## UD03 Diseño y realización de pruebas

Contesta a las siguientes preguntas sobre las pruebas de software.

1. ¿Qué es validar y qué es verificar software?

**Validar**: se trata de evaluar el sistema o de uno de sus componentes, para determinar si satisface los requisitos especificados. CAJA NEGRA

**Verificar**: es la comprobación de que un sistema o parte de un sistema, cumple con las condiciones impuestas. Con la verificación se comprueba si la aplicación se está construyendo correctamente. CAJA BLANCA

1. ¿Qué son las pruebas de caja blanca?

Se pretende verificar la estructura interna de cada componente de la aplicación, independientemente de la funcionalidad establecida para el mismo.

Su función es comprobar que se van a ejecutar todas la instrucciones del programa, que no hay código no usado, comprobar que los caminos lógicos del programa se van a recorrer, etc.

Se deben analizar las coberturas de sentencias, decisiones, condiciones, condiciones y decisiones, caminos y camino de prueba.

Se utilizan Grafos para determinar la complejidad ciclomática o de McCabe (nos interesa que sea lo menor posible) que determinará o estará determinada por los caminos recorridos. Se debe generar un listado de los casos de uso y listar el resultado obtenido.

1. ¿Qué son las pruebas de caja negra?

Tratan de comprobar si las salidas que devuelve la aplicación son las esperadas en función de los parámetros de entrada.

Para ello se pueden usar las siguientes metodologías:

* Particiones equivalentes (el menor posible): cada una alberga todos los casos que se comportan igual.
* AVL- Análisis de valores límite: se eligen como entradas los valores del límite de las clases de equivalencia
* Pruebas aleatorias: con generadores de pruebas al azar
* Conjetura de errores: por experiencia errores típicos

1. ¿Qué diferencia hay entre las pruebas de caja blanca y las pruebas de caja negra?

Las primeras (caja blanca) analizan la corrección y complejidad del código por dentro sin importar los resultados y las entradas.

Las segundas (caja negra) analizan los requisitos que se piden mediante el análisis de los datos de entrada y sus correspondientes de salida, sin importar el contenido del código.

1. ¿Cuáles son las recomendaciones para hacer las pruebas de caja blanca?

Se basan en la cobertura lógica, y para ello se analiza:

* Cobertura de las sentencias: todas deben ejecutarse al menos una vez
* Cobertura de decisiones: crear los suficientes casos de prueba para que cada opción resultada de una comprobación lógica del programa, se evalúe al menos una vez a cierto y otra a falso, atendiendo a && y ||.
* Cobertura de condiciones: crear los suficientes casos de prueba para que cada elemento de una condición se evalúe al menos una vez a falso y otra a verdadero atendiendo a los casos de && y ||
* Cobertura de condiciones y decisiones: las dos anteriores simultáneamente
* Cobertura del camino de prueba

1. ¿En qué consiste la prueba del camino básico?

Se basa en que en la mayoría de los casos es imposible en una aplicación probar todos los casos posibles, por lo que habrá que reducirlos.

Sí puede analizarse al menos caminos independientes para confeccionar casos de prueba que garanticen que se verifiquen dichos caminos de ejecución.

Consiste en elaborar un árbol Grafo que mediante nudos y aristas representen todos los caminos independientes del flujo del programa. Los caminos independientes son aquellos que introducen una nueva arista.

Con ello se pude definir la complejidad ciclomática y acotarla.

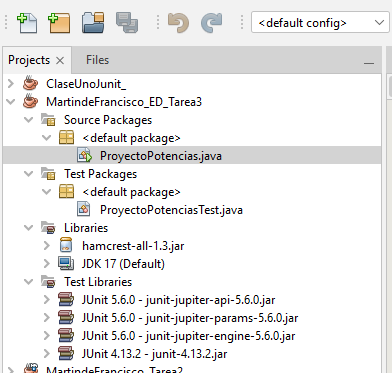
El resultado será los casos de prueba independientes (caminos), con sus resultados.

7.1. Diseña un caso de prueba que permita verificar el método PotenciaIterativa.

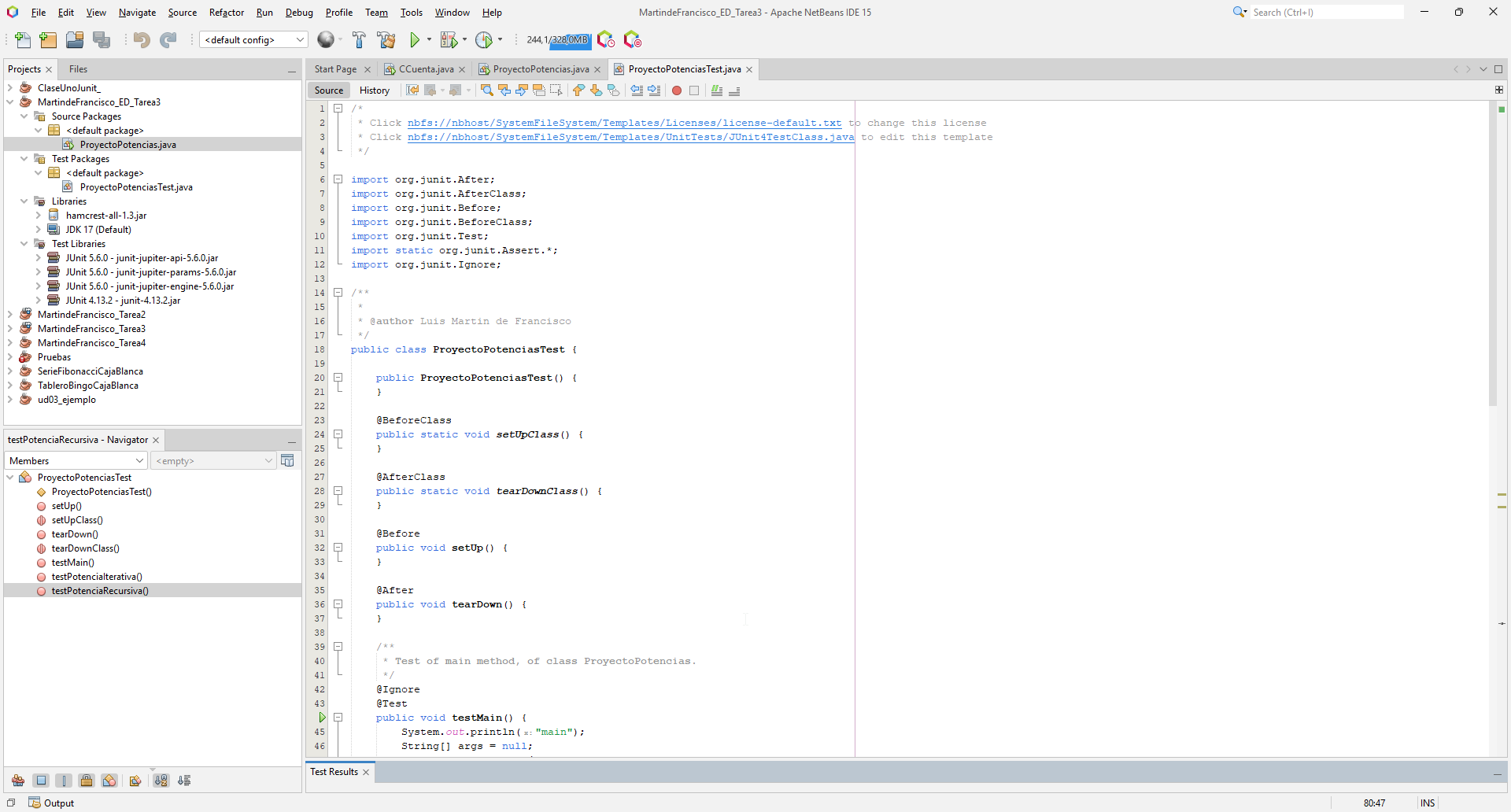
Comenzamos creando el archivo de prueba

Como vemos, Java Ant no admite hacer tests con Junit5 (no admite etiquetas como @Ignore), por lo que he tenido que incluir Junit 4.13.2.

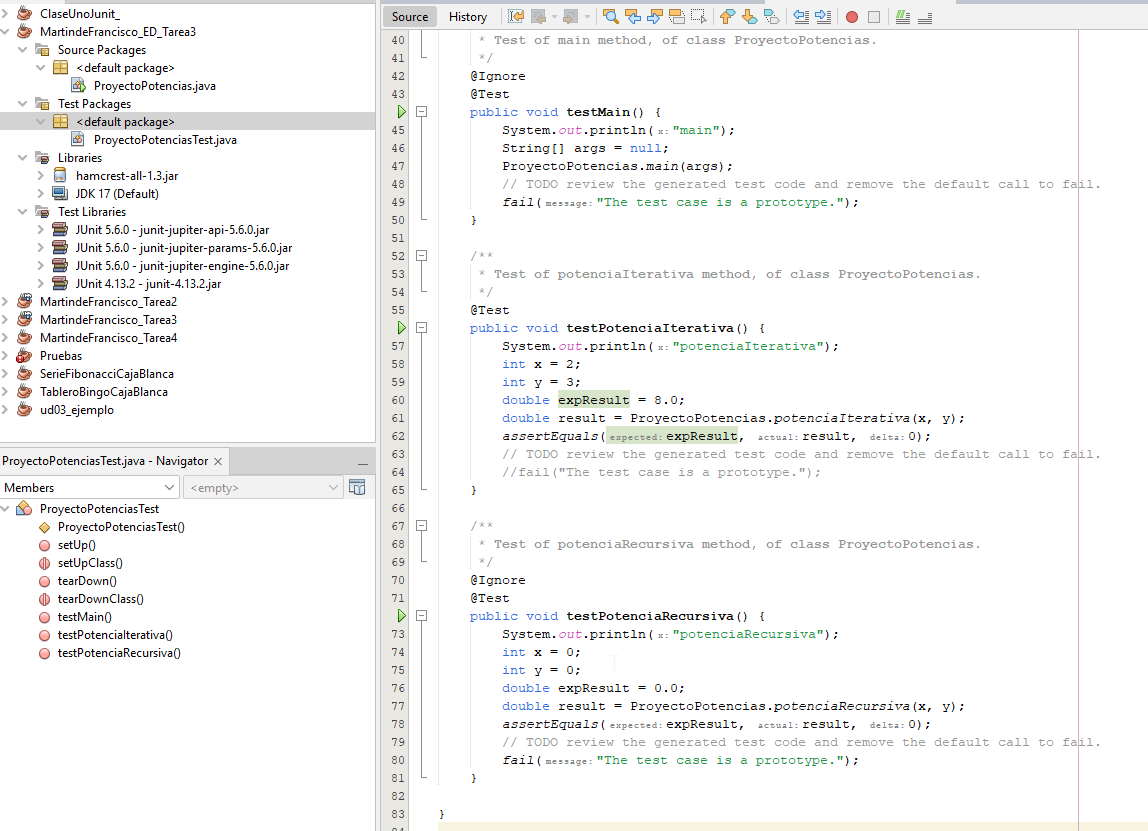
Además, con ayuda de una compañera he incluido la librería hamcrest-all-1.3.jar para poder ejecutar el test.



Una vez arreglado la configuración para poder ejecutar tests, creamos un test del tipo “Test for existing class” seleccionando el proyecto “ProyectoPotencias”.



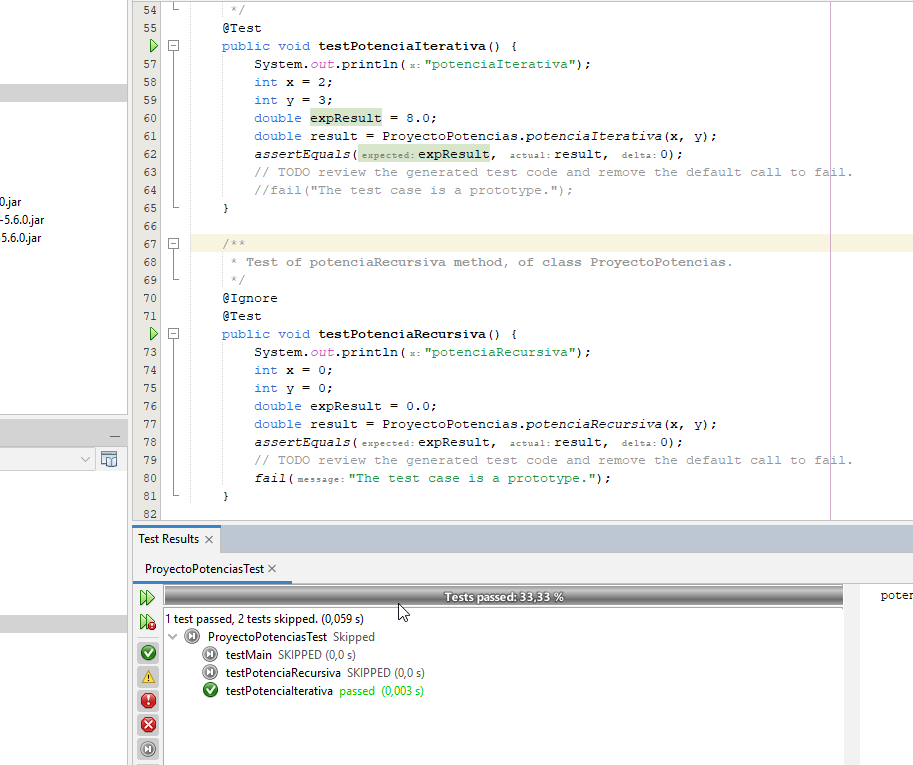
A partir de aquí, IGNORAMOS con @Ignore la clase main y la “testPotenciaRecursiva” para que sólo realice los tests en “testPotenciasIterativa”.



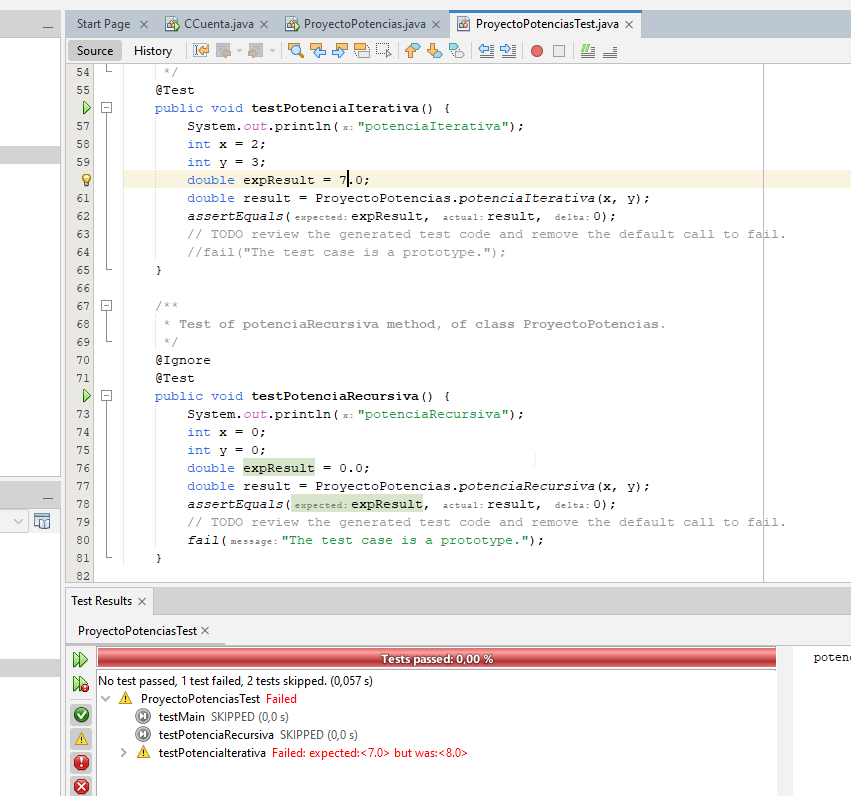
Ejecutamos un caso verdadero y uno que resulte en error.

Por ejemplo, utilizamos como base 2 exponente 3 y resultado 8 (7 como error).

CASO POSITIVO



CASO ERRONEO

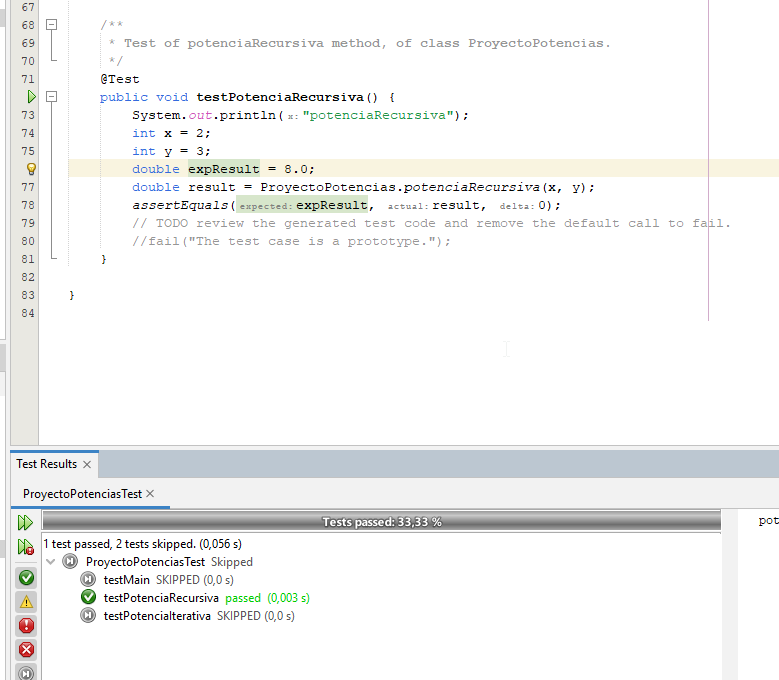


Vemos como nos ofrece el resultado y el error en el valor esperado.

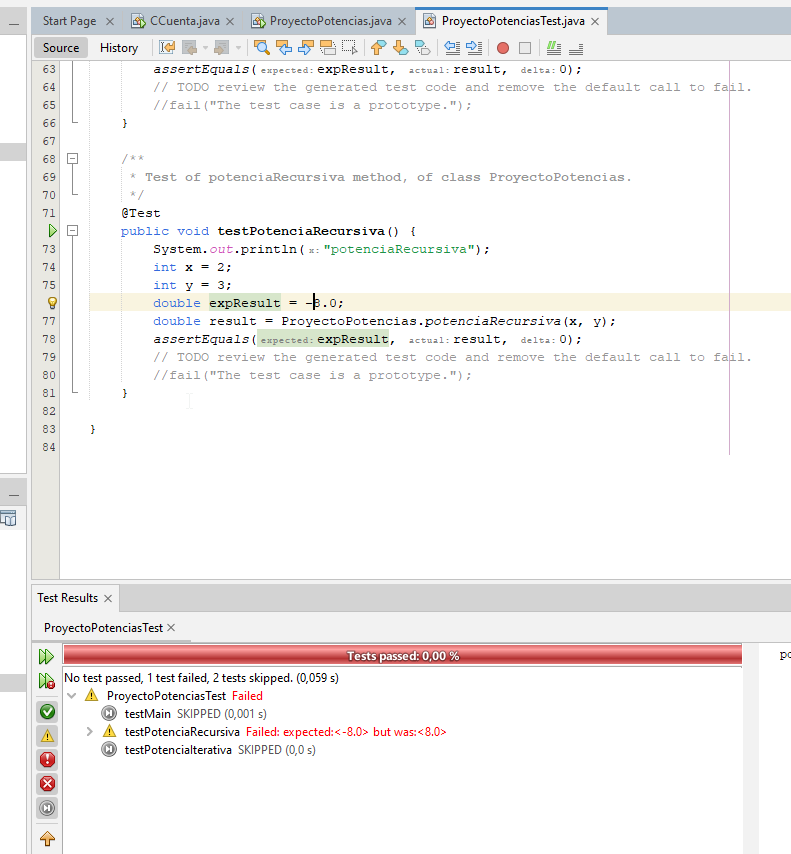
Haremos igual con la PotenciaRecursiva

7.2. Diseña un caso de prueba que permita verificar el método PotenciaRecursiva

CASO POSITIVO



CASO NEGATIVO



7.3. Indica el funcionamiento del método assert en la realización de pruebas. Justifica su utilización o no en este caso.

Las pruebas que se pueden desarrollar sobre JUnit se basan en los métodos assert, es decir, es la definición de una condición que debe cumplirse para considerar que nuestra clase probada funciona de la manera esperada.

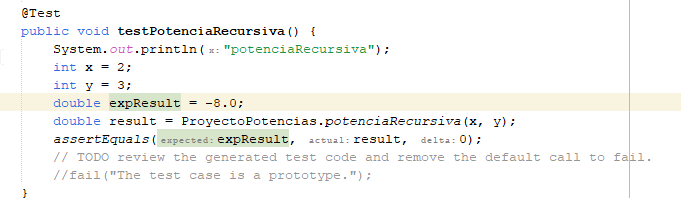
Dentro de los métodos podemos utilizar casos particulares como:

* assertEquals(resultado esperado, resultado actual): le pasamos el resultado que nosotros esperamos y invocamos la función que estamos testando.
* assertNull(objeto): si un objeto es null el test sera exitoso.
* assertNotNull(objeto): al contrario que el anterior.
* assertTrue(condición): si la condición pasada (puede ser una función que devuelva un booleano) es verdadera el test sera exitoso.
* assertFalse(condición): si la condición pasada (puede ser una función que devuelva un booleano) es falsa el test sera exitoso.
* assertSame(Objeto1, objeto2): compara las referencias de los objetos.
* assertNotSame(Objeto1, objeto2): al contrario que el anterior.

En nuestro caso la utilización del método assertEquals nos compara al utilizar la clase que estemos probando el resultado ejecutado y el esperado. Es por ello que introducimos valores a la base y exponente así como al valor esperado. El test ejecuta el método con los datos introducido, hace el cálculo y nos indica si lo que sale es lo mismo que lo que esperábamos.

Por lo tanto ASSERT tiene mucho sentido a la hora de diseñar los test unitarios.

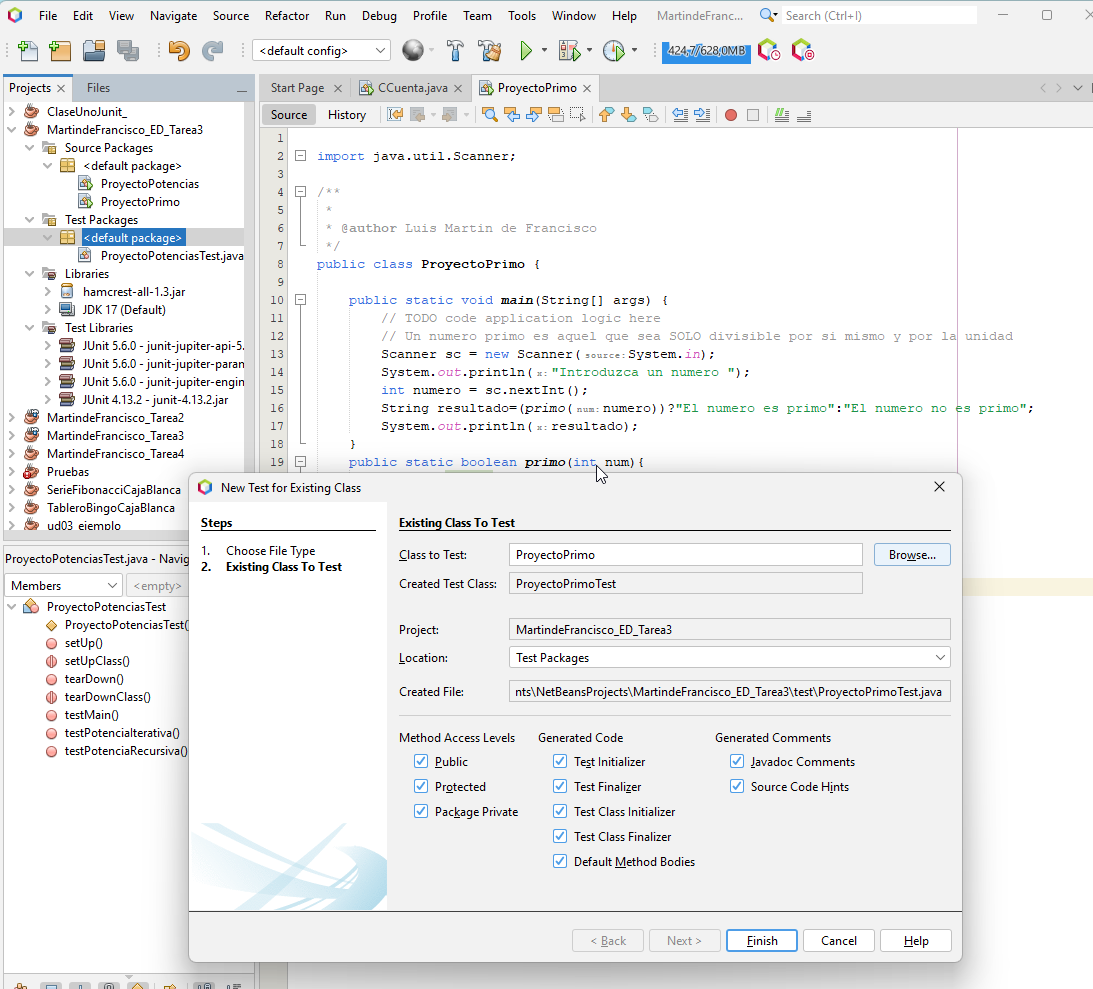
Podemos ver como se hace llamad del método assertEquals junto con dos argumentos, el cálculo realizado, y el esperado. El 0 (tercer parámetro) es la sensibilidad.



7.4. Haz que uno de los tests falle. ¿Cómo sería?

En los puntos 1 y 2 queda reflejado

7.4. Crea una prueba unitaria para el proyecto Primo

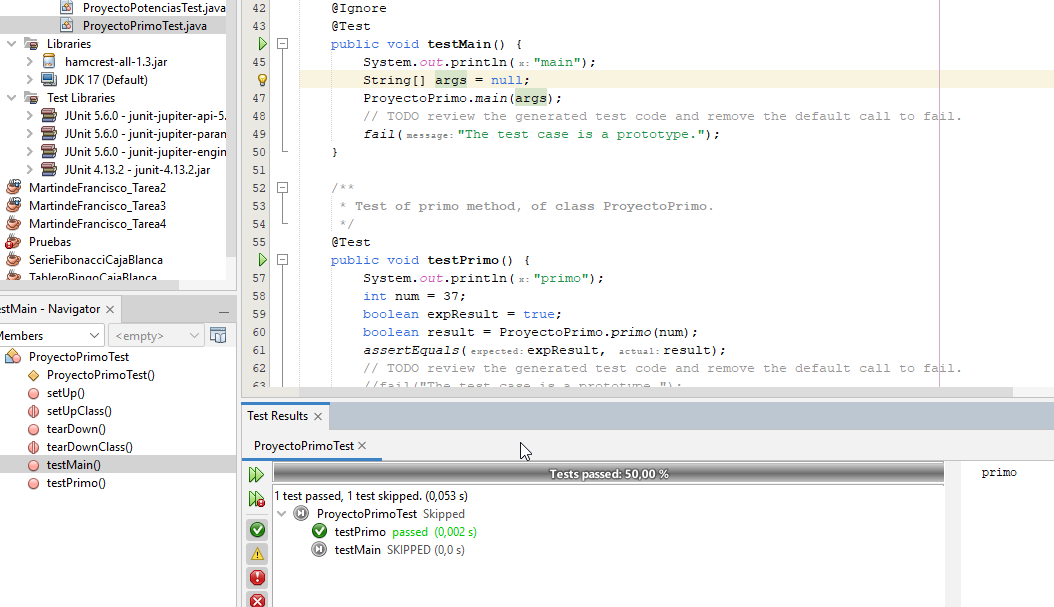
Creamos el Test

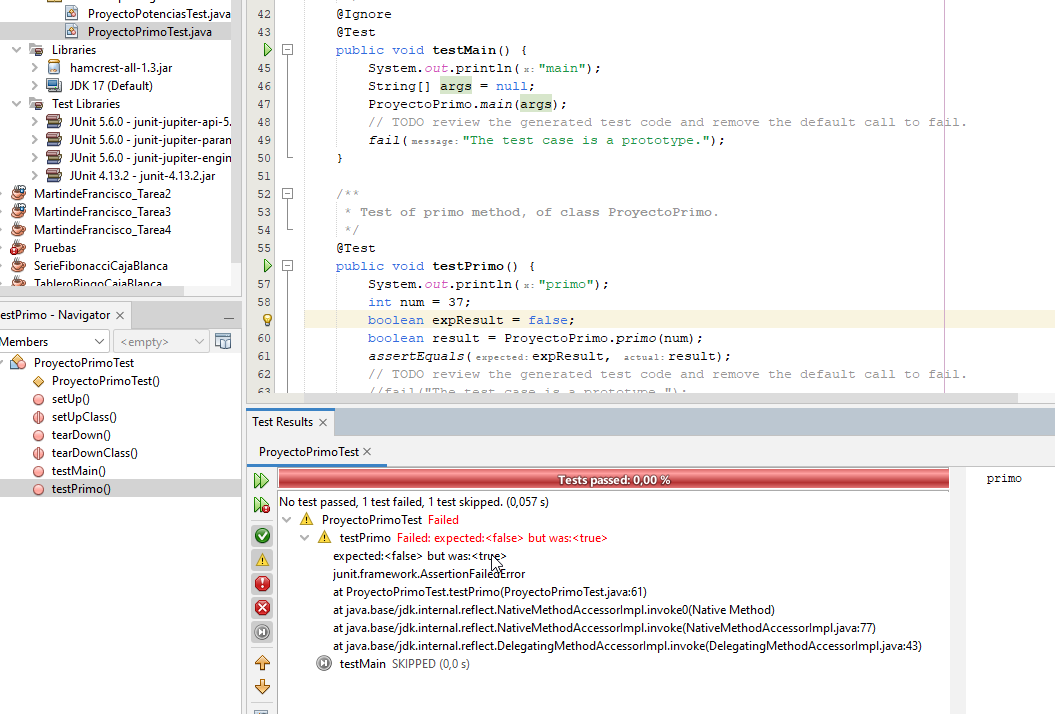
Interfaz de usuario gráfica, Texto, Aplicación, Word

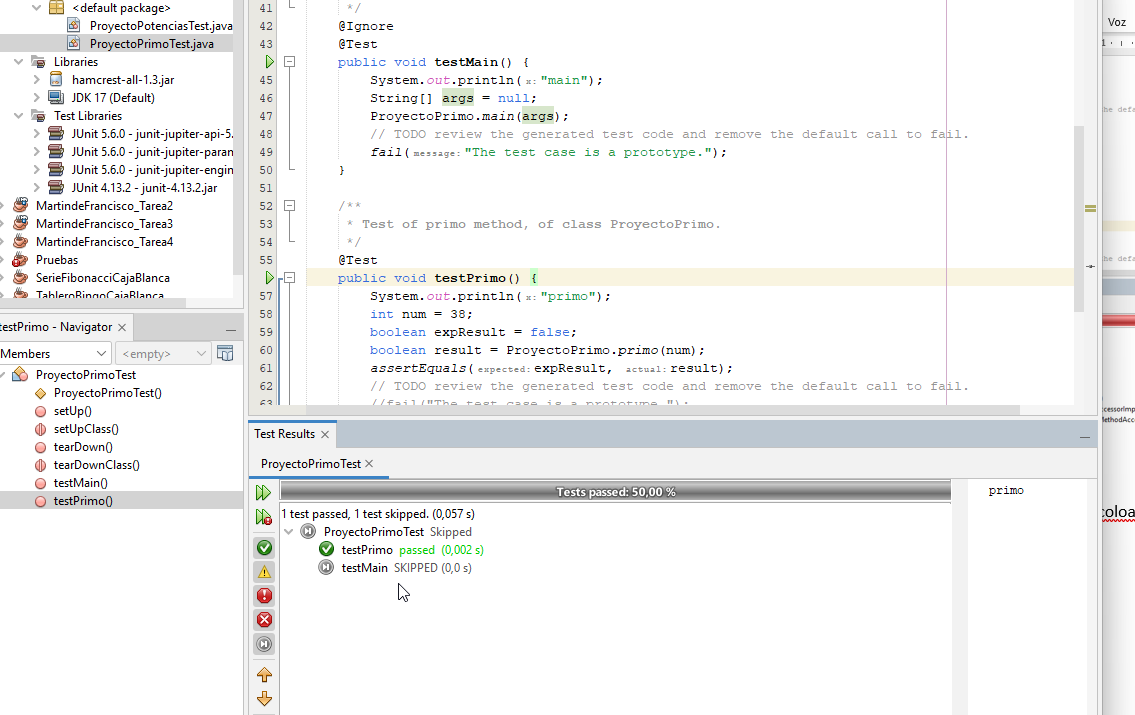
Descripción generada automáticamente

Ahora probaremos los valores 37 y 38 tal como se realiza en el video, pero con Junit.

Con 37 si colocamos exResult=true debe salir el test correcto, si coloamos exResult=false debe salir erroneo.





Con 38 si colocamos exResult=false debe salir el test correcto, si coloamos exResult=true debe salir erroneo.

