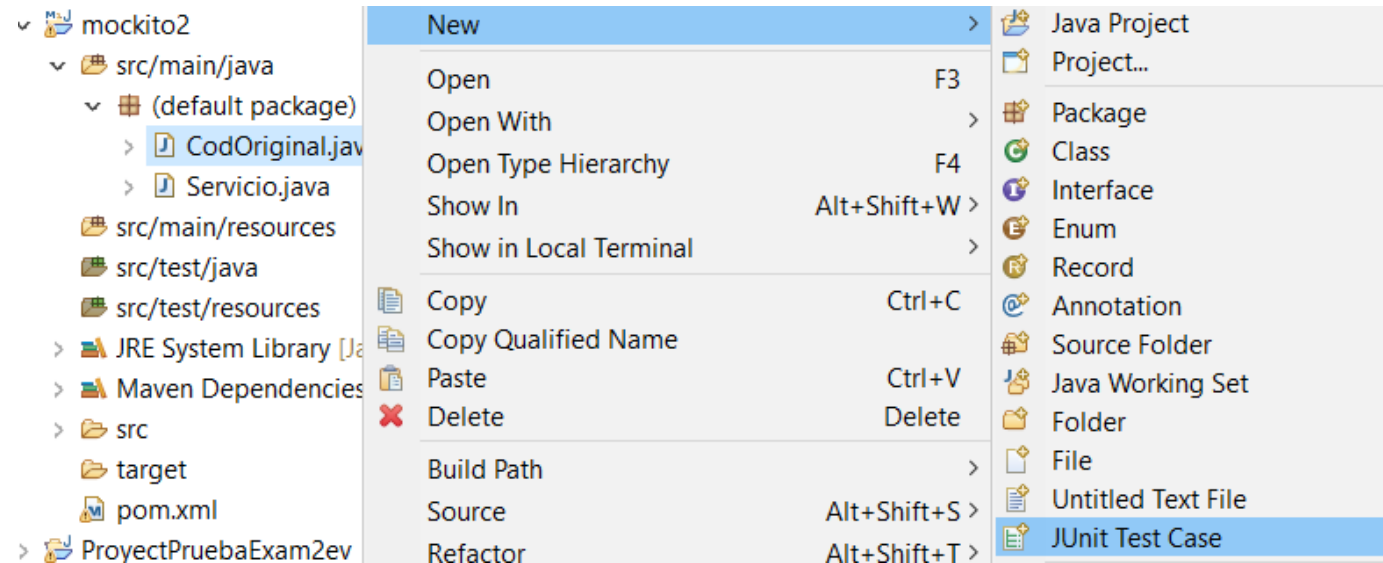
    int usoServicio(int x) {
        int r=x+serv.darServicio(x);
        return r;
    }
}
```

```
public class Servicio {
    int x=3;
    int darServicio(int a){
        return a*2+x;
    }
}
```

Actualizamos el proyecto->



Generamos un test



Codificamos el test

```

@ExtendWith(MockitoExtension.class)
class CodOriginalTest1 {
    @InjectMocks
    private CodOriginal cod;

    //Podemos definir el mock aquí o posteriormente

    @Mock
    private Servicio serv;

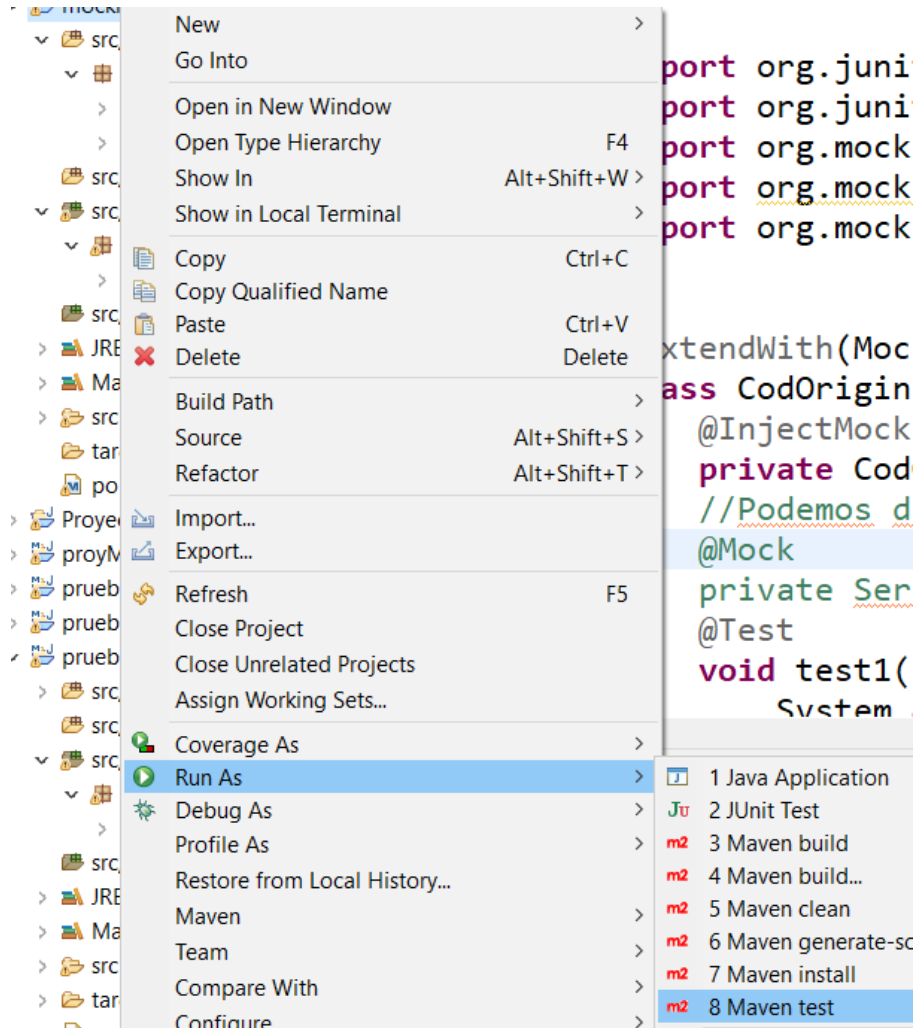
    @Test
    void testMock1() {
        System.out.println( "Test mockito Inicio");
        //Creamos el mock
        serv = mock(Servicio.class);
        //Creamos el objeto original
        cod=new CodOriginal(serv);
        when (serv.darServicio(7)).thenReturn(754);
        int r= cod.usoServicio(7);
        System.out.println( "Test mockito resultado "+r);
        assertEquals (761, r);
    }
}

```

y lo ejecutamos

The screenshot illustrates the process of running a Mockito test in an IDE. The Package Explorer on the left shows the project structure, with 'CodOriginalTest1.java' selected. A context menu is open over the file, with 'Run As' highlighted. The 'Run As' submenu is also visible, showing options like '1 Run on Server', '2 JUnit Test', and '3 Maven build'. The Console window at the bottom shows the output of the test: 'Java HotSpot(TM) 64-Bit Server', 'Test mockito Inicio', and 'Test mockito resultado 760'. The Package Explorer on the right shows the test results for 'CodOriginalTest1' and 'testMock10'.

Compilamos, primero con Maven y después ejecutamos el junit test



```
Problems @ Javadoc Declaration Console ×
<terminated> C:\Program Files\Java\jdk-17\bin\javaw.exe (24 mar 2023 13:02:11) [pid: 11572]

[INFO] Running mockitomaven2.CodOriginalTest
Test mockito Inicio
Test mockito resultado 761
[INFO] Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time e
[INFO]
[INFO] Results:
[INFO]
[INFO] Tests run: 1, Failures: 0, Errors: 0, Skipped: 0
[INFO]
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
[INFO] Total time: 8.456 s
```

Calidad del software

La calidad es un tema que, desde hace años, tiene una importancia en el mundo de la comercialización de productos. El mercado actual es muy competitivo y la calidad es uno de los aspectos diferenciales que hace que un producto triunfe o fracase. Basta citar a **Blackberry**, empresa que, en medio de la vorágine de un mercado tan competitivo como el de la telefonía móvil, cometió varios **errores** tanto estratégicos como **técnicos** que la relegaron a un plano poco significativo.

Dadas sus características, garantizar la calidad del software es un proceso mucho más difícil que el de otro producto, dado que un proceso industrial es más fácil de testear que el proceso de desarrollo de software. El software tiene que estar libre de defectos y de errores y también tiene que adecuarse a los parámetros con los cuales se ha diseñado y desarrollado.

Las aplicaciones informáticas están presentes en multitud de ámbitos. Existe software en dispositivos que nadie puede imaginarse (lavadoras, televisiones, aires acondicionados, etc.).

Un proceso de desarrollo de calidad es básico para obtener un producto de calidad

En los años noventa, se vivió una crisis del software. Fueron en esos años en los que la calidad y **el proceso de desarrollo** no tenían mucha importancia en los que se vivieron las consecuencias de desarrollar un software con poca profesionalidad en muchos casos. Algunas características de esta crisis fueron:

- Calidad insuficiente del producto final. Muchos de los errores tenían su base en un análisis pobre con una poca comunicación con el cliente.
- **Estimaciones de duración de proyectos y asignación de recursos inexactas.** Con el problema que ello conlleva.
- **Escasez de personal cualificado** en un mercado laboral de alta demanda. Algunos programas no estaban desarrollados bajo el paradigma de la programación estructurada. No tenían una estructura racional ni lógica, con lo cual los errores se multiplicaban y el mantenimiento era un suplicio.
- Tendencia al **crecimiento del volumen y complejidad de los productos.** En algunos casos, dichos desarrollos complejos estaban poco probados, pobremente documentados, etc.

Con el tiempo, se ha constatado que la **calidad** no se mide solamente por unos parámetros de funcionamiento, sino que hay otros aspectos que son importantes, como el **soporte**, es decir, el respaldo organizacional que tiene un producto como la **formación**, la **asistencia** a problemas inesperados y el **mantenimiento** permanente y efectivo.

Para valorar dicha calidad, se lleva a cabo la evaluación y el rendimiento de las aplicaciones. Las **mediciones de rendimiento** de un software pueden estar **orientadas hacia el usuario** (tiempos de respuesta) u **orientadas hacia el sistema** (uso de la CPU). Son medidas típicas del rendimiento diferentes variables de tiempo (tiempo de retorno, tiempo de respuesta y tiempo de reacción), la capacidad de ejecución, la carga de trabajo, la utilización, etc.

Para evaluar el software, es necesario contar con criterios adecuados que permitan analizar el software desde diferentes puntos de vista.

Las **pruebas de carga** se realizan sobre el sistema simulando una serie de peticiones esperadas o un número de usuarios esperado trabajando de forma concurrente, realizando un número de transacciones determinado. En estas pruebas, se evalúan los tiempos de respuesta de las transacciones. Generalmente, se realizan varios tipos de carga (baja, media y alta) para evaluar el impacto y poder graficar el rendimiento del sistema.

Otro tipo de pruebas bastante útiles son las **pruebas de estrés** en las que la carga va elevándose más y más para ver cómo de sólida es la aplicación y cómo se maneja ante un número de usuarios y transacciones extremos.

También existen otros tipos de pruebas como las **pruebas de estabilidad** donde se somete de forma continuada al sistema a una carga determinada o bien pruebas de picos donde el volumen de carga va cambiando.

Se definen los criterios de calidad (o factores de calidad) de un software al principio de un proyecto y dichos criterios siguen teniéndose en cuenta durante toda su vida. No puede existir ningún criterio o factor de calidad que no pueda medirse. Algunos **criterios de calidad** pueden ser los siguientes:

- **Número de errores por un número determinado de líneas** de código.
- Número de **tiempo que la aplicación estará dando servicio**.
- **Número** medio de **revisiones** realizadas a una función o módulo de programa.

Generalmente, para evaluar los criterios de calidad, se realizan RTF o revisiones técnicas formales.

Los criterios o factores de calidad, como no podía ser de otra forma, se establecen mediante métricas o medidas. Véanse algunas de las métricas de calidad más utilizadas:

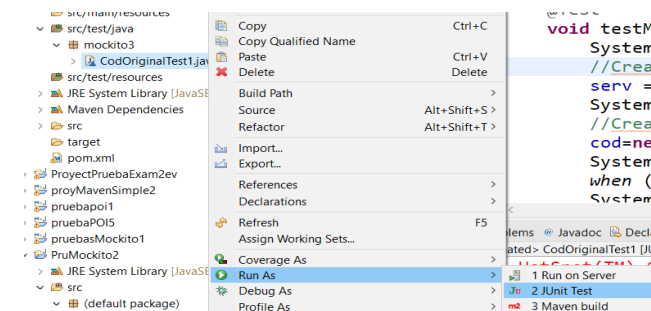
1. **Tolerancia a errores.** Mide los efectos que tiene un error sobre el software en conjunto. El objetivo es que no haya **errores**, pero, si los hay, que sus **efectos sean limitados**.
2. **Facilidad de expansión.** Mide la facilidad con la que pueden **añadírsele nuevas funcionalidades** a un software concreto. Cuanto más fácil sea de ampliar, mejor.
3. **Independencia de plataforma del hardware.** Es sabido que un programa en Java es de los más independientes que existen. Cuanto mayor sea el número de plataformas donde pueda ejecutarse un software, mejor.
4. **Modularidad.** Número de **componentes independientes** de un programa.
5. Estandarización de los datos. Se evalúa si se utilizan **estructuras de datos estándar** a lo largo de un programa.

Calidad del código

Define mediante métricas y reglas cómo de bien escrito está el código fuente que hemos creado.

Vamos a diferenciar siete grandes puntos que serán los ejes de la calidad del código software.

1. Tener líneas de **código duplicado** ("Duplicated code").
2. **No respetar** los **estándares** de codificación y las mejores prácticas establecidas ("Coding standards").
3. Tener una **cobertura baja** (o nula) **de pruebas** unitarias, especialmente en partes complejas del programa ("Unit tests").
4. Tener **componentes complejos** y/o una mala distribución de la complejidad entre los componentes ("Complex code").
5. Dejar **fallos potenciales sin analizar** ("Potential bugs").
6. **Falta de comentarios** en el código fuente, especialmente en las APIs públicas ("Comments").
7. Tener el temido diseño spaghetti, con multitud de **dependencias cíclicas** ("Design and architecture").



Estos siete ejes son los mismos que utiliza la herramienta Sonar para evaluar el código fuente.

SonarQube (conocido anteriormente como Sonar) es una plataforma para evaluar código fuente. Es software libre y usa diversas herramientas de análisis estático de código fuente como Checkstyle, PMD o FindBugs para obtener métricas que pueden ayudar a mejorar la calidad del código de un programa.



<https://www.sonarqube.org/>

Funciones

- Informa sobre código duplicado, estándares de codificación, pruebas unitarias, cobertura de código, complejidad ciclomática, errores potenciales, comentarios y diseño del software.
- Aunque pensado para Java, acepta otros lenguajes mediante extensiones.
- Se integra con Maven, Ant y herramientas de integración continua como Atlassian Bamboo, Jenkins y Hudson.

Sonar permite además realizar un análisis comparativo de las métricas en el tiempo, lo cual permite disponer de una línea de tiempo y comparar la salud del proyecto de un sprint a otro, hablando en términos de metodologías ágiles, iteraciones y entrega continua.

Para establecer un nivel de calidad mínimo en nuestros proyectos podemos definir el cumplimiento de, al menos, un umbral de dichas métricas.

Principios SOLID (<https://anahisalgado.com/principios-solid/>):

- S – Single Responsibility Principle (SRP)
- O – Open/Closed Principle (OCP)
- L – Liskov Substitution Principle (LSP)
- I – Interface Segregation Principle (ISP)
- D – Dependency Inversion Principle (DIP)

Objetivos :

- Crear un **software eficaz**: que cumpla con su cometido y que sea **robusto y estable**.
- Escribir un **código limpio y flexible** ante los cambios: que se pueda modificar fácilmente según necesidad, que sea **reutilizable** y **mantenible**.
- Permitir **escalabilidad**: que acepte ser ampliado con nuevas funcionalidades de manera ágil.