**Tarea 3 Entornos de desarrollo.**

1. **¿Qué es validar y qué es verificar software?**
   - Validar: Se refiere a asegurar que un producto de software cumple con los requisitos y expectativas establecidos por el cliente o usuario final. La validación se enfoca en evaluar si el software es adecuado para el propósito previsto y si satisface las necesidades del cliente. Esto implica realizar pruebas y revisiones para verificar la funcionalidad, usabilidad, rendimiento y otros atributos del software. La validación se lleva a cabo después de desarrollar y construir el software.
   - Verificar: Se refiere a confirmar que el software cumple con las especificaciones y requerimientos establecidos durante el proceso de desarrollo. La verificación se enfoca en examinar y evaluar el software en busca de errores, anomalías o desviaciones con respecto a las especificaciones. Esto implica realizar pruebas de unidad, pruebas de integración, pruebas de sistema y otras técnicas de prueba para detectar y corregir problemas en el código y asegurarse de que el software se está desarrollando de acuerdo con las especificaciones.

2**. ¿Qué son las pruebas de caja blanca?**

(White Box Testing): Se prueba la aplicación desde dentro, usando su lógica de aplicación.

3. **¿Qué son las pruebas de caja negra?**

(Black Box Testing): cuando una aplicación es probada usando su interfaz externa, sin preocuparnos de la implementación de la misma. Aquí lo fundamental es comprobar que los resultados de la ejecución de la aplicación, son los esperados, en función de las entradas que recibe.

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

La principal diferencia radica en el nivel de conocimiento de la estructura interna del software utilizado para diseñar y ejecutar las pruebas. Las pruebas de caja blanca se centran en la estructura interna y requieren acceso al código fuente, mientras que las pruebas de caja negra se centran en la funcionalidad externa y no requieren conocimiento detallado de la estructura interna del software.

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

1. Identificar los caminos de ejecución: Analizar el código fuente e identificar todos los caminos de ejecución posibles. Esto se logra a través de la revisión del código y de técnicas como el análisis estático del programa.

2. Diseñar casos de prueba para cada camino: Una vez identificados los caminos de ejecución, diseña casos de prueba específicos que cubran cada uno de ellos. Cada caso de prueba debe ser diseñado para alcanzar una condición única dentro del código.

3. Cobertura del código: Asegurarse de que tus casos de prueba cubran la mayor cantidad de líneas de código posible. Utilizar herramientas de cobertura de código para medir qué porcentaje de tus pruebas cubren el código fuente.

4. Pruebas de límites: Asegurarse de diseñar casos de prueba que prueben límites, tanto en el rango de entrada como en el rango de salida. Es posible que ciertos defectos se manifiesten solo en estos casos extremos.

5. Pruebas de integración y de unidad: Además de realizar pruebas de caja blanca, es recomendable complementarlas con pruebas de integración y de unidad. Esto te permitirá validar el comportamiento del software desde diferentes perspectivas.

6. Revisar el código: No limitarse a ejecutar pruebas, también revisar el código en busca de posibles errores o vulnerabilidades. Esto ayudará a detectar problemas en la lógica del programa.

7. Automatizar las pruebas: Siempre que sea posible, automatiza tus pruebas de caja blanca. Esto te permitirá repetirlas fácilmente cada vez que realicen cambios en el código o en la estructura del software.

8. Documentar y seguir las mejores prácticas: Asegurarse de documentar pruebas de caja blanca y seguir las mejores prácticas recomendadas para este tipo de pruebas. Esto incluye utilizar estándares de codificación, técnicas de depuración adecuadas, entre otros. Recordar que la selección y diseño de las pruebas de caja blanca dependen en gran medida de la comprensión y experiencia del tester en el dominio del proyecto.

**6. ¿En qué consiste la prueba del camino básico?**

La prueba del camino básico es una técnica de prueba de software que se utiliza para evaluar la cobertura de un conjunto de caminos en un programa de control de flujo. Consiste en identificar y ejecutar una combinación mínima de caminos que cubra todos los posibles escenarios y ramas del programa.

La prueba del camino básico se basa en la teoría de grafos y considera que cada camino independiente en el programa de control de flujo puede representarse como un camino en un grafo. Un camino básico se define como un conjunto de caminos independientes que cubre todos los nodos y aristas del grafo.

El objetivo de la prueba del camino básico es asegurar que se han considerado todos los posibles escenarios de ejecución del programa y que se han cubierto todas las ramas y condiciones de decisión. Para lograr esto, se utilizan técnicas de generación automática de casos de prueba que permiten identificar y ejecutar una combinación mínima de caminos básicos.

Algunas de las técnicas utilizadas en la prueba del camino básico incluyen:

1. Identificación de caminos independientes: Se identifican y enumeran todos los caminos independientes en el programa de control de flujo. Un camino independiente es aquel que no se puede alcanzar a través de otro camino.

2. Selección de caminos básicos: Se selecciona un conjunto mínimo de caminos básicos que cubra todos los nodos y aristas del grafo. Esto se logra a través de técnicas de cobertura de código, como el criterio de cobertura de todos los caminos (CACC).

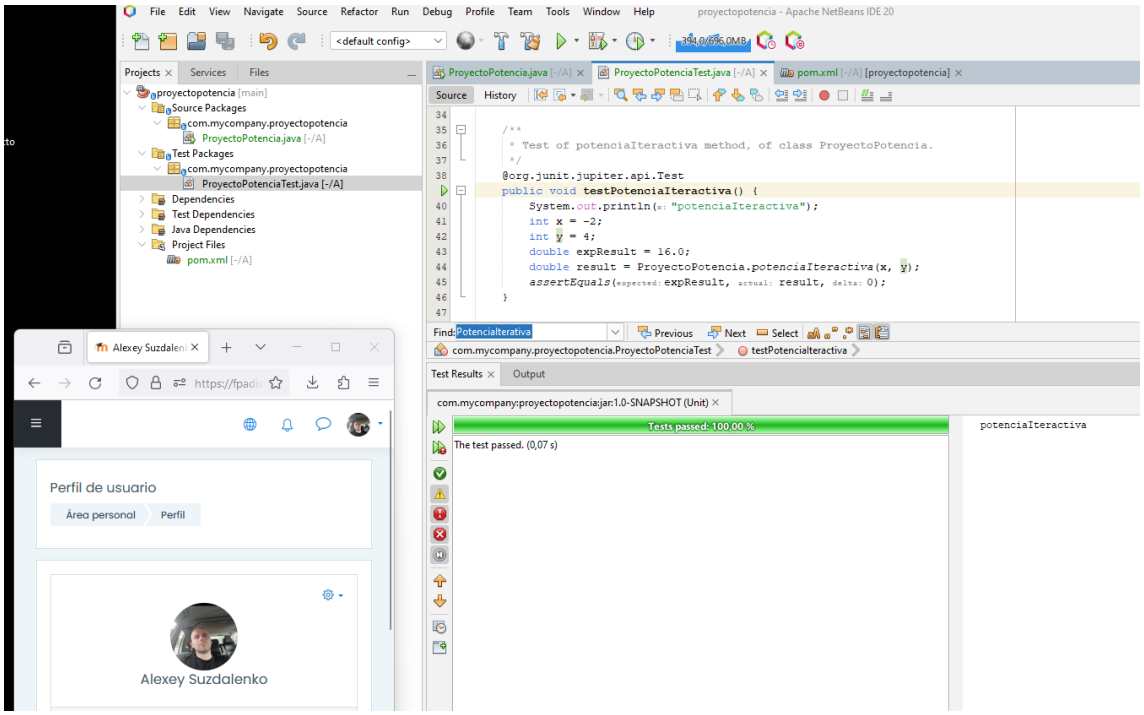
3. Generación de casos de prueba: Se generan automáticamente casos de prueba que ejecuten los caminos básicos seleccionados. Estos casos de prueba se utilizan para ejecutar el programa y verificar su comportamiento en cada uno de los escenarios identificados.

En resumen, la prueba del camino básico es una técnica de prueba de software que consiste en identificar y ejecutar una combinación mínima de caminos independientes para evaluar la cobertura de un programa de control de flujo. Esta técnica permite asegurar que se han

considerado todos los escenarios posibles y se han cubierto todas las ramas y condiciones de decisión del programa.

**7.**

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



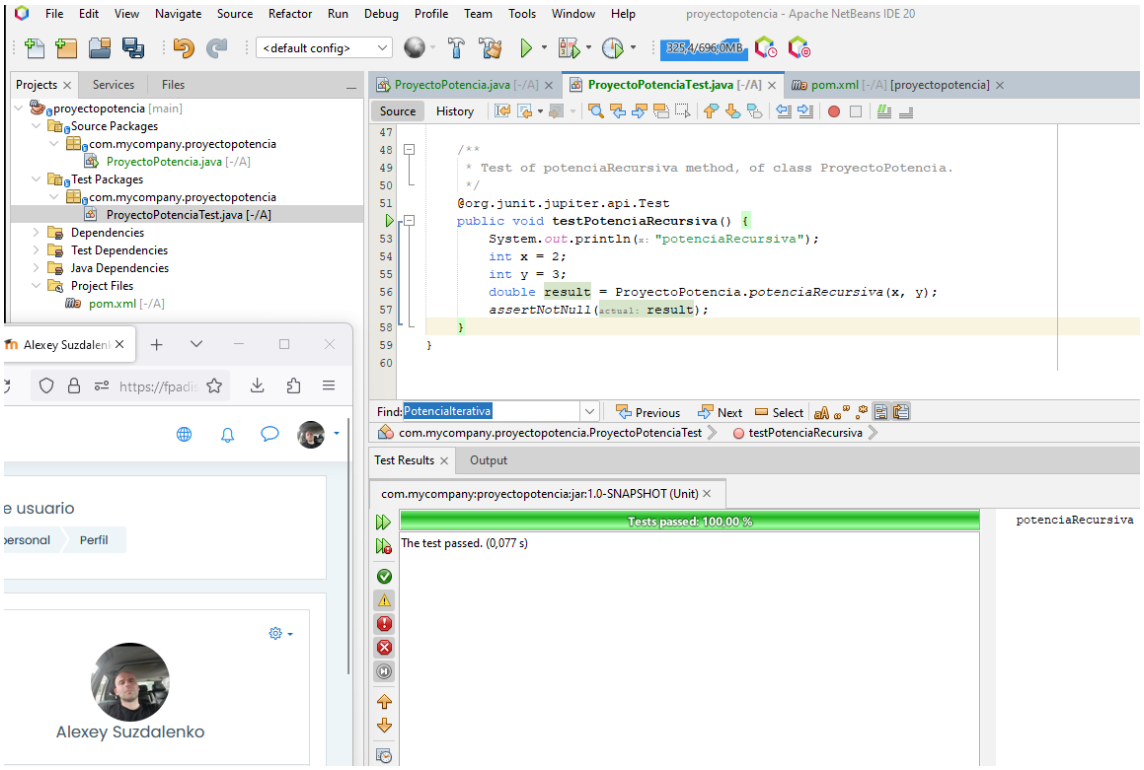

```
@org.junit.jupiter.api.Test
public void testPotenciaIteractiva() {
    System.out.println("potenciaIteractiva");
    int x = -2;
    int y = 4;
    double expResult = 16.0;
    double result = ProyectoPotencia.potenciaIteractiva(x, y);
    assertEquals(expResult, result, 0);
}
```

En este caso comprobare que se la función **potenciaIteractiva** hace la cuenta correctamente.

Usare el valor de entrada -2 y valor de potencia 4, si el resultado de test es correcto tiene que darme 16.0, inicio el test y si, el test pasa favorable.

En mi caso assertEquals(16.0, 16.0, 0); 16.0 es igual a 16.0

**2.Diseña un caso de prueba que permita verificar PotenciaRecursiva.**




```
@org.junit.jupiter.api.Test
    public void testPotenciaRecursiva() {
        System.out.println("potenciaRecursiva");
        int x = 2;
        int y = 3;
        double result = ProyectoPotencia.potenciaRecursiva(x, y);
        assertNotNull(result);
    }
```

En este caso comprobare que el resultado que obtengo desde la función **testPotenciaRecursiva** no es nulo. Tengo el valor de entrada 2 y la potencia 3, el resultado es 8.0 – no es nulo. Inicio el test y su resultado es favorable.

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

En el junit test **testPotenciaIteractiva** uso el método **assertEquals(“parámetroEsperado”, “parametroReal”, “mensajeDeError”).** En mi caso a la hora de ejecutarse **testPotenciaIteractiva** los valores en método **assertEquals** son los siguientes

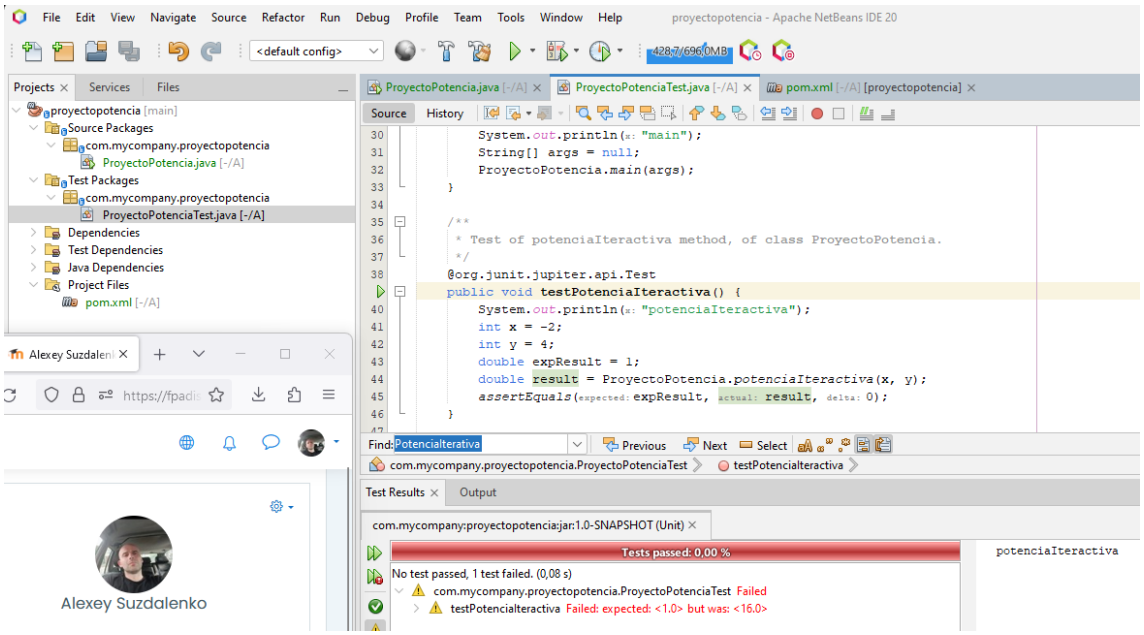
assertEquals(16.0, 16.0, 0); en otras palabras el valor esperado(16.0) es igual al que obtengo con el resultado de cálculo(16.0) por ello el test es favorable.

En el junit test **testPotenciaRecursiva** uso el método **assertNotNull**(“parametroReal”), este método los que compraba que le entra por parámetro no sea nulo, si no es nulo el test es favorable, si es nulo el test fallara. En mi caso a la hora de ejecurar **testPotenciaRecursiva** el valor que le llega por parámetro a **assertNotNull(16.0)**, ya que 16.0 no es null, pues el resultado de test es favorable.

En mi caso para hacer los test **testPotenciaIteractiva** y **testPotenciaRecursiva** he utilizado **assertEquals()** es mas restrictivo, por que comprueba que el cálculo este echo correctamente.

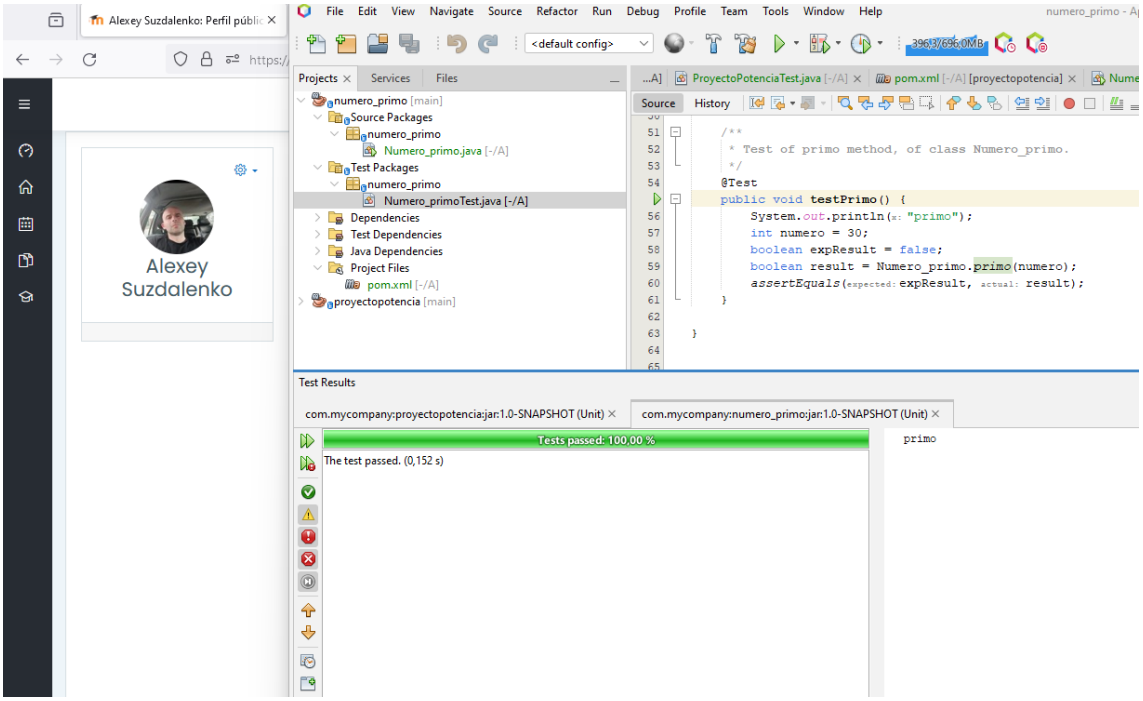
**assertNotNull()** simplemente comprueba que le resultado del cálculo no sea NULO.

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




Aquí hago test con método **testPotenciaIteractiva** y compruebo el resultado de cálculo con **assertEquals**(), en este caso el valor de calculo es 16 y yo pongo que estoy esperando el valor 1, en **assertEquals**(1 , 16, 0) se encuentran estos valores y el test falla.

## 5. Crea una prueba unitaria para el proyecto Primo




Creo prueba unitaria para el método "**primo**", el método del test se llama "**testPrimo**" y el método con el que hago la comprobación final es **assertEquals,** en **assertEquals** comparo el valor correcto(false) con el que se calcula en el método primo(30) que me devuelve es (false). Entonces en **assertEquals** se encuentran false y false. Entonces el test pasa correctamente y no hay fallos. Ya que 30 no es primo.