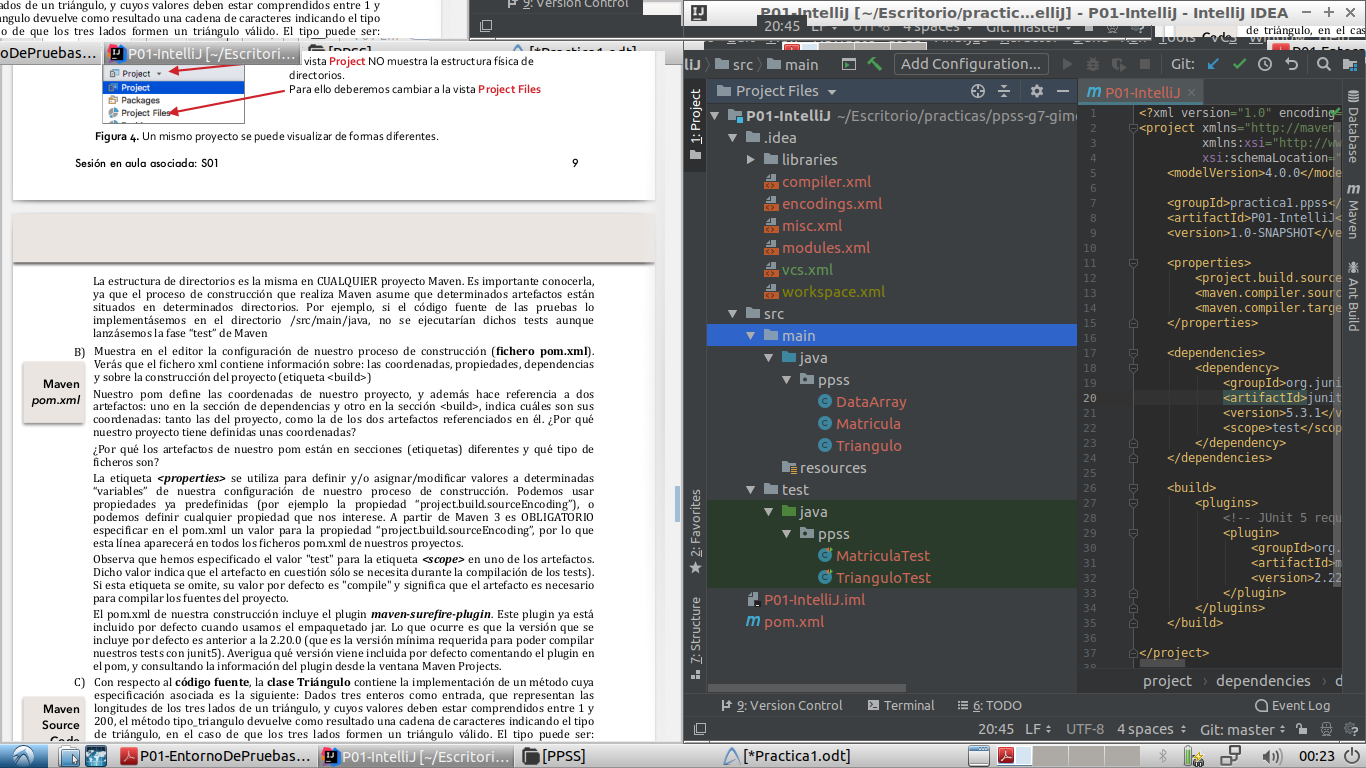
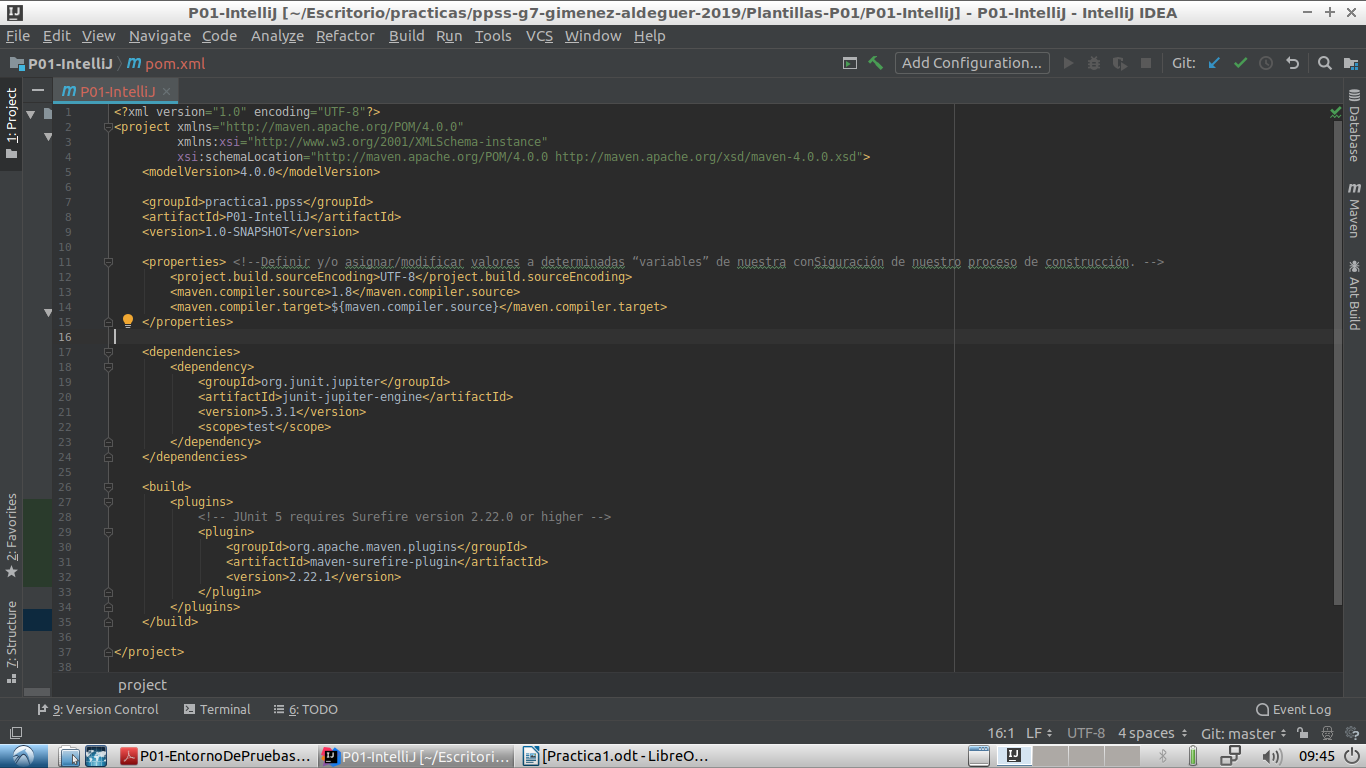
1. **P01a+P01b: Entorno de pruebas**
   1. **Ejercicio 1:**
   2. **A: Anota la estructura de directorios del proyecto.**
   3. 

**B: Muestra en el editor la configuración de nuestro proceso de construcción (-fichero pom.xml).**



Nuestro pom define las coordenadas de nuestro proyecto, y además hace referencia a dos pom.xml artefactos: uno en la sección de dependencias y otro en la sección <build>, **indica cuáles son sus coordenadas: tanto las del proyecto, como la de los dos artefactos referenciados en él.**

Proyecto:

<groupId>practica1.ppss</groupId>

<artifactId>P01-IntelliJ</artifactId>

<version>1.0-SNAPSHOT</version>

Dependencias:

<groupId>org.junit.jupiter</groupId>

<artifactId>junit-jupiter-engine</artifactId>

<version>5.3.1</version>

Procesos de construcción:

<groupId>org.junit.jupiter</groupId>

<artifactId>junit-jupiter-engine</artifactId>

<version>5.3.1</version>

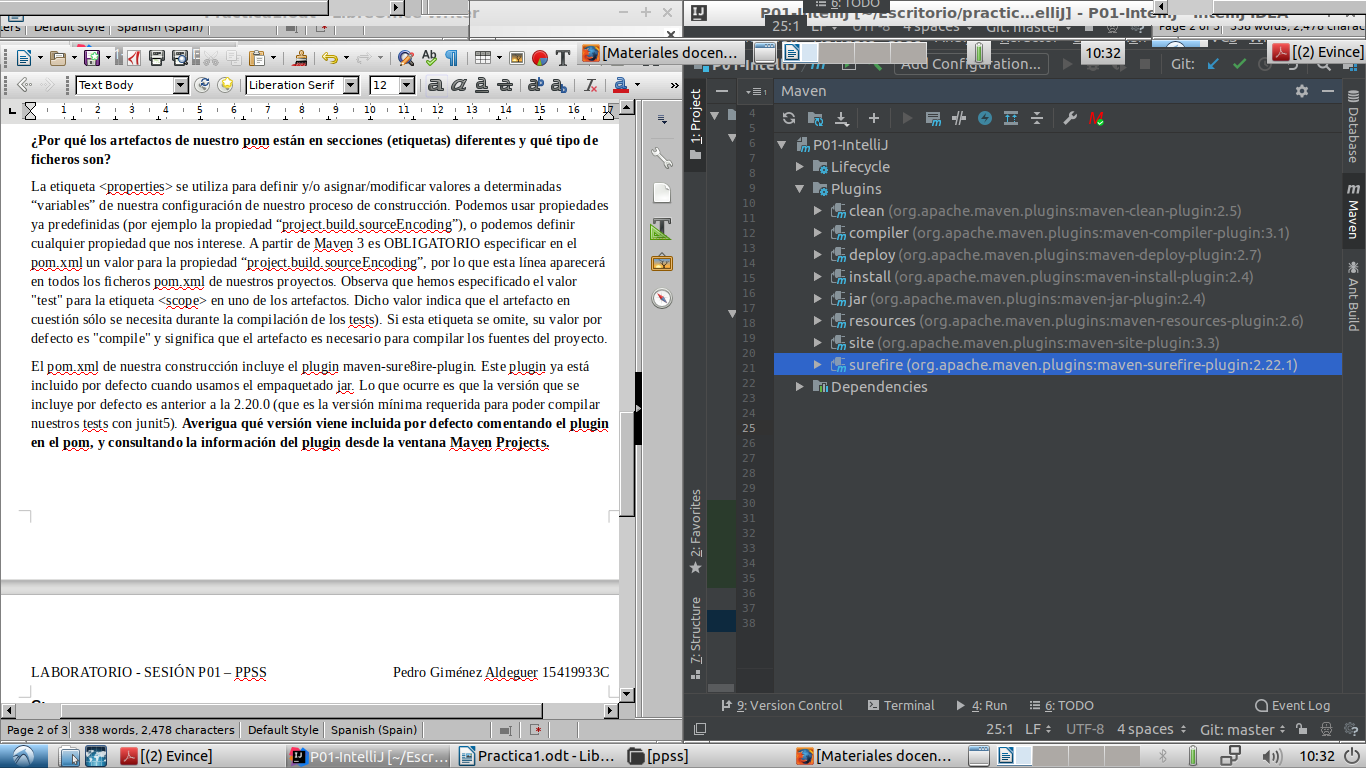
**¿Por qué nuestro proyecto tiene definidas unas coordenadas?**

Porque es necesario para poder identificar un artefacto utilizado por Maven.

**¿Por qué los artefactos de nuestro pom están en secciones (etiquetas) diferentes y qué tipo de ficheros son?**

La etiqueta <properties> se utiliza para definir y/o asignar/modificar valores a determinadas “variables” de nuestra configuración de nuestro proceso de construcción. Podemos usar propiedades ya predefinidas (por ejemplo la propiedad “project.build.sourceEncoding”), o podemos definir cualquier propiedad que nos interese. A partir de Maven 3 es OBLIGATORIO especificar en el pom.xml un valor para la propiedad “project.build.sourceEncoding”, por lo que esta línea aparecerá en todos los ficheros pom.xml de nuestros proyectos. Observa que hemos especificado el valor "test" para la etiqueta <scope> en uno de los artefactos. Dicho valor indica que el artefacto en cuestión sólo se necesita durante la compilación de los tests). Si esta etiqueta se omite, su valor por defecto es "compile" y significa que el artefacto es necesario para compilar los fuentes del proyecto.

El pom.xml de nuestra construcción incluye el plugin maven-sure8ire-plugin. Este plugin ya está incluido por defecto cuando usamos el empaquetado jar. Lo que ocurre es que la versión que se incluye por defecto es anterior a la 2.20.0 (que es la versión mínima requerida para poder compilar nuestros tests con junit5). **Averigua qué versión viene incluida por defecto comentando el plugin en el pom, y consultando la información del plugin desde la ventana Maven Projects.**



Al comentar la línea en vez de 2.22.1 sale 2.12.4.

**C: Con respecto al código fuente, la clase Triángulo contiene la implementación de un método cuya especificación asociada es la siguiente:**

Dados tres enteros como entrada, que representan las longitudes de los tres lados de un triángulo, y cuyos valores deben estar comprendidos entre 1 y 200, el método tipo\_triangulo devuelve como resultado una cadena de caracteres indicando el tipo de triángulo, en el caso de que los tres lados formen un triángulo válido.

El tipo puede ser: “Equilatero”, “Isosceles”, o “Escaleno”. Para que los tres lados proporcionados como entrada puedan formar un triángulo tiene que cumplirse la condición de que la suma de dos de sus lados tiene que ser siempre mayor que la del tercero. Si esto no se cumple, el método devolverá el mensaje “No es triangulo”. Si alguno de los tres lados: a, b, ó c, es mayor que 200 o inferior a 1, mostrará directamente (independientemente de si es triángulo o no) el mensaje “Valor x fuera del rango permitido”, siendo x el carácter a, b, ó c, en función de que sea el primer, segundo, o tercer valor de entrada el que incumpla la condición.

Este es uno de los **ejemplos más utilizados en la literatura sobre pruebas**, quizá porque contiene una lógica clara, pero a la vez compleja. Fue **utilizado por primera vez por Gruenberger en 1973**, aunque en una versión algo más simple.

**D: La clase TrianguloTest contiene la implementación de cuatro casos de prueba (los cuatro métodos anotados con @Test) asociados a la especificación del apartado anterior. ¿Cuáles son exactamente? Identifícalos como C1, C2, C3, y C4, y muéstralos en una tabla con cuatro filas con la siguiente información:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Identificador del Caso de prueba** | **Datos de entrada (todos)** | **Resultado esperado** | **Resultado real** | **Informe** |
| **C1** | a = 1 ; b = 1 ; c = 1 | Equilátero | Equilátero | OK |
| **C2** | a = 1 ; b = 1 ; c = 11 | No triángulo | No triángulo | OK |
| **C3** | a = 1 ; b = 2 ; c = 0 | Fuera de rango | No triángulo | FALLO |
| **C4** | a = 14 ; b = 10 ; c = 10 | Isósceles | Isósceles | OK |

Esta tabla se denomina “Tabla de casos de prueba”. Recuérdala porque la utilizaremos en sesiones posteriores cuando diseñemos los casos de prueba.

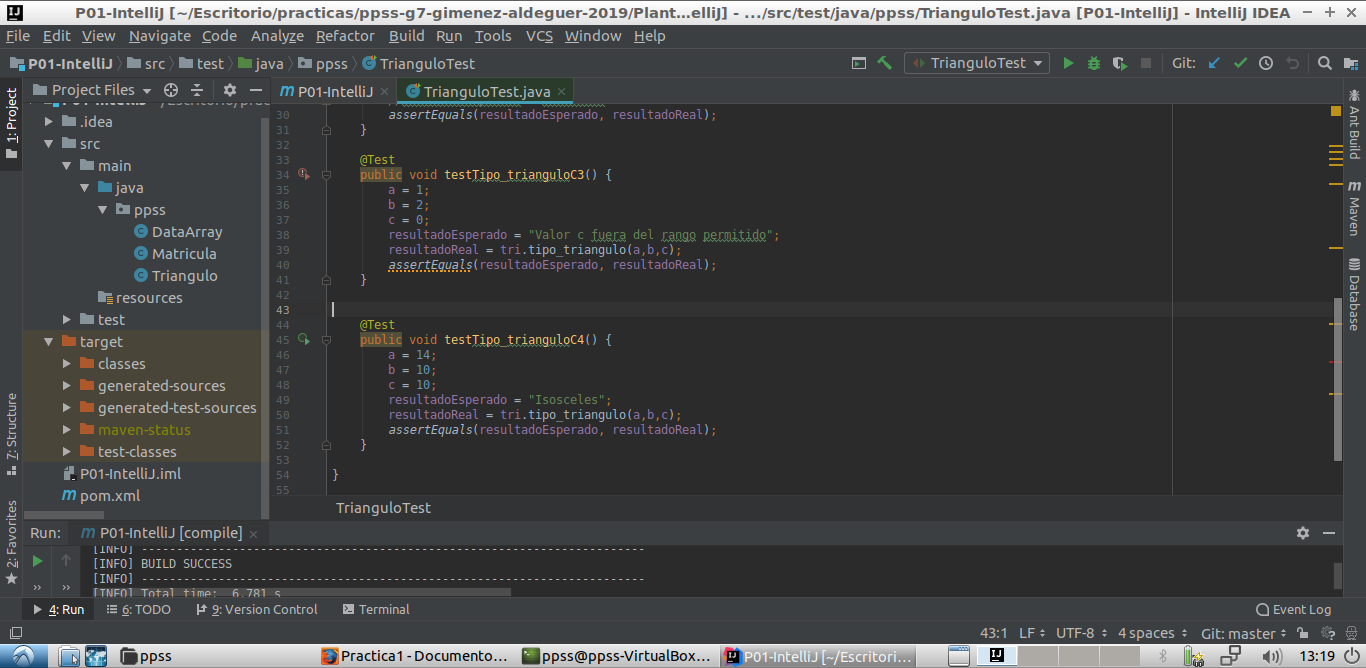
**E:** **Observa la implementación de cada test y verás que todos ellos siguen la misma lógica de programa. Anota el algoritmo que refleja dicha lógica. Recuérdalo porque lo utilizaremos también en sesiones posteriores.**

Darle valor a las variables que vas a utilizar para pasar el test.

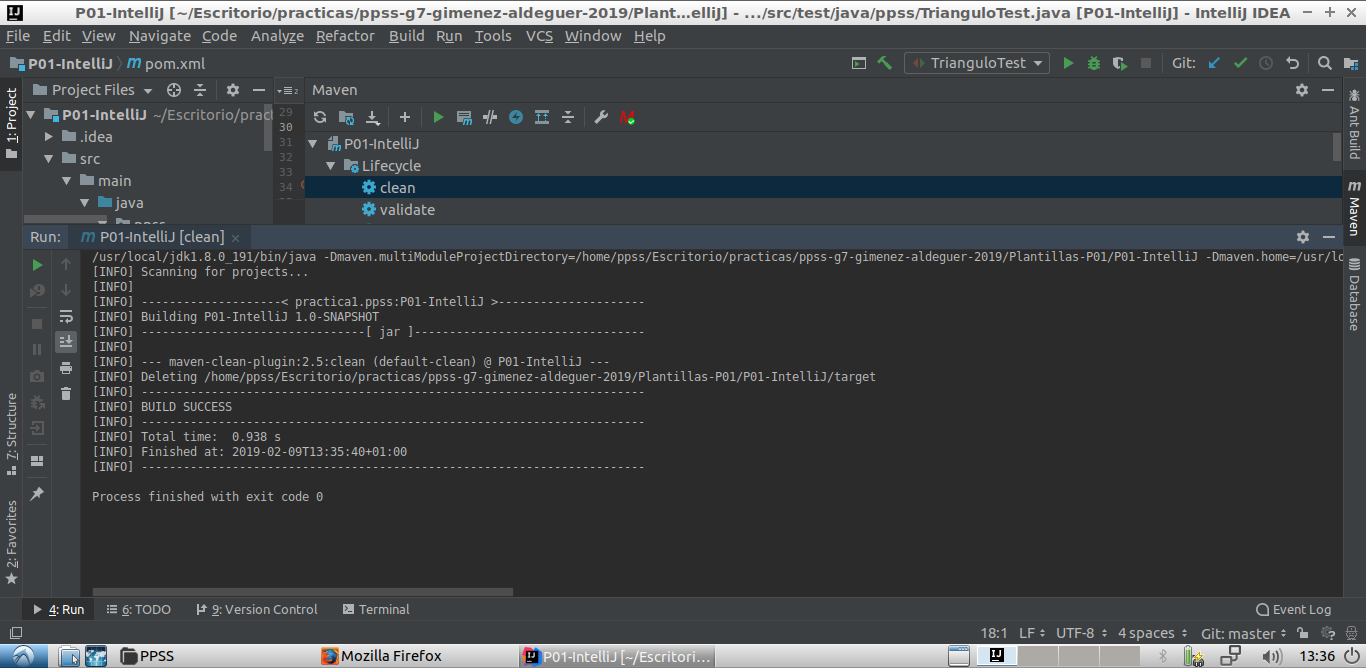
Darle valor a la variable resultado esperado.

Darle valor a la variable resultado real, utilizando el método del programa pasándole las variables necesarias.

Comparar si el resultado esperado es igual al resultado real, y devolver cierto o falso.

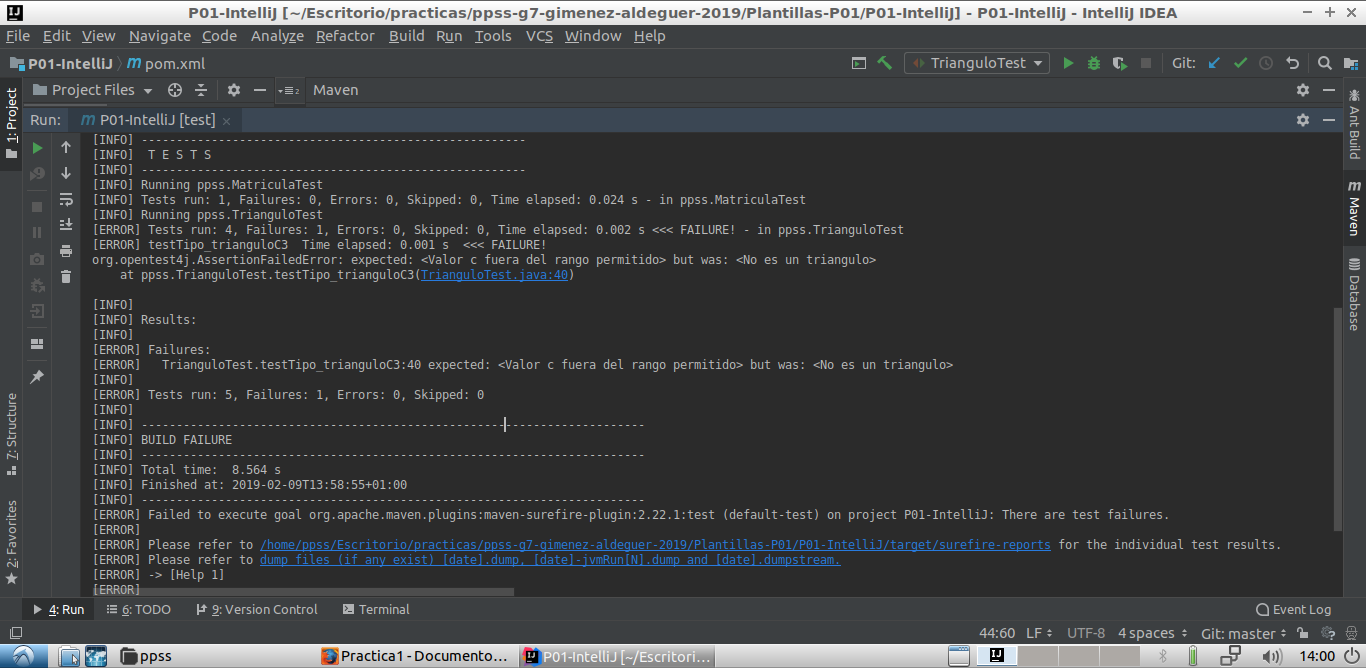
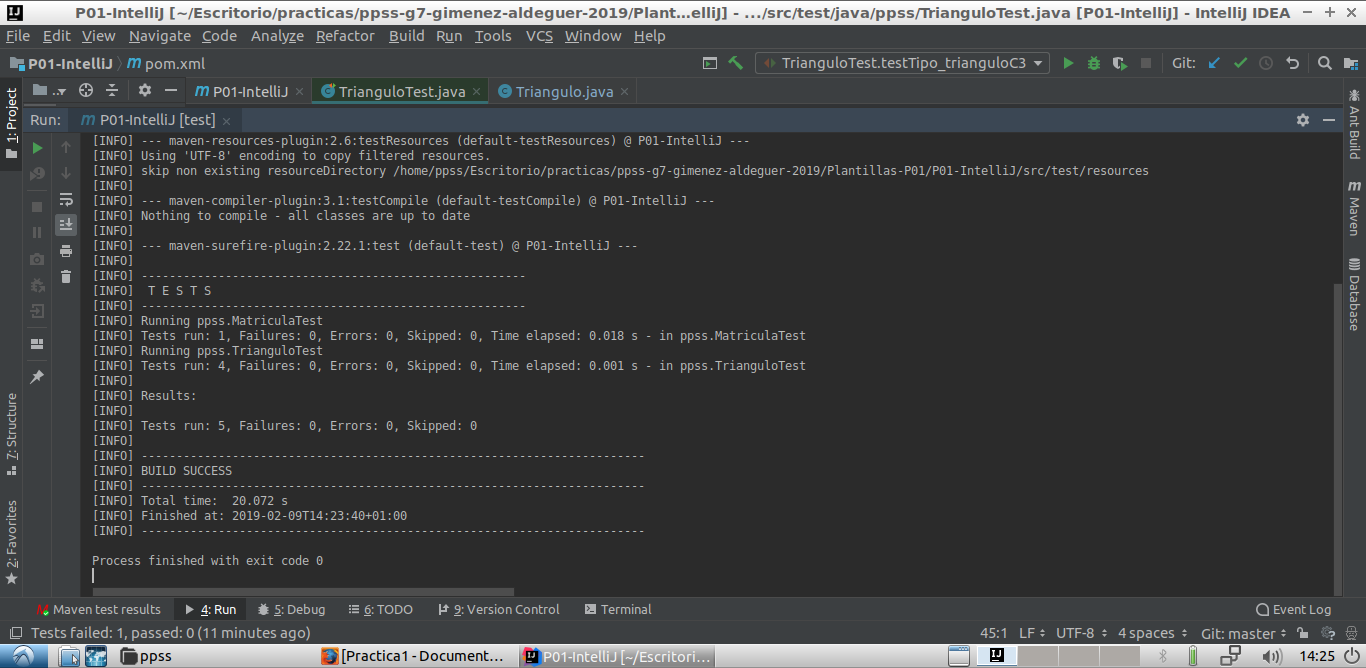
* 1. **Ejercicio 2:**
  2. **A) Observa que ha aparecido un directorio nuevo en nuestro proyecto maven (en el panel de la izquierda). Anota la nueva estructura de directorios creada, y qué artefactos contienen.**
  3. ****
  4. Esta nueva estructura, así como la ubicación de los artefactos también es común para CUALQUIER proyecto maven.
  5. **Ahora vamos a ejecutar la fase “clean”. Observa lo que ocurre y anota las goals que se ejecutan. Ahora vuelve a compilar el proyecto. Fíjate en la secuencia de acciones que se muestran en la ventana inferior y en que NO se han ejecutado los tests.**

Al ejecutar clean elimina toda la carpeta target y no ejecuta ningún test.

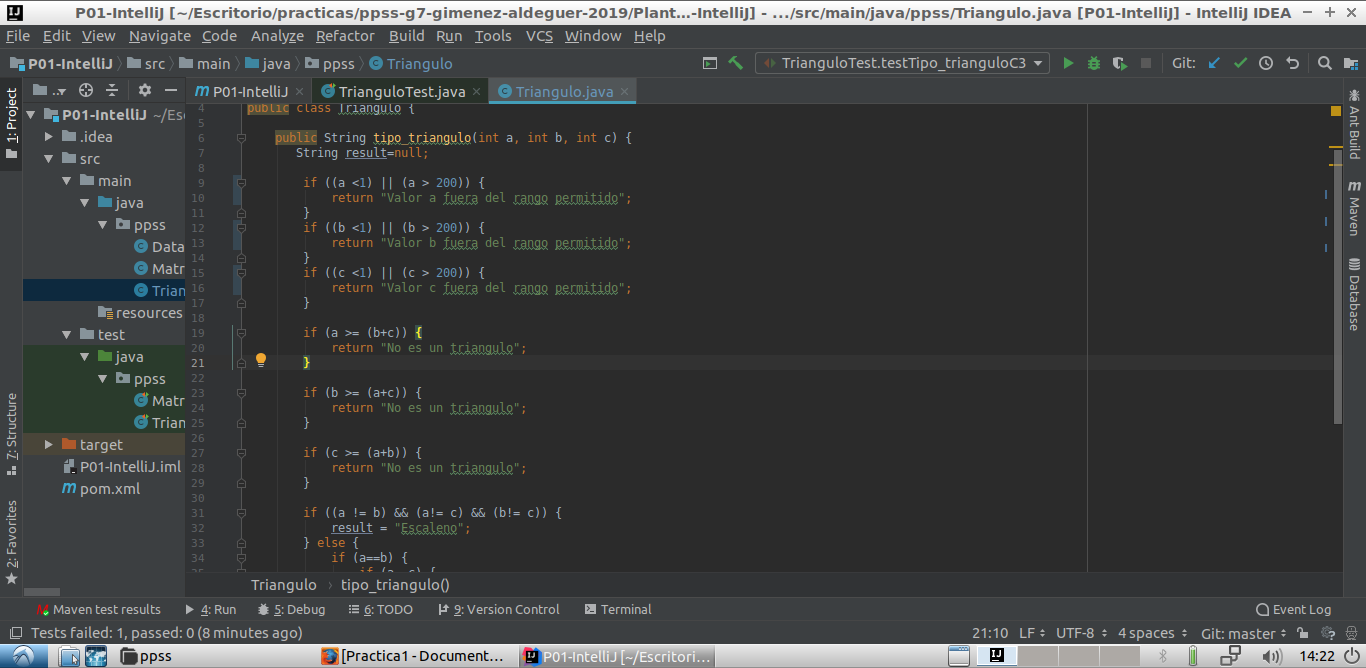


Las goals que se ejecutan son, clean:clean y clean:help.

Al compilar el proyecto de nuevo, NO se han ejecutado los tests.

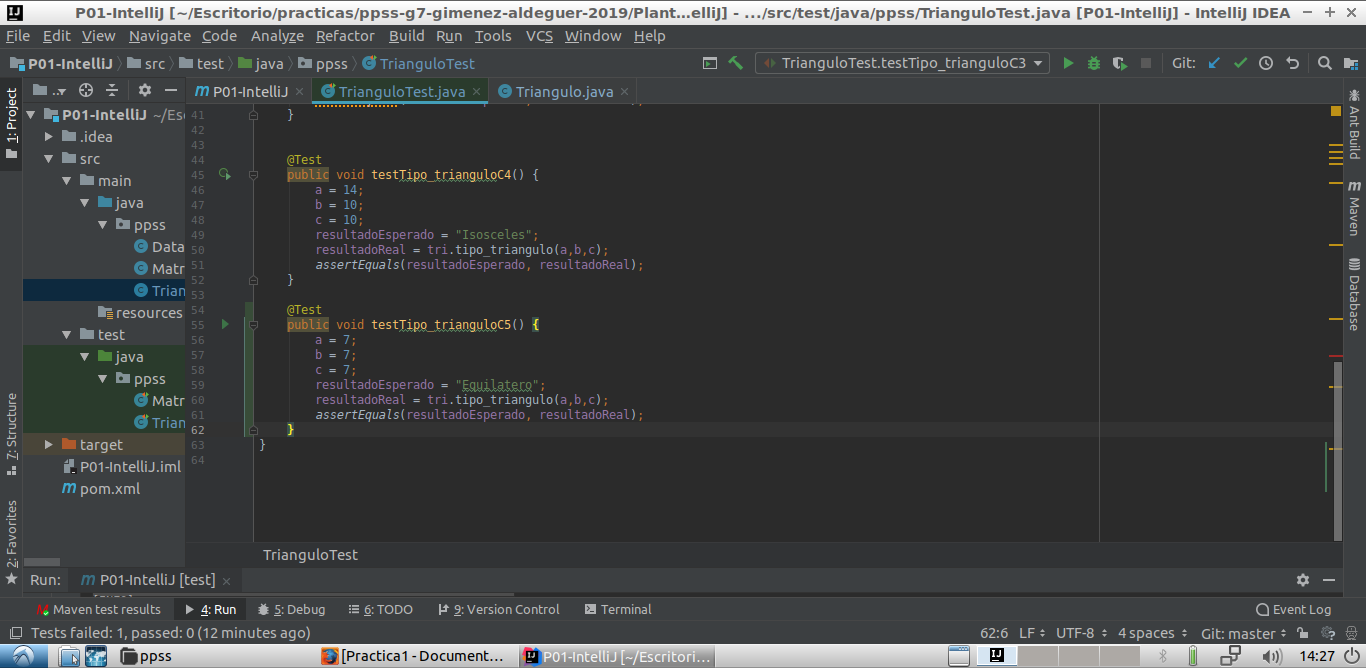
* 1. **B) Para ejecutar los tests vamos a hacer doble click sobre la fase “test”. Fíjate en la salida por pantalla, concretamente (y de momento) nos interesa la siguiente información: “Tests run” indica el número total de tests ejecutados. “Failures” indica el número de tests cuyo resultado esperado NO coincide con el real. Observa que junit proporciona un tercer tipo de resultado: “Error”, del que hablaremos en sesiones posteriores.**
  2. ****
  3. **C) Para poder concluir nuestro proceso de construcción con éxito: BUILD SUCCESS, necesitamos eliminar el problema/s que provoca el fallo de ejecución. A continuación vuelve a ejecutar la fase test (repite el proceso hasta que los cuatro tests estén en “verde”, y el proceso de construcción termine con: BUILD SUCCESS**
  4. 

Aquí están los cambios realizados:

****

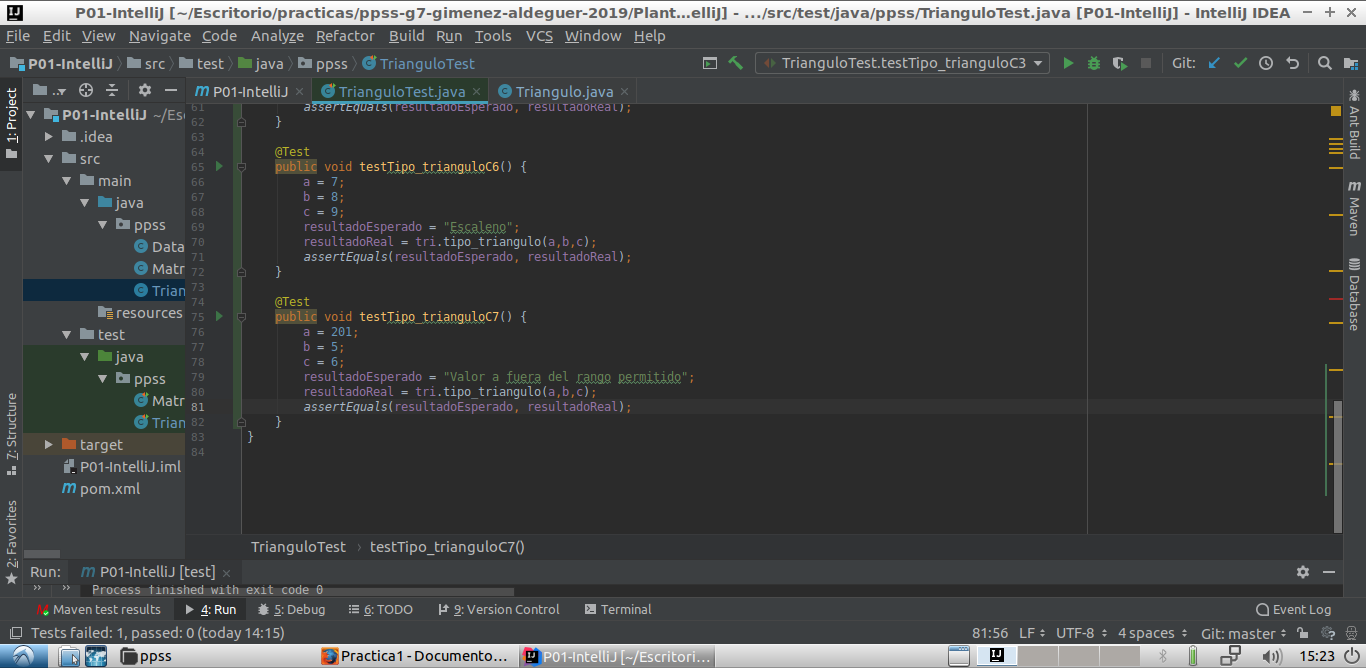
* 1. **D) Observa qué tienen en común el test C1 y un posible test adicional C5 con datos de entrada: a=7,b=7,c=7, y razona la conveniencia o no de incluir C5 al conjunto de tests. De la misma forma razona si son necesarios los tests C2 y C3. Basándote en tu razonamiento anterior, piensa en dos posibles casos de prueba adicionales que “aporten valor” al conjunto de casos de prueba (no sean innecesarios) justificando tu respuesta.**

El test C1 y el C5 son exactamente iguales, el resultado esperado es el mismo.



Los test C2 y C3 son necesarios ya que se trataría de dos caminos independientes.

El C6 tiene como resultado espera, si se trata de un Escaleno el triángulo y el C7 compruebo si “a” supera el rango permitido (200), la segunda condición del if sería otro camino independiente.



* 1. **Ejercicio 3:**

**La clase Matricula contiene el método calculaTasaMatricula() que devuelve el valor de las tasas de matriculación de un alumno en función de la edad, de si es familia numerosa, y si es o no repetidor, de acuerdo con la siguiente tabla (asumiendo que se aplican sobre un valor inicial de tasa=500 euros)**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | **Edad < 25** | **Edad < 25** | **Edad 25..50** | **Edad 51..64** | **Edad ≥ 65** |
| **Edad** | SI | SI | SI | SI | SI |
| **Familia Numerosa** | NO | SI | SI |  |  |
| **Repetidor** | SI |  |  |  |  |
| **Valor tasa-total** | tasa + 1500 | tasa/2 | tasa/2 | tasa - 100 | tasa/2 |

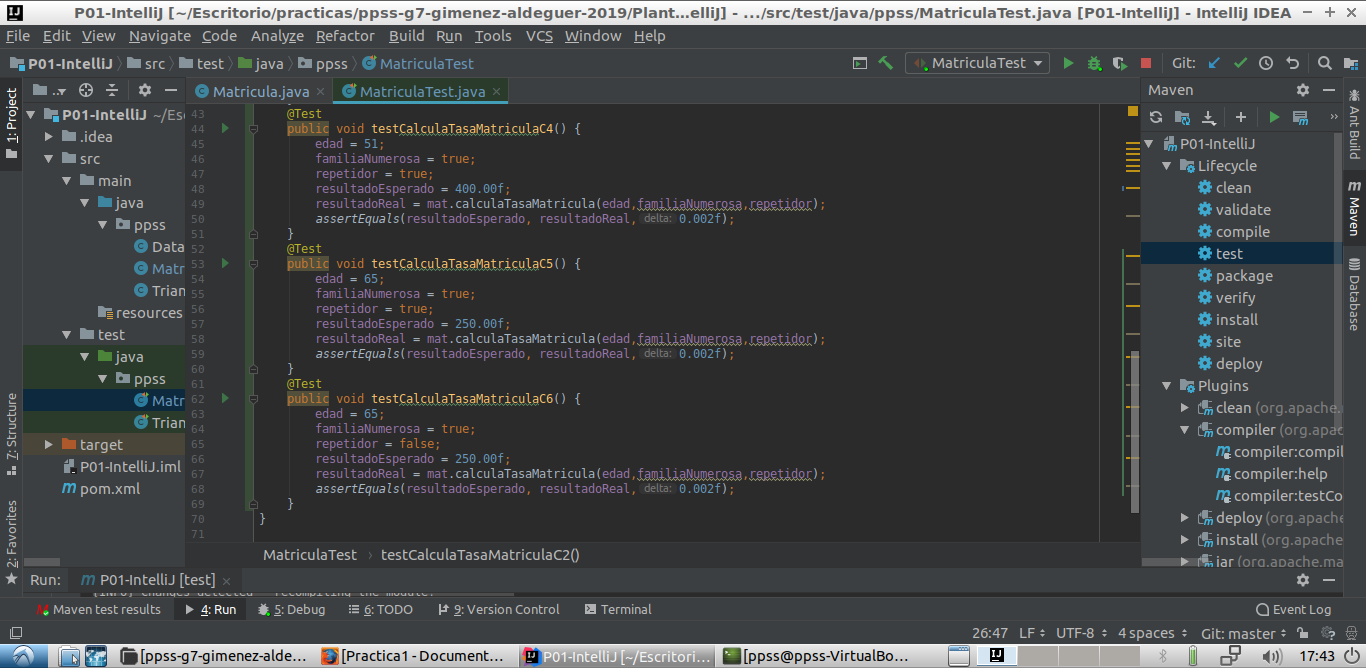
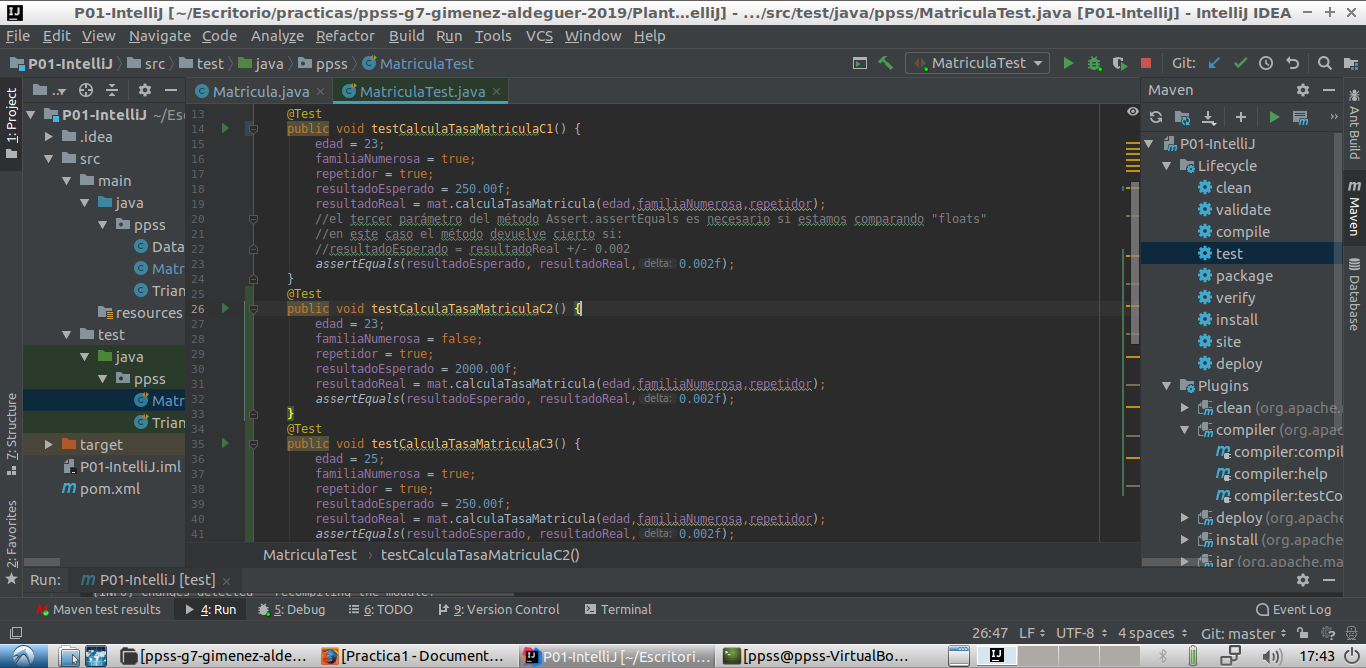
**A) En este caso, hemos proporcionado la implementación de un único test, en la clase MatriculaTest. Rellena una tabla de casos de prueba con el test que hay implementado y piensa 5 nuevos tests que no sean “redundantes” y añadelos a la tabla. En tu opinión, ¿son suficientes o deberíamos añadir alguno más?**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Identificador del Caso de prueba** | **Datos de entrada (todos)** | **Resultado esperado** | **Resultado real** | **Informe** |
| **C1** | ed=23 fam=true rep=true | 250 | 250 | OK |
| **C2** | ed=23 fam=false rep=true | 2000 | 2000 | OK |
| **C3** | ed=25 fam=true rep=true | 250 | 250 | OK |
| **C4** | ed=51 fam=true rep=true | 400 | 150 | FALLO |
| **C5** | ed=65 fam=true rep=true | 250 | 250 | OK |
| **C6** | ed=65 fam=true rep=false | 250 | 250 | OK |

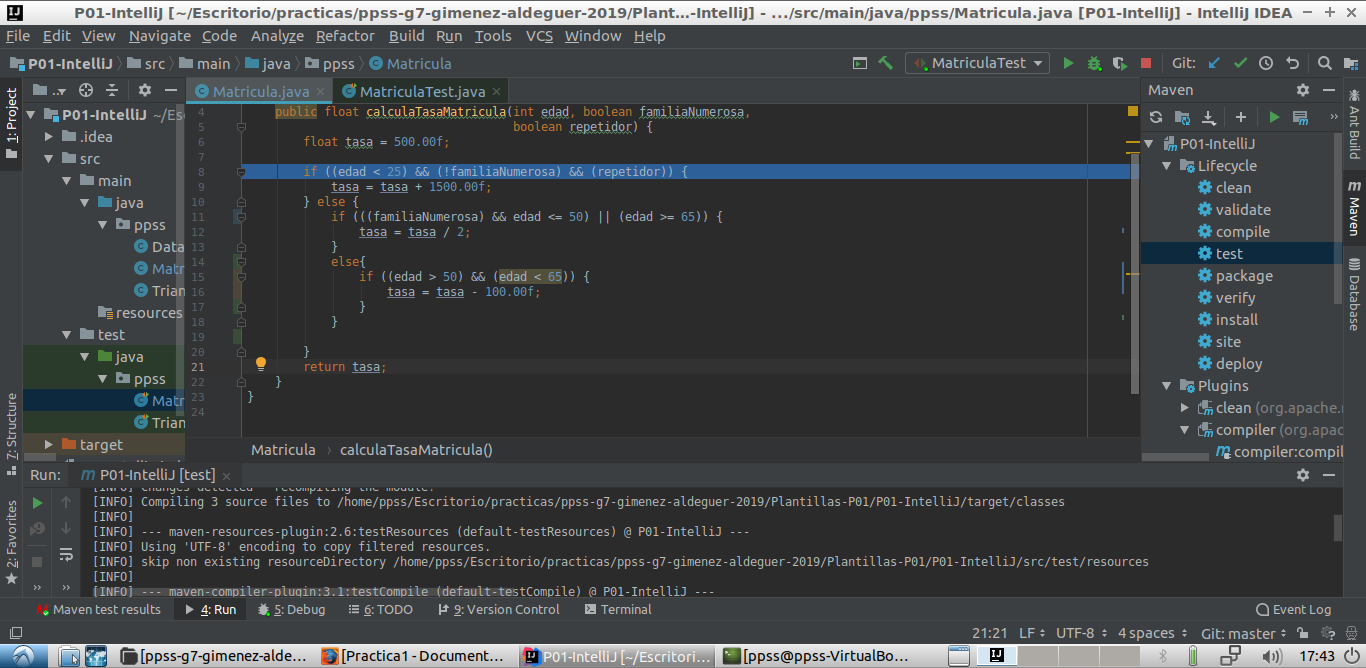
No son suficientes, ya que existen otros caminos independientes que no comprobamos en los test que hemos añadido.

**B) Implementa los casos de prueba que has añadido a la tabla y ejecuta todos los tests. Recuerda que si encuentras algún error debes depurarlo.**

He implementado los casos de prueba y he ejecutado los test, en la tabla del ejercicio 1 he mostrado cual me ha dado fallo.



Los cambios en el código que he tenido que hacer para que todos los test hayan funcionado sin ningun fallo son los siguientes:



**C) Pregunta a tus compañeros si han sido capaces de encontrar errores en el código ejecutando alguno de los tests implementados. Indica en qué te has basado para decidir los 5 nuevos casos de prueba que te hemos pedido, y contrasta con tus compañeros en qué se han basado ellos para tomar su decisión.**

Han sido capaces de encontrar errores en el código ejecutando alguno de los tests implementados. Se han basado en la tabla que nos dan en el ejercicio para tomar las decisiones necesarias.

* 1. **Ejercicio 4:**

**A) Ejecuta la fase “package” y observa los cambios en la ventana del proyecto. Anota qué acciones se han llevado a cabo para construir el proyecto y qué artefactos nuevos se han generado. Justifica el tipo de dichos nuevos artefactos.**

Se generan estos nuevos artefactos en la ventada de proyectos:

* pom.properties : fichero con las coordenadas del artefacto generado.
* ppss.archivoTest.txt : archivo con los resultados finales del test.
* TEST-ppss.archivoTest.xml : datos necesarios para la ejecución de los tests.
* archivoTest.class : el código de los tests.

Package lo que hace es compilar y pasar los test, generando artefactos nuevos que sean necesarios.

**Es importante que tengas claro qué artefactos se generan en cada fase, para poder utilizar de forma adecuada maven. Modifica uno de los tests, de forma que dé un resultado fallido, ejecuta la fase “clean”, y a continuación la fase “package” de nuevo, observa lo que ocurre y explica por qué ocurre esto.**

Compila el código, genera la carpeta target y realiza los tests con un fallo.

**De igual forma, ahora, en lugar de introducir un error en los tests, edita el fichero Matricula.java, quita un punto y coma para provocar un error de compilación y vuelve a ejecutar “package”. Explica lo que ha pasado y por qué.**

Compila el código con fallo, genera la carpeta target y realiza los test.

**B) Vuelve a reparar todos los errores introducidos y ejecuta la fase “install”. En este caso el resultado es menos “obvio” ya que en esta fase se “instala” (copia) el artefacto generado en la fase anterior en el repositorio local. El repositorio local de maven se encuentra en $HOME/.m2/repository. Indica la ruta exacta del artefacto generado y justifica el por qué de dicha ruta.**

En el repositorio local se almacenan todos los artefactos que maven ha utilizado para construir el proyecto, más todos aquellos artefactos que hayamos “instalado”, por ejemplo, utilizando la fase install. Si borras el directorio .m2 no importa, maven lo volverá a crear automáticamente durante la próxima ejecución del comando mvn. La primera vez que ejecutemos maven, si el repositorio local está vacío, maven se descarga todos aquellos ficheros que necesita de sus repositorios remotos. Esto significa que, a medida que vayas ejecutando maven y necesitando los artefactos (ficheros jar, war, ear, pom,...) para construir el proyecto, tu repositorio local irá creciendo. Si maven encuentra en el repositorio local el artefacto correspondiente, no será necesario proceder a su descarga, por lo que se reducirá el tiempo de construcción del proyecto.

**Puesto que para la construcción de nuestro proyecto hemos necesitado utilizar junit, dicha librería estará también en nuestro repositorio local. Búscala e indica exactamente la ruta en la que se encuentra.**

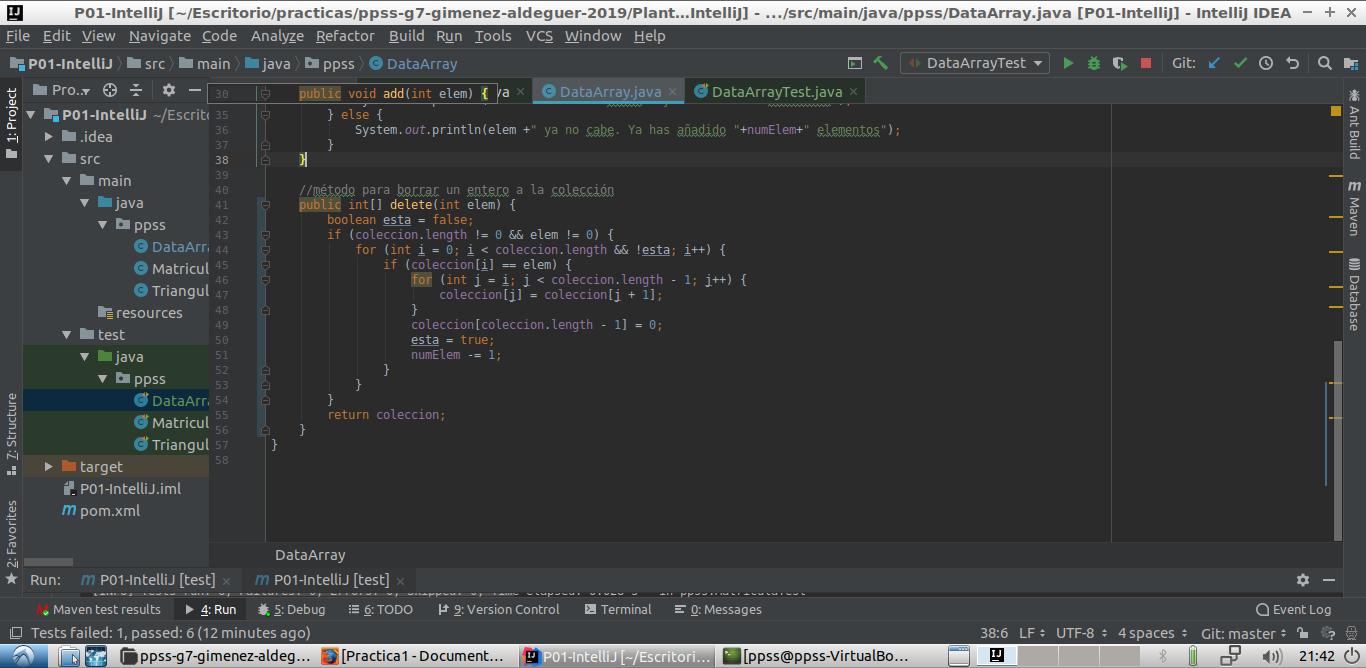
Se encuentra en “~/apache-jmeter-5.0/lib/junit”

* 1. **Ejercicio 5:**

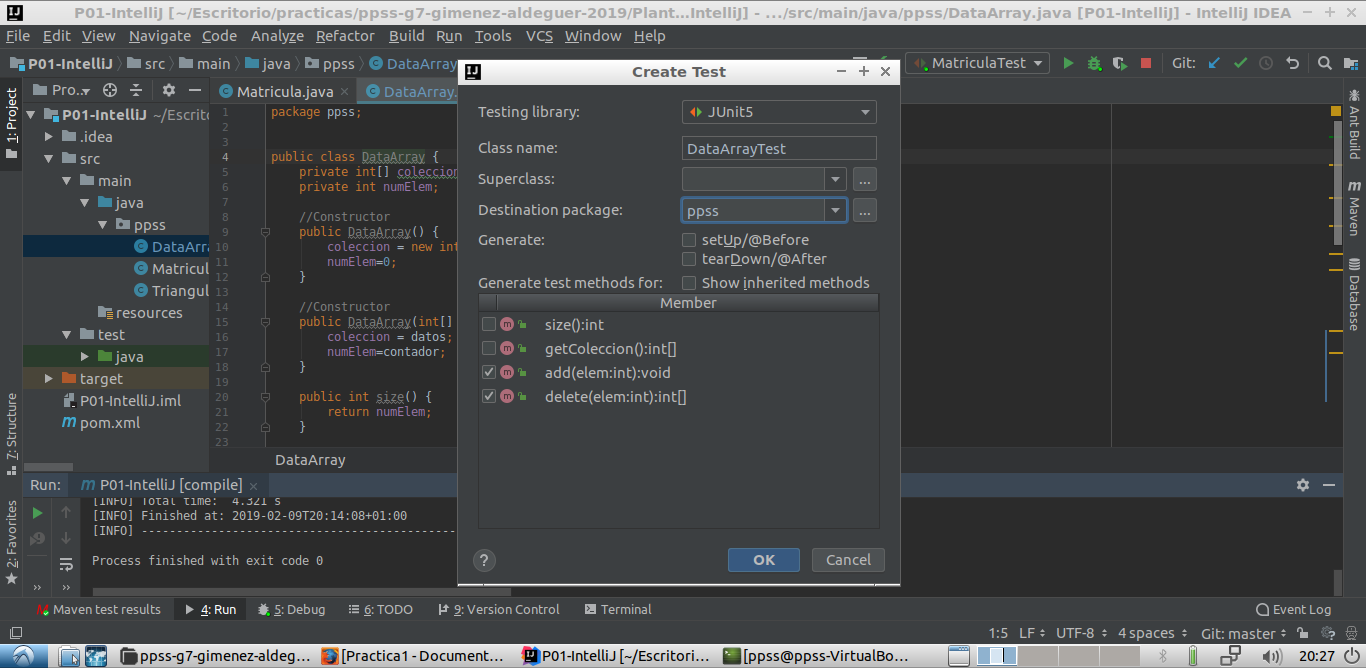
**A) Crea una tabla para el método delete() con los casos de prueba que pienses que deberías probar para asegurarte de que una futura implementación es correcta. Recuerda: una fila por caso de prueba. El identificador del caso de prueba tendrá que aparecer en el nombre del método correspondiente que implemente dicho caso de prueba.**

|  |  |  |  |
| --- | --- | --- | --- |
| **Identificador del Caso de prueba** | **Datos de entrada (todos)** | **Resultado esperado** | **Resultado real** |
| **C1** | 3 | Array con un 0 |  |
| **C2** | 1 | Array |  |
| **C3** | 0 | Array |  |
| **C4** | 3 | Array vacío |  |

**B) Implementa el método delete(), y asegúrate de que no tiene errores de compilación.**

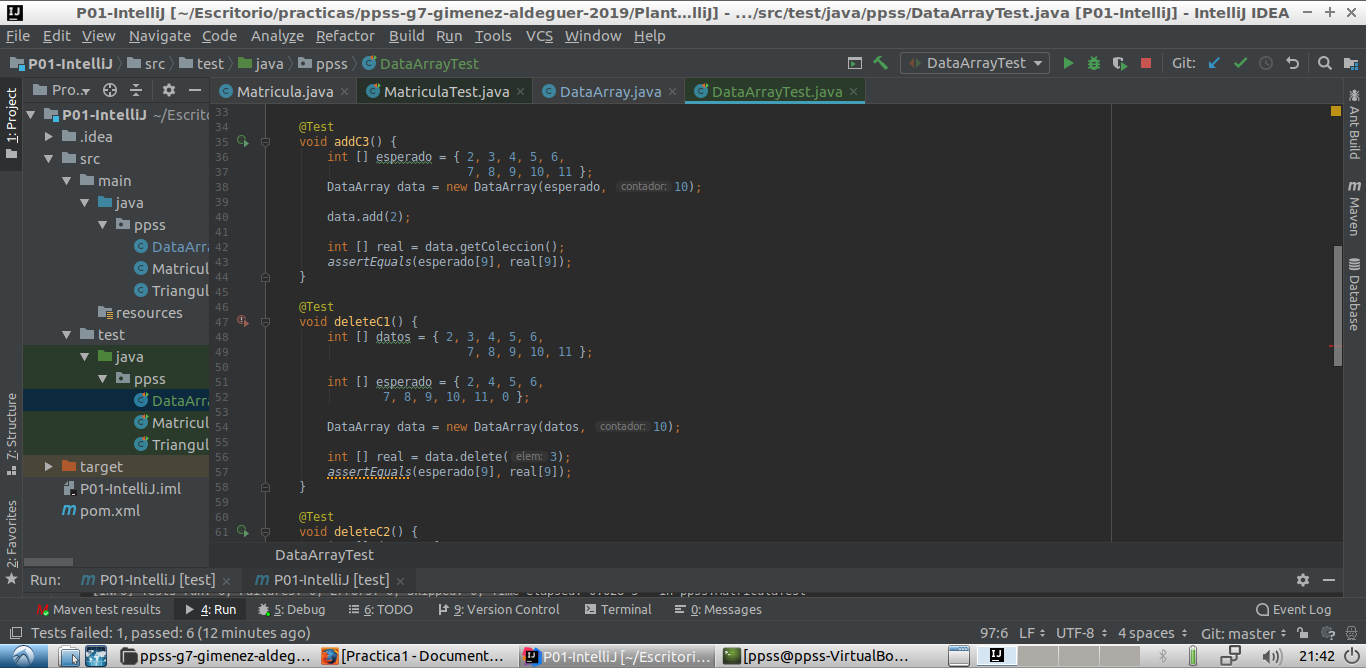
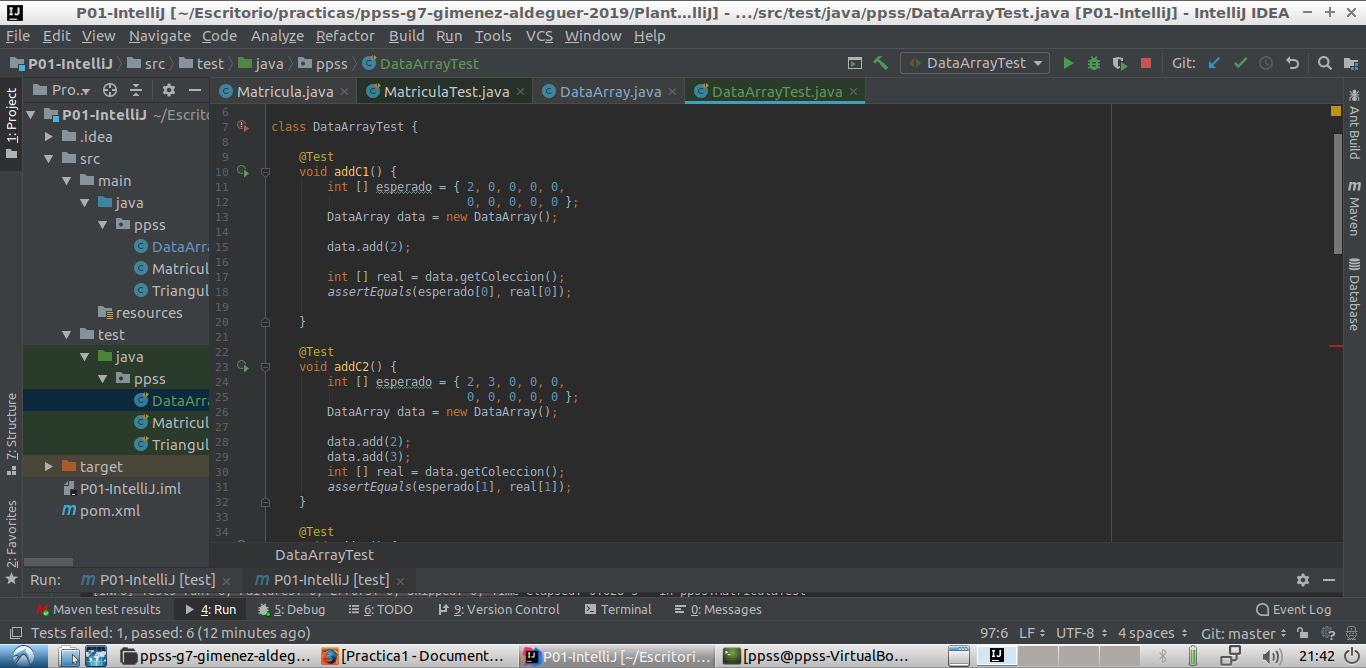
****

**C) Vamos a crear la clase que contendrá los tests para los métodos add() y delete(). Lo haremos a través del asistente de IntelliJ. Para ello seleccionamos en el código (DataArray.java) el nombre de la clase “DataArray” y a continuación pulsamos el botón derecho del ratón (para mostrar el menú contextual) y elegimos la opción “Generate... → Test...”. La librería a utilizar debe ser “Junit5”, marcaremos las casillas de los métodos add() y delete() y pulsamos sobre “OK”. Esto nos creará una nueva clase java denominada DataArrayTest en el directorio src/test/java (dentro del paquete ppss), con dos métodos anotados con @Test. Una vez creada la clase implementa tres tests para el método add(), considerando estas situaciones:**

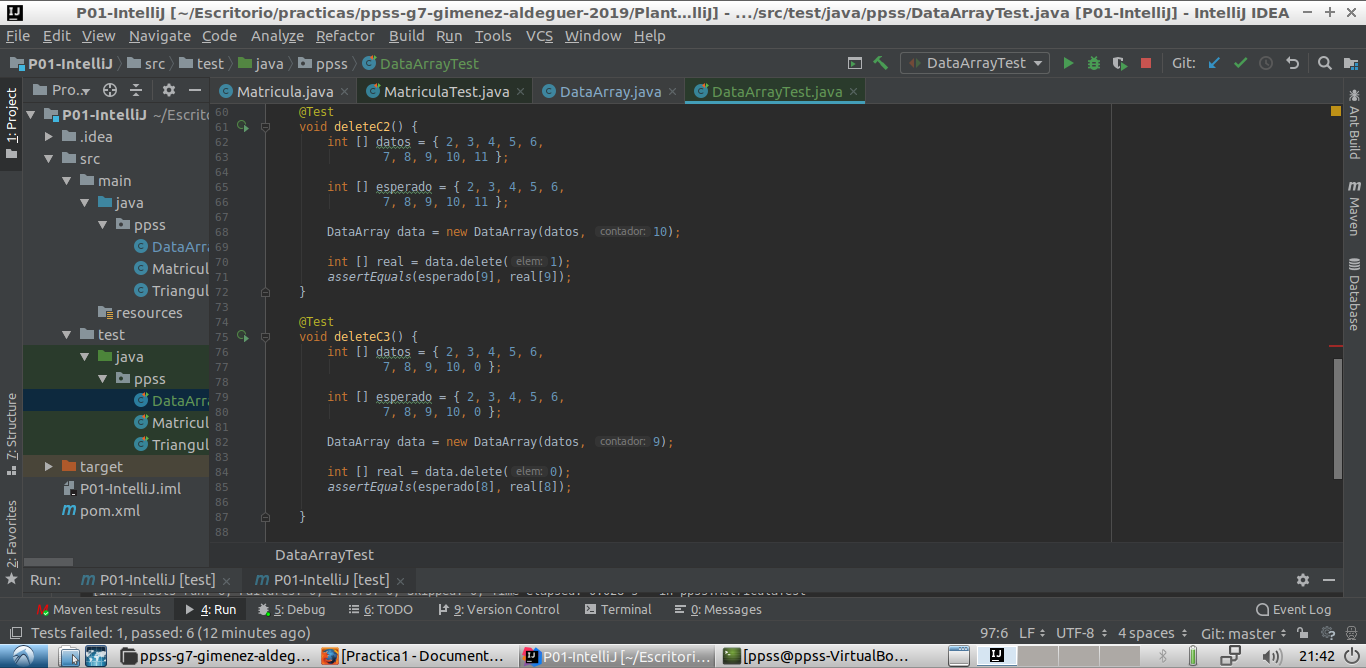
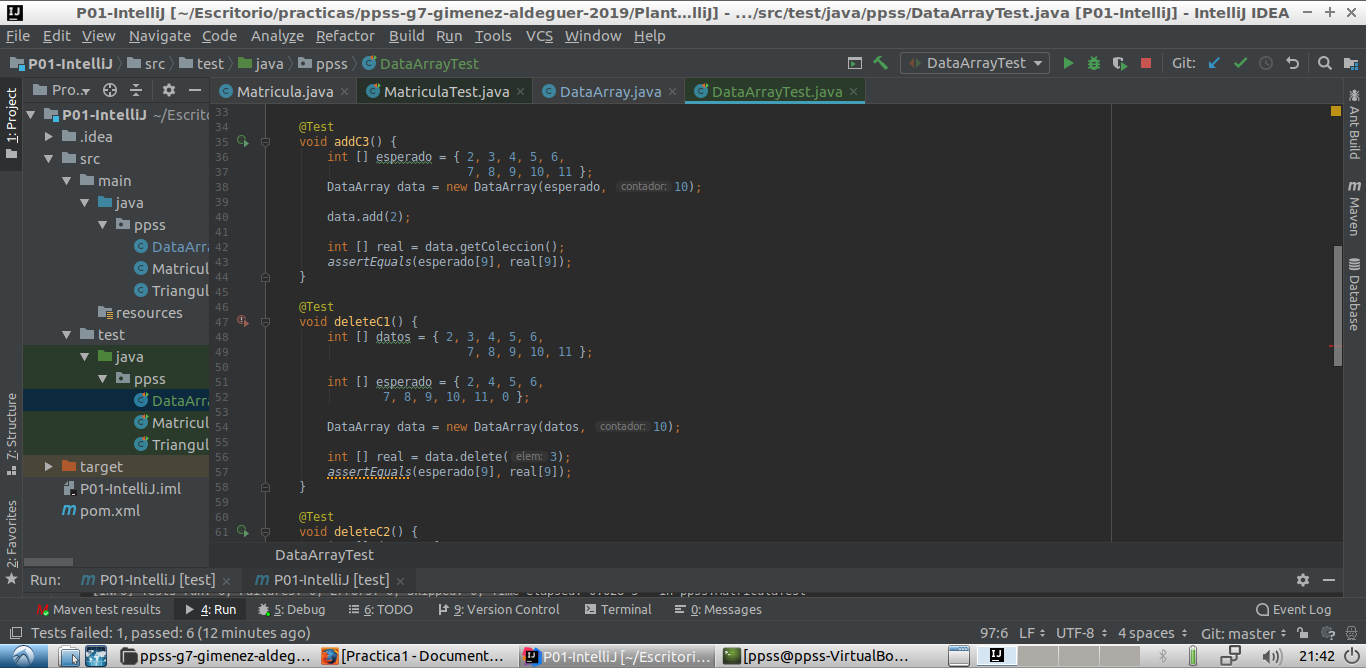
****

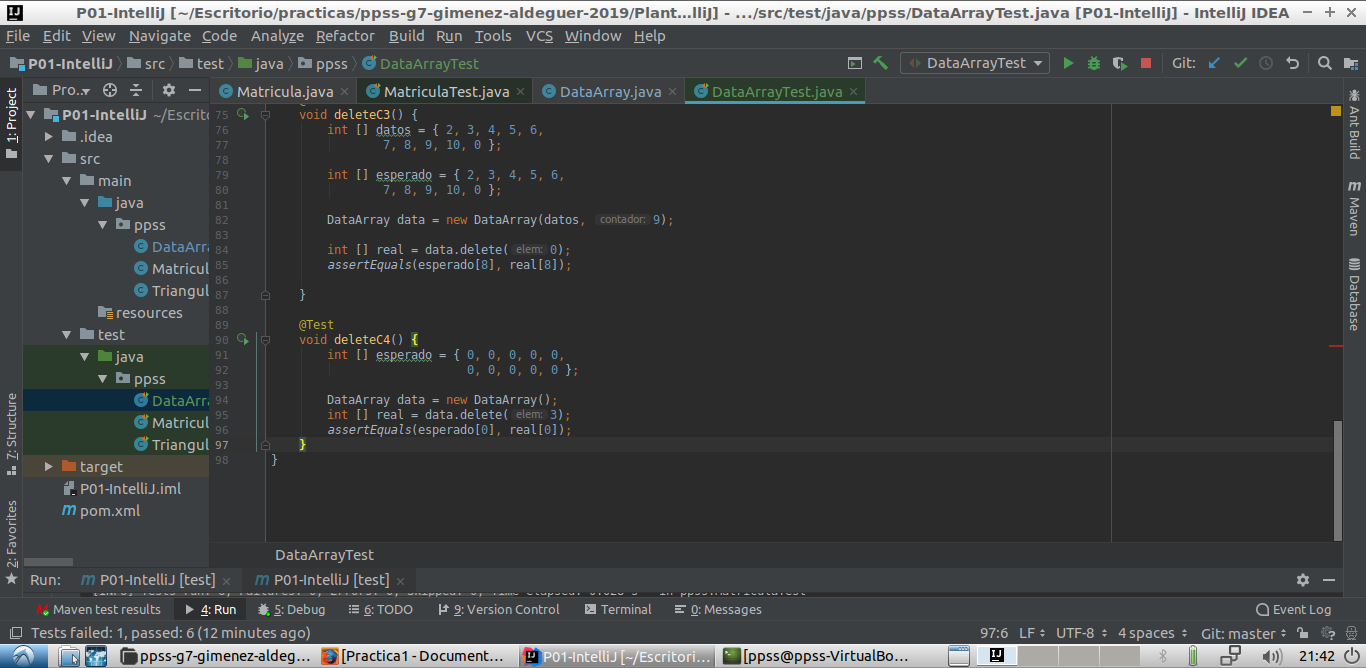
* **Añadir un elemento a una colección vacía**
* **Añadir un elemento a una colección de más de un elemento**
* **Añadir un elemento a una colección llena**

**Para implementar los tests, puedes generar de forma automática el prototipo de cada método seleccionando el nombre de la clase de puebas (“DataArrayTest”), y eligiendo desde su menú contextual, la opción “Generate...→ Test Method...”.**

****

**D) Implementa los tests de la tabla del apartado anterior para el método delete(). Depura los errores para que todos los tests estén en “verde”.**

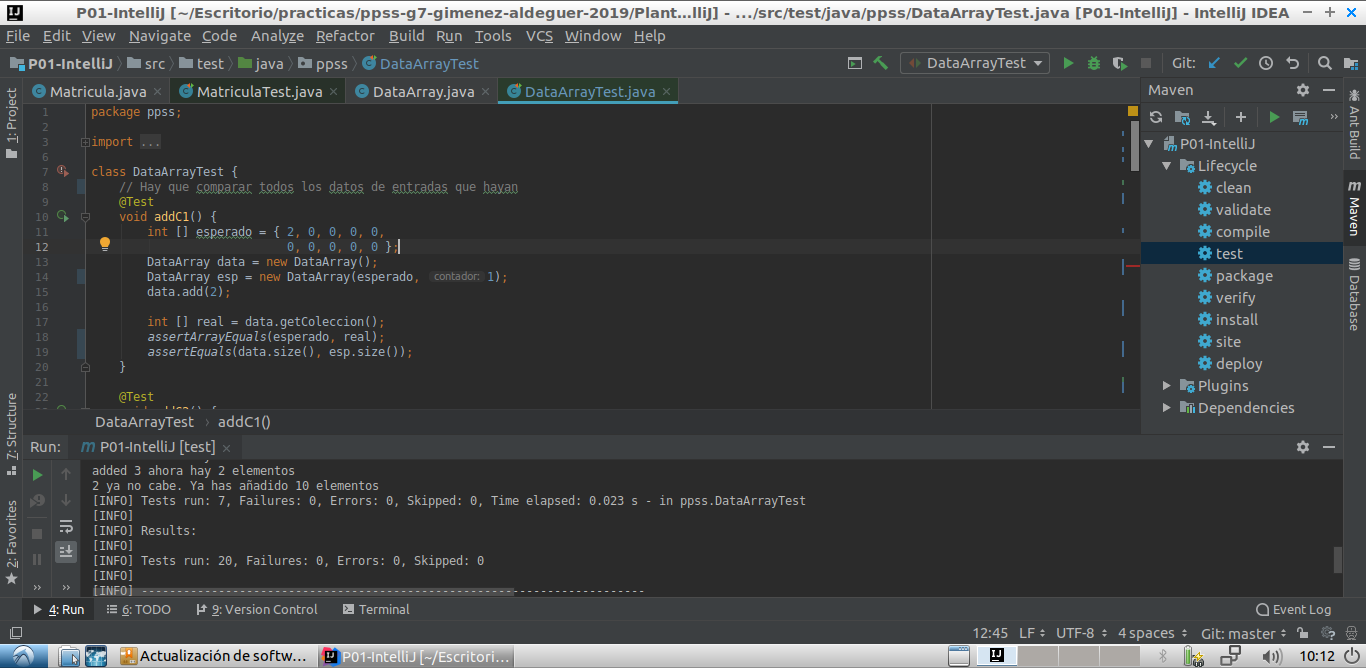
****

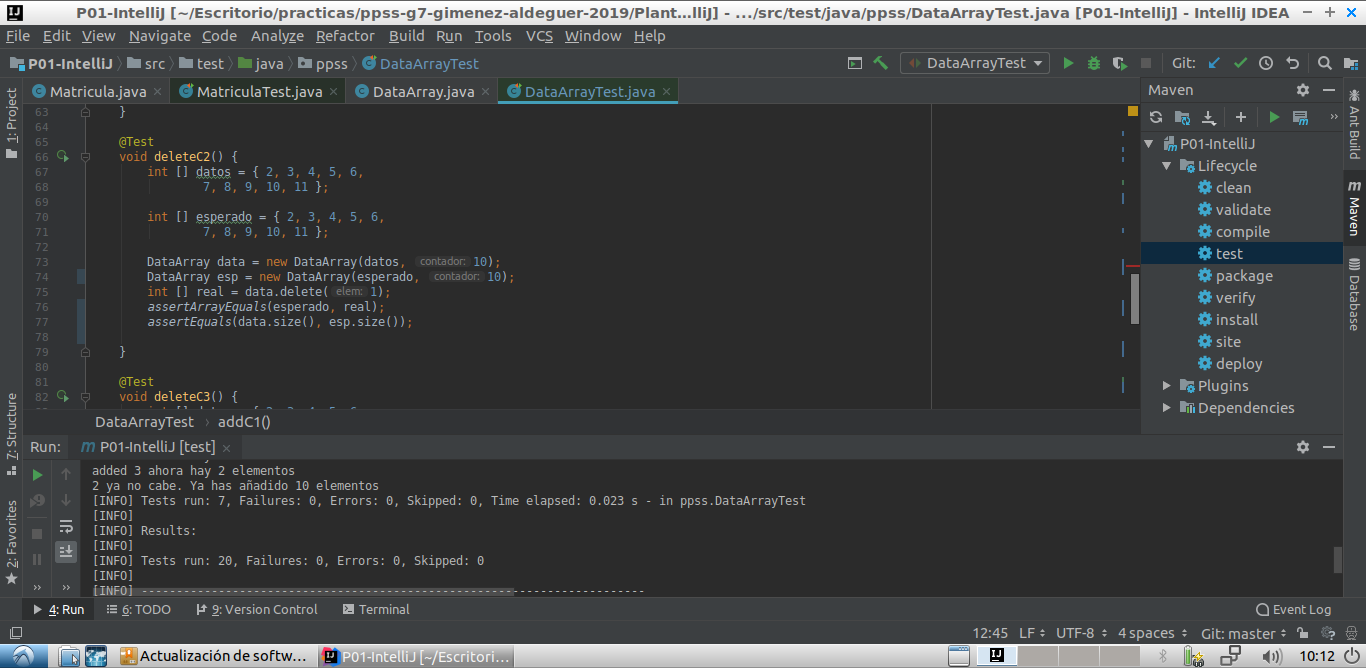
****

**Corrección:**

* La mayoría de veces, no habrá que poner cosas en el resultado real, ya que no vamos a disponer de los test a ejecutar.
* Si no tenemos la especificación de un camino independiente, no podemos poner el resultado esperado (?).
* En los test se tiene que comprobar todos los datos de entrada. Existe “assertArrayEquals();” para comparar arrays.
* Hay que especificar correctamente (cosa que yo no he hecho) cuales son los datos de entrada.

**Imágenes nuevas:**

****

****