SANTA ANA
Y SAN RAFAEL
Madrid

COLEGIOS
MARIANISTAS

DEPURACIÓN Y REALIZACIÓN DE PRUEBAS

UD 3 Presentación

ENTORNOS DE DESARROLLO – 1ero DAM

DEPURACIÓN Y REALIZACIÓN DE PRUEBAS

ANÁLISIS DE CÓDIGO

Una de las tareas más importantes que deberemos realizar mientras desarrollamos software es asegurarnos de que **nuestro código funcionará correctamente**.

PRUEBAS DEL CÓDIGO

Es la ejecución del proceso de pruebas; consistente en la **planificación, preparación y evaluación** de un producto de software y sus artefactos de trabajo, para determinar si **satisfacen los requerimientos, demostrar que son aptos para el propósito, y detectar defectos**.

DEPURACIÓN DEL CÓDIGO

La depuración de programas es el proceso de **identificar y corregir errores** de programación. Consistente en el análisis de fallos y la reparación de defectos (**bugs**).

DEPURACIÓN Y REALIZACIÓN DE PRUEBAS

PRUEBAS DE SW: ESTRATEGIAS

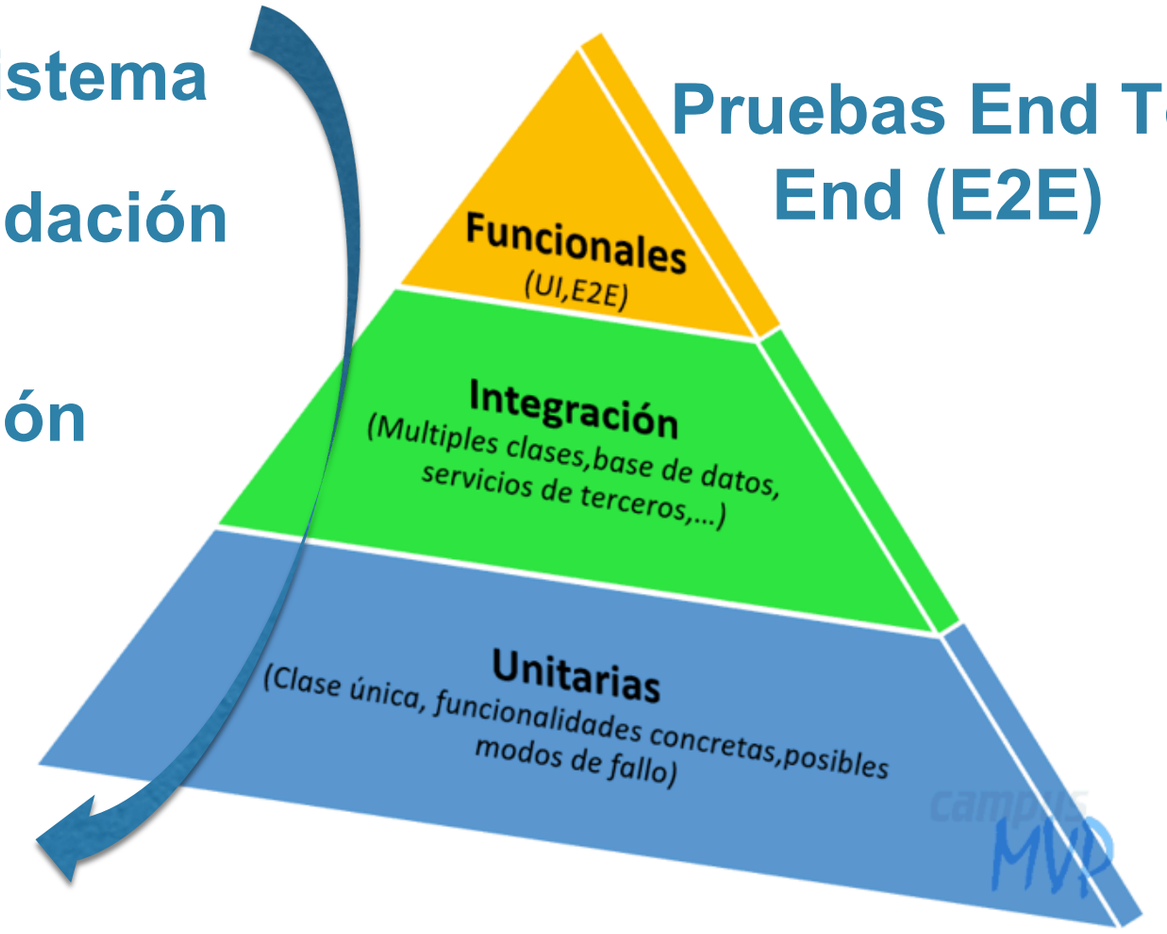
Son una espiral en éste orden:

1. **Pruebas unitarias:** Se establecen casos de prueba de cada unidad programada.
2. **Pruebas de integración**: Se observa cómo interaccionan los distintos módulos probados en las pruebas unitarias. Podrán ser **incrementales**, o **no incrementales**, las incrementales van añadiendo módulos progresivamente, y las no incrementales prueban todo junto a la vez (no recomendable).
3. **Pruebas de validación**: Pruebas realizadas por el cliente, que podrán ser pruebas **Alfa**, en las que las pruebas se hacen en presencia también del desarrollador quien tomará nota de los fallos, o pruebas **Beta**, en las que el cliente probará sólo, anotará los fallos (reales o no), y le pasará el informe al desarrollador.
4. **Pruebas de sistema:** Es la última fase, y consta de tres pruebas, la de **Recuperación**, que fuerza el fallo de software y comprueba su recuperación, la de **Seguridad**, que verifica que el sistema está protegido contra accesos ilegales, y la de **Resistencia o Stress**, que enfrenta al sistema con situaciones que demandan gran cantidad de recursos.

DEPURACIÓN Y REALIZACIÓN DE PRUEBAS

PRUEBAS DE SW: ESTRATEGIAS

4. Pruebas de Sistema
3. Pruebas de Validación
2. Pruebas de Integración
1.- Pruebas Unitarias



DEPURACIÓN Y REALIZACIÓN DE PRUEBAS

DOCUMENTACIÓN PARA LAS PRUEBAS

Plan de Pruebas: Describe el alcance, el enfoque, los recursos, y el calendario de las actividades de prueba. Identifica los elementos a probar, las características que se van a probar, las tareas que se van a realizar, el personal responsable de cada tarea, y los riesgos.

Especificaciones de la prueba: Se compone de tres documentos:

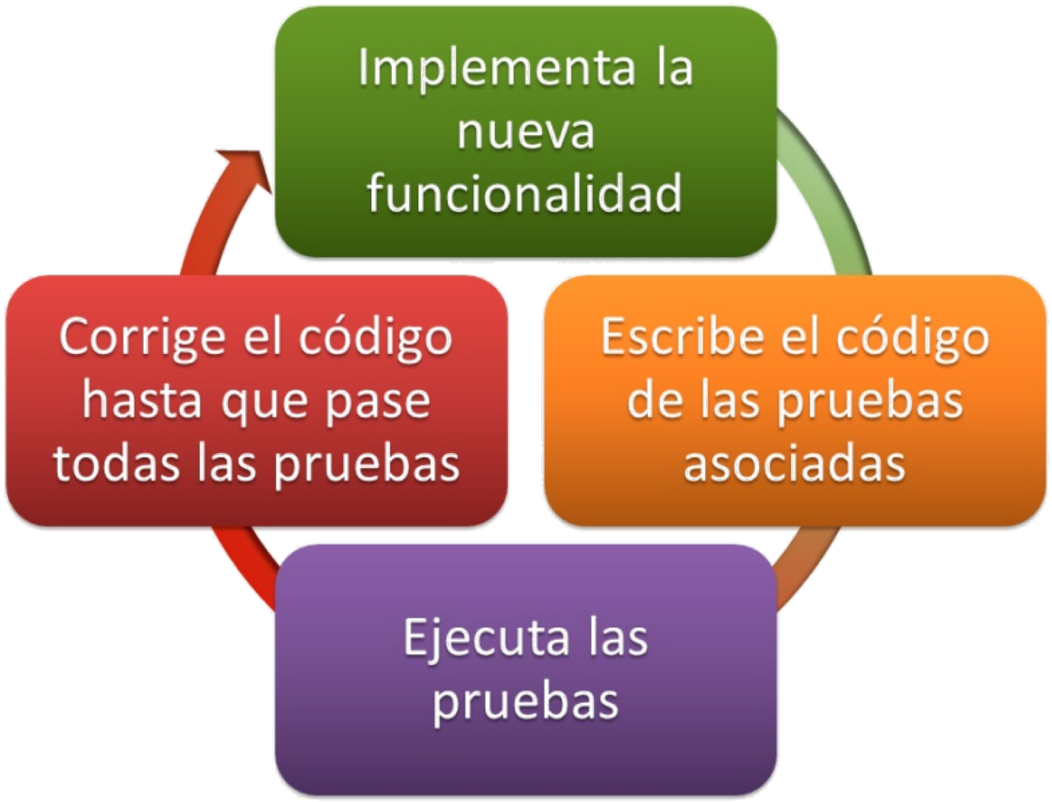
- **Especificación del diseño de la prueba** (identificación de requisitos, casos de prueba y procedimientos)
- **Especificación de los casos de prueba** (valores reales a probar y resultados previstos de salida)
- **Especificación de los procedimientos de prueba**

Informes de pruebas: Se componen de cuatro informes, el **informe de los elementos probados**, el **registro de las pruebas**, el **informe de incidentes** en la prueba, y el **informe resumen**.

DEPURACIÓN Y REALIZACIÓN DE PRUEBAS

PRUEBAS UNITARIAS

Las pruebas deberían implementarse de manera sistemática con cada nueva funcionalidad que se añada, teniendo de este modo las pruebas actualizadas en cada momento. Es importante que las **pruebas se realicen y se ejecuten de manera incremental**.



DEPURACIÓN Y REALIZACIÓN DE PRUEBAS

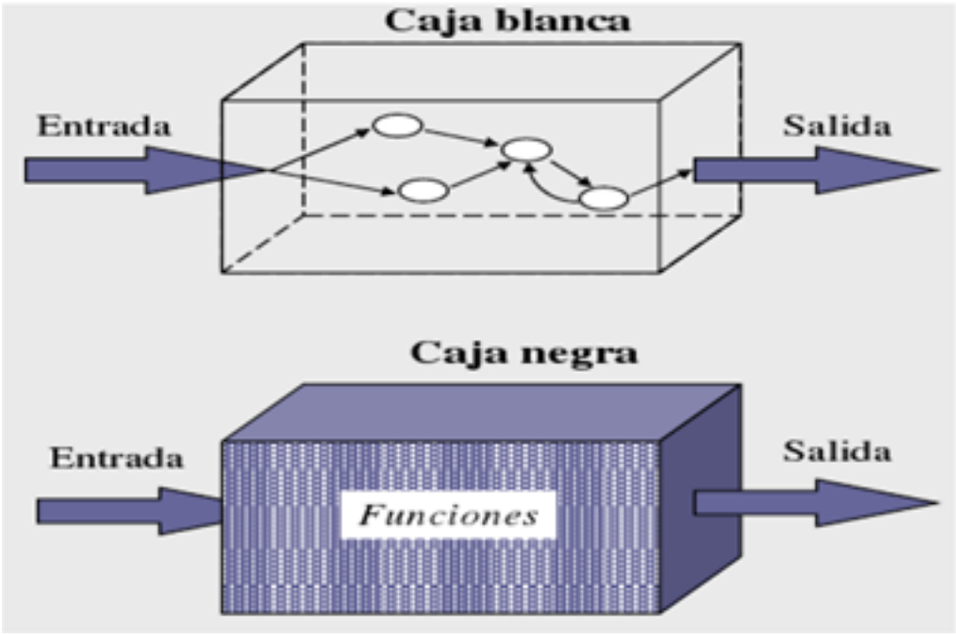
PRUEBAS UNITARIAS: CASOS DE PRUEBA

Los **casos de prueba** son una serie de condiciones que se establecen con el objetivo de determinar si la aplicación funciona correctamente según lo esperado.

Para cada tarea, pueden surgir diversos casos de prueba, teniendo en cuenta todos los factores posibles para no dejar ningún cabo suelto sin probar y evitar así que ocurran errores no conocidos.

Existen diversos tipos de casos de prueba que se definen por la naturaleza de las pruebas:

- ✓ Las pruebas de **caja blanca** van ligadas al procedimiento en sí (ejemplo: una correcta evaluación de un condicional)
- ✓ Las pruebas de **caja negra** se realizan desde el punto de vista del usuario final, su funcionalidad.



(Podemos estar probando lo mismo pero con distintos puntos de vista)

DEPURACIÓN Y REALIZACIÓN DE PRUEBAS

PRUEBAS UNITARIAS: CAJA BLANCA

Las **pruebas de caja blanca** se centran en el funcionamiento interno del programa, observando y comprobando cómo se realiza una operación. También se las conoce como **pruebas estructurales** o de **caja de cristal**.

Mediante ésta técnica los casos de prueba:

- ✓ Garantizan que se ejecutan al menos una vez todos los caminos independientes de cada módulo.
- ✓ Ejecutan todas las sentencias al menos una vez.
- ✓ Ejecutan todas las decisiones lógicas en TRUE y FALSE.
- ✓ Ejecutan todos los bucles en sus límites.
- ✓ Utilizan todas las estructuras de datos internas.

La prueba más utilizada es la **Prueba del camino básico:** Este método se basa en el principio que establece que cualquier diseño procedimental se puede representar mediante un **grafo de flujo**. La **complejidad *ciclomática*** de dicho grafo establece el número de caminos independientes, cada uno de esos caminos se corresponde con un nuevo conjunto de sentencias o una nueva condición.

DEPURACIÓN Y REALIZACIÓN DE PRUEBAS

PRUEBAS UNITARIAS: CAJA NEGRA

Las **pruebas de caja negra** se enfocan en los métodos de entrada y salida de la aplicación, no en cuestión de formato, sino de validar y controlar los datos de entrada para evitar errores y, por supuesto, al igual que en todas las pruebas, obtener los resultados esperados. También llamadas **pruebas de comportamiento**.

Con éstas pruebas se intentan detectar los errores de:

- ✓ Funcionalidades incorrectas o ausentes.
- ✓ Errores de interfaz.
- ✓ Errores en estructuras de datos o en accesos a bases de datos.
- ✓ Errores de rendimiento.
- ✓ Errores de inicialización y finalización.

Existen diferentes técnicas para confeccionar éstas pruebas, pero las más utilizadas son la **Partición equivalente o clases de equivalencia**, y el **Análisis de valores límite**.

DEPURACIÓN Y REALIZACIÓN DE PRUEBAS

PRUEBAS UNITARIAS: HERRAMIENTAS

Para realizar pruebas unitarias en JAVA (tanto de caja blanco como de caja negra) usaremos el **framework** (entorno de trabajo)

JUnit: conjunto de bibliotecas utilizadas en programación para hacer pruebas unitarias de aplicaciones en Java

DEPURACIÓN Y REALIZACIÓN DE PRUEBAS

PRUEBAS UNITARIAS: JUNIT

- ✓ Para realizar pruebas unitarias en JAVA (tanto de caja blanco como de caja negra) usaremos el **framework** (entorno de trabajo) **JUnit** que está integrado en Eclipse con Java.

- ✓ **JUnit**: conjunto de bibliotecas utilizadas en programación para hacer pruebas unitarias de aplicaciones en Java. Es un entorno para crear **pruebas unitarias**.

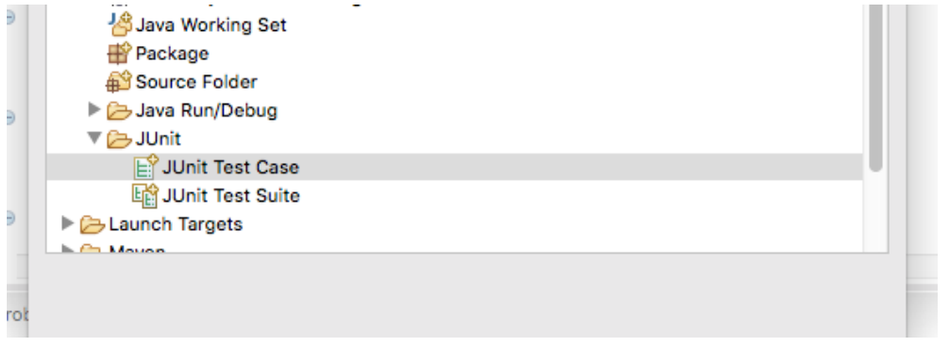
- ✓ El desarrollador del código (1) crea el programa que ofrece la solución y además (2) crea una **suite o conjunto de pruebas** para que otros (Departamento Pruebas, Cliente) verifiquen que el funcionamiento del programa es correcto (**pruebas estructurales** o de **caja blanca**) y que el comportamiento del programa es el deseado (**pruebas de comportamiento** o de **caja negra**).

- ✓ Hay dos maneras de crear pruebas unitarias de nuestro código:
  1. Incluir la prueba en el código de manera manual. Tiene como desventaja que es incómodo, molesto e incluso podemos equivocarnos en la misma prueba.
  2. Utilizar librerías y herramientas que **independiza el código de las pruebas**.

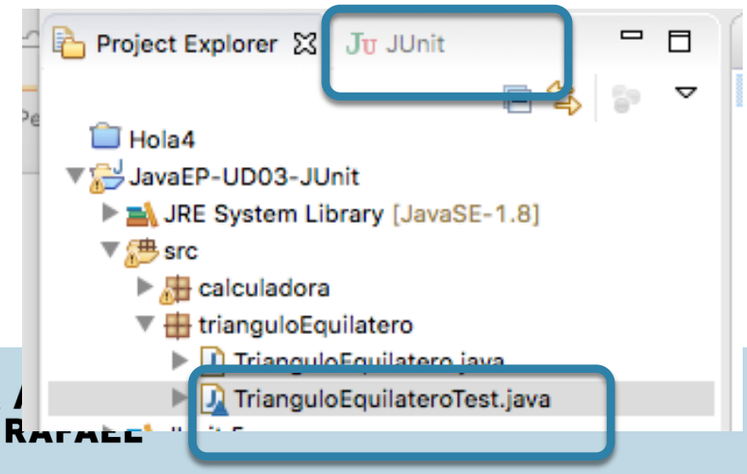
DEPURACIÓN Y REALIZACIÓN DE PRUEBAS

JUNIT: PLAN DE PRUEBAS DE UNA CLASE

1. Partimos de una clase con su constructor y sus métodos: Triángulo Equilátero
2. Creamos un nuevo recurso en Eclipse, bajo el mismo paquete de la clase: New → **JUnit Test Case**



3. Se crea una nueva clase **nombreClaseTest.java**




```java
public class TrianguloEquilatero {
    private double lado;
    private double altura;

    TrianguloEquilatero (double lado, double altura){
        this.lado = lado;
        this.altura = altura;
    }

    public double getLado() {
        return lado;
    }

    public void setLado(double lado) {
        this.lado = lado;
    }

    public double getAltura() {
        return altura;
    }

    public void setAltura(double altura) {
        this.altura = altura;
    }

    double calculaArea() {
        double area = 0;
        if (altura==0){
            throw new ArithmeticException("No es posible que la altura sea cero");
        } else {
            //supongamos que esta fórmula fuese la correcta del cálculo de área
            //la hemos modificado para probar el salto de la excepción
            area = (lado * 2)/altura;
        }
        return area;
    }

    double calculaPerimetro() {
        double perimetro = lado*3;
        return perimetro;
    }

}
```

DEPURACIÓN Y REALIZACIÓN DE PRUEBAS

JUNIT: MÉTODO @TEST

4. La clase de Test nos va a permitir definir métodos de prueba utilizando la directiva **@Test. JUnit** lo reconoce como método para lanzar como prueba.
5. Desde Eclipse ejecutamos (RUN) la clase de prueba y desde el entorno de **JUnit**, comprobamos el resultado de la prueba: error por la función **fail**.
6. Creamos un método de prueba para comprobar el cálculo del área del triángulo (método calculaArea() → Probar que se ejecuta bien o que se ejecuta mal)
7. **assertEquals** es una función o aserto de JUnit que ayuda en las pruebas.

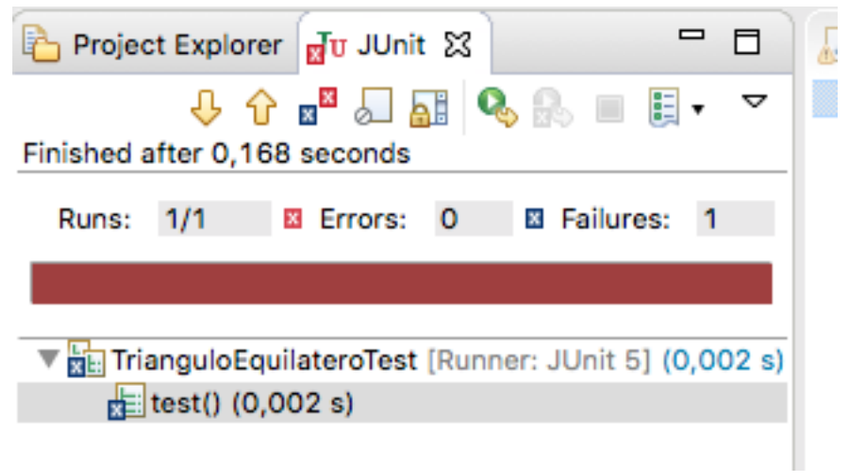
```java
package trianguloEquilatero;

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

class TrianguloEquilateroTest {

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

}
```




```java
//Test para probar CalculaArea con valores correctos
    @Test
    void testCalculaArea() {
        System.out.println("Test Area");
        TrianguloEquilatero tprueba = new TrianguloEquilatero(3,5);
        double resultadoArea = tprueba.calculaArea();
        double esperado = 7.5; //(3*5)2
        assertEquals(esperado,resultadoArea);
    }
```

DEPURACIÓN Y REALIZACIÓN DE PRUEBAS

JUNIT: @BEFOREEACH

8. Creamos otro método **@Test** para probar el cálculo del perímetro.

7. Crear un objeto se realiza en todas las pruebas. Podemos pasarlo a un método que se ejecuta antes de cada prueba o método **@Test**: **@BeforeEach**.
8. Por cada directiva de **JUnit** hay que incluir su correspondiente **Api**.

9. Ejecutamos y vemos mensajes.

10. Al igual que hay un método que se ejecuta siempre antes de cada prueba, también hay una directiva para indicar que se ejecute siempre después de cada clase: **@AfterEach**

```
//Test para probar CalculaPerimetro con valores correctos
@Test
void testCalculaPerimetro() {
    System.out.println("Test Perimetro");
    TrianguloEquilatero tprueba = new TrianguloEquilatero(3,5);
    double resultadoPerimetro = tprueba.calculaPerimetro();
    double esperado = 9; //3*3
    assertEquals(esperado,resultadoPerimetro);
}

 //Se ejecuta antes de cada @Test
    @BeforeEach  //Before en Junit4
     void testBefore() {
         System.out.println("Empieza un Test");
         lado = 3.0;
         altura = 5.0;
         tprueba = new TrianguloEquilatero(lado,altura);
     }
```

```
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;
```

```
Empieza un Test
Test Area
Empieza un Test
Test Perimetro
```

```
    //Se ejecuta después de cada @Test
    @AfterEach  //After en Junit4
    void testAfter() {
        System.out.println("Acaba un Test");
    }
```

DEPURACIÓN Y REALIZACIÓN DE PRUEBAS

JUNIT: @BEFOREALL

11. También hay una directiva para indicar a **JUnit** que dicho método se ejecuta siempre al inicio de todas las pruebas **@BeforeAll** y otra directiva para ejecutarse al final de todas las pruebas: **@AfterAll**

```java
//Se ejecuta al principio de la prueba
@BeforeAll //BeforeClass en Junit4 sin ser static
static void testBeforeAll() {
    System.out.println("Empieza el Junit");
}

//Se ejecuta al final de la prueba
@AfterAll  //AfterClass en Junit4 sin ser static
static void testAfterAll() {
    System.out.println("Acaba el Junit");
}
```

12. En la clase original tenemos una clase que lanza una excepción. Es muy importante probar que efectivamente la excepción salta cuando es necesario. Usamos el aserto **assertThrows**

```java
double calculaArea() {
    double area = 0;
    if (altura==0){
        throw new ArithmeticException("No es posible que la altura sea cero");
    } else {
        //supongamos que esta fórmula fuese la correcta del cálculo de área
        //la hemos modificado para probar el salto de la excepción
        area = (lado * 2)/altura;
    }
    return area;
}
```

```java
//Test para probar CalculaArea con fallo por excepción
@Test //(expected = ArithmeticException.class) en Junit4
void testCalculaAreaError() {
    System.out.println("Test Area Error");
    //TrianguloEquilatero tprueba = new TrianguloEquilatero(3,0);
    tprueba.setAltura(0);
    assertThrows(ArithmeticException.class, () -> {
        tprueba.calculaArea();
        });
}
```

DEPURACIÓN Y REALIZACIÓN DE PRUEBAS

DEPURACIÓN DEL CÓDIGO

La **depuración** es uno de los procesos más importantes en el desarrollo de software, nos permite identificar y corregir errores de programación mediante la ejecución controlada del software.

Esta tarea la realiza el **depurador**, uno de los componentes básicos de un entorno de desarrollo, es, sin duda, **una de las herramientas más importantes de un IDE.**

Gracias al **depurador**, podemos ver el **proceso** de un programa paso a paso, observando los valores de nuestros métodos, variables y objetos, lo cual facilita sumamente la tarea, o podemos simplemente establecer puntos de control que **interrumpen** la ejecución del programa mostrándonos el código en donde pusimos el punto de **interrupción** con los valores actuales.

DEPURACIÓN Y REALIZACIÓN DE PRUEBAS

DEPURACIÓN DEL CÓDIGO

- ✓ Podemos situar **puntos de ruptura** o **puntos de interrupción** en líneas concretas de nuestro código fuente para que el depurador detenga la ejecución del programa cuando alcance uno de ellos.
- ✓ Se pueden colocar todos los puntos de **interrupción** que se desee, lo cual resulta muy útil cuando se está depurando un error que tiene una larga **pila de llamadas**.
- ✓ Los puntos de interrupción se pueden personalizar, añadiendo una condición o un filtro.
- ✓ Las condiciones que podemos utilizar en el **punto de interrupción condicional** no tienen límite y son extremadamente útiles para establecer puntos de interrupción muy concretos, especialmente en estructuras de bucle.
- ✓ También podemos utilizar **puntos de seguimiento** que tienen una funcionalidad semejante a los puntos de interrupción. La diferencia radica en que un punto de seguimiento no detiene el programa. Son útiles para monitorizar o inspeccionar valores en diferentes momentos de ejecución.
- ✓ Mediante la opción de **INSPECCIÓN**, podemos ver los valores de las variables, propiedades de los objetos, elementos, etc, y cómo van cambiando mientras depuramos.

DEPURACIÓN Y REALIZACIÓN DE PRUEBAS

HERRAMIENTA DE DEPURACIÓN

- ✓ El IDE Eclipse nos ofrece un entorno completo de Depuración