Practica 8: Pruebas, JUnit y Mocks

Glosario

- Tipos de Pruebas
- Framework Junit
  - Introducción
  - Test Class
    - Test Method – Caso de prueba
      - Condiciones de Aceptación: Assert
      - ¿Qué significa igual?
  - Arrange: Fixtures
  - Probar Excepciones
  - Pruebas de rendimiento
  - Organización y Ejecución de pruebas
    - Deshabilitar Test.
    - Tag
    - Nested
- Instalar herramienta de Cobertura. ECLEMMA
- Mock. (Simuladores)
  - Stubs
  - Mockito
  - Secuencia de uso
- Ejemplo de Junit con Eclipse

Tipos de Pruebas

- Clasificación por nivel
  - Test Unitarios
  - Test de Integración
  - Test de Sistema
  - Test de Integración de Sistemas
- Pruebas funcionales
  - Test de Aceptación
    - α, β testing
  - Test de Regresión
- Pruebas no funcionales
  - Rendimiento,
  - Carga, Estrés, Resistencia
  - Seguridad,
  - usabilidad,
  - Mantenibilidad,
  - …

Tipos de Pruebas

- Clasificación por nivel
  - **SUT**: Sistema bajo Test
- Objetivo
  - Probar el correcto funcionamiento de cada SUT
- Tipos
  - **Test Unitarios**:
    - El SUT es el elemento de software más simple posible.
    - Se corresponde con un módulo o en POO con una clase o un objeto.
  - **Test de Integración**:
    - El SUT incluye a varios módulos operando coordinadamente en un contexto de funcionamiento, por ejemplo, para entregar una funcionalidad.
    - En *POO* pueden ser varias clases colaborando en una funcionalidad.
  - **Test de Sistema**
    - El SUT es el sistema software completo.

Tipos de Pruebas.

Test Unitarios. Ventajas

1. Facilitan la identificación y localización de errores
2. Facilitan el cambio.
   - Hacer pruebas repetibles y automatizadas que satisfacen el punto 1 proporciona pruebas de regresión que aumentan la confianza en los cambios.
3. Simplifican la integración.
   - Tener los módulos probados simplifica la identificación de errores en la integración.
4. Separación de Interfaz e implementación.
   - Ya que los casos de prueba ejecutan llamadas a los módulos.
     - Se pueden refactorizar Pruebas y Código sin consecuencias
     - Los casos de Prueba documentan el código ya que explican como se usa.

Tipos de Pruebas

- Pruebas funcionales
  - **Test de Regresión**: Son las pruebas de software que intentan descubrir errores producidos por la realización de cambios en el software.

NOTA: El almacenamiento y automatización de pruebas de unidad e integración entregan un banco de pruebas de regresión.

NOTA: Las metodologías ágiles, en particular TDD o BDD, se esfuerzan en generar pruebas de regresión tan buenas como el código. Cualquier error introducido causa la ruptura de los test de regresión y, en particular, de test simples que permiten identificar fácilmente su origen.

El framework JUnit

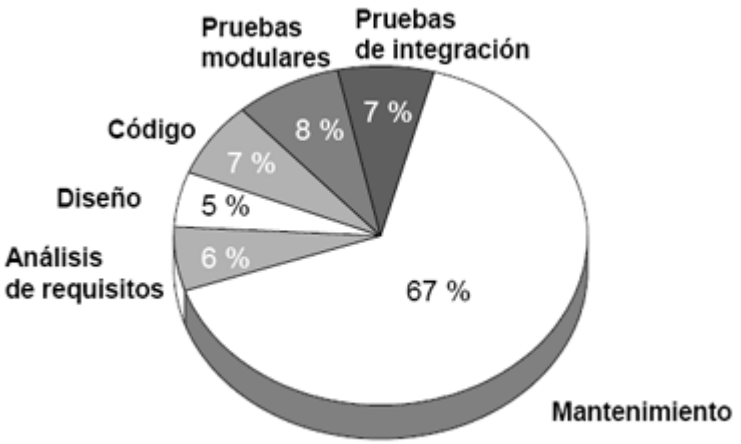
- Introducción
- Test Class
  - Test Method – Caso de prueba
    - Condiciones de Aceptación: Assert
    - ¿Qué significa igual?
- Arrange: Fixtures
- Probar Excepciones
- Pruebas de rendimiento
- Organización y Ejecución de pruebas
  - Deshabilitar Test.
  - Tag
  - Nested

El framework JUnit

- JUnit es un “framework” para automatizar las pruebas de programas Java
- Escrito por Erich Gamma y Kent Beck
- Open Source, disponible en https://junit.org/junit5/
- Es uno de los frameworks de tests unitarios conocidos colectivamente como xUnit

El framework JUnit

- Permite crear distintos tipos de pruebas.
  - Las pruebas son código y deben ser tratadas con la misma rigurosidad.
    - Las pruebas también pueden contener errores.
    - Código simple y con lógica fácil de entender



  - Proporciona las siguientes características.
    - Pruebas repetibles y automatizables.
    - Pruebas Independientes y completas (cobertura)

El framework JUnit

- Utilidades para crear clases de prueba.
  - Los casos de prueba se codifican como métodos anotados que pueden ser reejecutados.
  - Los casos de prueba cuentan con afirmaciones que verifican los resultados.
  - Los casos de prueba pueden complementarse con métodos anotados adicionales anteriores y posteriores al caso de prueba.
  - Se pueden controlar excepciones y tiempos de espera.
- Permite organizar las pruebas.
- Ejecución controlada de las pruebas.

Test Class

Source Folder: code

```
package operaciones;

public class Aritmetica {

    public Aritmetica() {
        super();
    }

    public int suma(int a, int b) {
        return a + b;
    }
}
```

Source Folder: test

```
package operaciones;

import static org.junit.Assert.fail;
import org.junit.Test;
import code.Aritmetica;

public class TestCalculadora {

    @Test
    public void testSuma() {
        fail("Not yet implemented");
    }
}
```

- La clase *TestCalculadora* prueba *Aritmetica*. Incluye **n** métodos **Test**.
- "Source Folder" distintos para código y pruebas, idénticos packages.
- No es necesario main.

Métodos Test

- Los métodos Test se anotan con @Test
- Tienen que ser ~~públicos,~~ sin parámetros y void

```
public class TestCalculadora {
   @Test
   void testSuma() {
      fail("Not yet implemented");
   }

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

- En el proceso de prueba JUnit crea una instancia de la Test Class para ejecutar cada uno de sus @Test métodos
- **Los métodos @Test se ejecutan en cualquier orden**.

Comentar un Test
Junit 5. Jupiter

- Anotación **@DisplayName**. Sustituye al nombre del método en las herramientas de test permitiendo introducir nombres de métodos más compactos.
- Ignorar un test.
  - Anotación **@Disabled** para comentar un test.

```
@Disabled("Mensaje explicativo")
@Test
@DisplayName("Test para probar la suma")
public void testEnSuma() {
   fail("Not yet implemented");
}
```

Métodos Test

- Creamos los métodos de test con el patrón AAA
  - Arrange (Preparar. Creación manual de objetos esperados)
  - Act (Actuar. Ejecución de elementos a probar.)
  - Assert (Afirmar. Condición de aceptación)

```
@Test
public void testSuma() {
   // Arrange
   Aritmetica calculadora = new Aritmetica();
   int esperado=2;
   // Act
   int real=calculadora.suma(1, 1);
   // Assert
   assertEquals(esperado,real,"Comentario de fallo");
}
```

Condiciones de aceptación: Assert

Método assertxxx()	Qué comprueba
fail([Msg])	hace que el test termine con fallo
assertEquals(E, R,[Msg])	comprueba que Esperado (E) sea igual a Real (R). Utiliza el método equals().
assertSame(E, R, [Msg])	comprueba que los objetos E y R son el mismo objeto. Utiliza operador ==. Equals sin sobreescribir.
assertNull(R, [Msg])	comprueba que el elemento es nulo.
assertTrue(C, [Msg])	Verifica que la condición es true.
assertNotEquals, assertNotSame, assertNotNull, assertFalse,	Los métodos anteriores negados.
assertThrows(E<T>, Exe,[Msg])	comprueba que el Exe lanza la excepción del tipo T
assertAll([Msg], Exe)	comprueba cada assert en Exe
assertTimeout(Ti,Exe) **assertTimeoutPeemptively(Ti,Exe)**	Comprueban si el Exe dura más del Ti, assertTimeout continua la ejecución pero assertTimeoutPeemptively la otra la detiene en Ti.
assertArrayEquals(E, R, [Msg])	comprueba que los arrays E y R son iguales
assertIterableEquals(E,R,[Msg])	comprueba que los iterables son “deeply equal”
assertLinesMatch(E, R)	comprueba el match de dos listas de Strings

Condiciones de aceptación: Assert

Método assertxxx()	Qué comprueba
assertEquals(E, R,[Msg])	comprueba que Esperado (E) sea igual a Real (R). Utiliza el método equals().
assertSame(E, R, [Msg])	comprueba que los objetos E y R son el mismo objeto. Utiliza operador ==. Equals sin sobreescribir.
assertNotEquals, assertNotSame	Los métodos anteriores negados.

- E, R puede ser cualquier tipo de objeto.
- La función equal del objeto puede tener que redefinirse para que assertEquals funcione propiamente.

Comparando con assertEquals.

- ¿Cuando dos ficheros son iguales?.
  - File1==File2
  - Son el mismo objeto. Tienen la misma dirección de memoria.
- Dos objetos distintos pueden guardar ficheros iguales, por tanto:
  - File1!=File2
  - Pero los ficheros serían iguales
- Se debería usar:
  - File1.equals(File2)
  - Pero por defecto tiene el mismo comportamiento
- Es necesario reescribir File.equals para que:
  - File1.equals(File2)==true ⇔ File1.raw==File2.raw.
  - Return this.raw==File2.raw; //Es insuficiente.

Comparando con assertEquals.

- **Reglas que sigue el método equals**
  - **Reflexivo**: Para cualquier referencia al valor **a**, a.equals(a) debe devolver true.
  - **Simétrico**: Para cualquier referencia a los valores **a** y **b**, **a**.equals(**b**) debe devolver true si y solo si **b**.equals(**a**) es true.
  - **Transitivo**. Para cualquier referencia a los valores **a**, **b** y **c**, si a.equals(b) devuelve true y b.equals(c) devuelve true, entonces a.equals(c) debe devolver true.
  - **Consistente**: Para cualquier referencia a los valores **a** y **b**, múltiples invocaciones a a.equals(b) consistentemente devolveran true o false, si es que los valores utilizados para la comparación de los objetos no ha sido modificada.
  - Para cualquier referencia no nula al valor **a**, a.equals(null), debe devolver false.

Comparando con assertEquals.

- ¿Cuando dos ficheros son iguales?.
  - Sobreescribir equals.

```
// Autogenerado eclipse
@Override
public boolean equals(Object obj) {
   if (this == obj)
      return true;
   if (obj == null)
      return false;
   if (getClass() != obj.getClass())
      return false;
   DatosFile other = (DatosFile) obj;
   if (raw != other.raw)
      return false;
   return true;
}
```

Métodos Test

- Cuando dos ficheros son iguales.
  - Sólo queremos comparar sus contenidos.
  - Si se sobreescribe equals es **OBLIGATORIO** sobreescribir hashCode generándolo en base a los campos que se utilizan en la comparación con equals.
  - Fundamental para que funcionen correctamente las listas del objeto.
  - Dos objetos iguales por equals deben devolver el mismo hashcode

```
@Override
public int hashCode() {
final int prime = 31;
int result = 1;
result = prime * result + raw;
return result;
}
```

Concepto: Fixture

- Las pruebas, igual que el código debe refactorizarse para optimizarlas.
- Para reusar elementos comunes en distintos casos de prueba usamos ***fixtures.*** (≈ elementos fijos)
  - **Es un estado fijo de un conjunto de objetos usados como línea base para test.**
  - Son atributos o variables miembro de la clase Test
  - Se les asigna valor en métodos ejecutados antes de cada método test de la clase.
  - Se liberan en *métodos ejecutados después de cada método test. (Finalización del test)*

Junit 5. Jupiter

- Se utilizan anotaciones para definir métodos pre/post Clase
  - **@BeforeAll**
  - **@AfterAll**

  y pre/post Test.
  - **@BeforeEach**
  - **@AfterEach**
- Los métodos pre/post Clase, **@BeforeAll** y **@AfterAll**, son estáticos.

```
import org.junit.jupiter.api.*;

class JUnit5NestedExampleTest {

    @BeforeAll
    static void setUpAll() throws  Exception{
        System.out.println("Before all test methods");
    }

    @BeforeEach
    void setUpEach() throws  Exception{
        System.out.println("Before each test method");
    }

    @AfterEach
    void tearDownEach() throws  Exception{
        System.out.println("After each test method");
    }

    @AfterAll
    static void tearDownAll() throws  Exception{
        System.out.println("After all test methods");
    }
}
```

Test de excepciones
Junit 5

- JUnit 5 gestiona el tratamiento de las excepciones a través de funciones Lambda que las generan con la llamada al SUT.

```
@Test
@DisplayName("Check the expectThrows")
void checkExceptions(){
   String message = "Paradigma rules!";
   Throwable exception = assertThrows(IllegalArgumentException.class,
       () -> {throw new IllegalArgumentException(message);
   });
   assertEquals(message, exception.getMessage(), "No es el msg");
}
```

- Después de verificado que la excepción se recibe es posible incluir tantas aserciones sobre el resultado como se desee.
- Las excepciones se tratan como Clases de Equivalencia "NO VÁLIDAS". Esto es, se genera un caso de prueba por cada una, y en cada C.P. sólo se pone a prueba la excepción.

Test de Rendimiento
Junit 5

- JUnit 5 usa:
  - **assertTimeout**(java.time.Duration timeout, Executable exe).
  - **assertTimeoutPreemptively** (java.time.Duration timeout, Executable exe).

```
@Test
void timeoutNotExceeded() {
   // The following assertion succeeds.
   assertTimeout(ofMinutes(2), () -> {
// Perform task that takes less than 2 minutes.
   });
}
```

- La diferencia es que **assertTimeoutPreemptively** detiene la ejecución cuando se supera el timeout y **assertTimeout** no, e informa del tiempo total de ejecución.
- Es posible recoger retornos de la función invocada y hacer assert sobre ellos

Condiciones de aceptación: assert

Junit 5

- ***assertAll().*** Verifica aserciones múltiples. El éxito o fallo de cada una se comprueba de forma independiente.

```
assertAll("Prueba de multiples cosas",
        () -> assertTrue(true),
        () -> assertTrue(false),
        () -> assertEquals(1, 1),
        () -> assertEquals(1, 2),
        () -> assertEquals(5, 2));
```

```
org.opentest4j.MultipleFailuresError: Prueba de multiples
cosas (3 failures)
  <no message> in org.opentest4j.AssertionFailedError
  expected: <1> but was: <2>
  expected: <5> but was: <2>
```

Organizar Test
JUnit 5

- Marcar y filtrar.
  - @Tag(“tag”)
  - Filtrar en Eclipse: Run As/Run Configurations…
    - Include and exclude tags: Configure
- @Nested

```
@DisplayName("A stack")
class TestingAStackDemo {
    Stack<Object> stack;
    @Test
    @DisplayName("is instantiated with new Stack()")
    void isInstantiatedWithNew() { new Stack<>(); }

    @Nested
    @DisplayName("when new")
    class WhenNew {
        @BeforeEach
        void createNewStack() { stack = new Stack<>(); }
        @Test
        @DisplayName("is empty")
        void isEmpty() { assertTrue(stack.isEmpty()); }

        @Test
        @DisplayName("throws EmptyStackException when popped")
        void throwsExceptionWhenPopped() {
            assertThrows(EmptyStackException.class, () -> stack.pop());
        }
    }
}
```

Test Parametrizados
Junit 5

- @ParameterizedTest
- Fuente de argumentos
  - @CsvSource - @CsvFileSource
  - @EnumSource - @MethodSource

```
@ParameterizedTest
@CsvSource({ "2, 1, 1", "3, 2, 1",  "4, 3, 1" })
public void testSuma(int esperado, int num1, int num2) {
  Aritmetica calculadora = new Aritmetica();
  int calculado = calculadora.suma(num1, num2);
  assertEquals(esperado, calculado, 0);
}
```

- **Otras fuentes**. En métodos con un único argumento
  - @NullSource, @EmptySource, @NullAndEmptySource, @ValueSource

Revisar la cobertura

- ECLEMMA
  - Es una herramienta de cobertura pero no es la única
- Instalación
  - Buscar e instalar en “Help/EclipseMarketplace”
  - Instalación manual
    - http://www.eclemma.org/installation.html
- Ejecución
  - “Run/Coverage as” o menú contextual.

Comprobar en los test de prueba

- El test unitario no puede modificar el estado del sistema. Los test se ejecutan en cualquier orden
- Un test unitario debe ser pequeño. Prueba una sola funcionalidad para un único objeto.
- Usar nombres descriptivos para los test. No importa que sean largos, usar @DisplayName.
- Indicar un mensaje en todas las llamadas assertXXXX([parámetros],”Comentario obligatorio”)
- *Hacer en el código los cambios que os sugiera el resultado de las pruebas*. Hacer que los test mejoren el código.

Simuladores. MOCKS

- Los mocks abordan el problema de las clases colaboradoras.
- Para responder a una función una clase, que sería el SUT, debe apoyarse en otras que serían las clases colaboradoras.
- El Mock permite SIMULAR el comportamiento de las clases colaboradoras que pueden, o no, estar implementadas en el sistema.

Simuladores. MOCKS

- Son objetos similares a stubs
  - Clases con comportamientos mínimos
  - Ejemplo: método que devuelve una temperatura

```
public float getTemperatura() {
      return 28.0;
   }
```

  - Los stubs también pueden grabar información sobre llamadas.
- Los mocks se centran en el comportamiento
  - Permiten variar el comportamiento
  - Permiten simular comportamientos complejos
  - Recuerdan las interacciones

Simuladores. MOCKS

- Mockito es OpenSource
  - http://mockito.org/
- La implementación pasa por varias etapas
  - Creación del objeto mock
  - Descripción del comportamiento esperado
  - Llamadas al SUT (Uso de los mocks)
  - Verificación que la interacción con los mocks es la esperada.
  - Finalización. Garantizar que los objetos no se cambian de estado. Borrado y reset.

Ejemplo Test Mock.

Partimos de las siguientes dos clases

Clase A será la que se Simule.

```
public class ClaseA {
    public ClaseA() {
    }

    public void func1() {
    }

    public void func2() {
    }

    public int func3(int a, int b) {
    return a + b;
    }
}
```

```
public class ClaseB {
    ClaseA ca;

    public ClaseA getCa() {
    return ca;
    }
    public void setCa(ClaseA ca) {
    this.ca = ca;
    }
    public ClaseB() {super();}
    public void met1() {
    ca.func1();
    ca.func1();
    System.out.println(ca.func3(5, 6));
    }
    public void met2() {
    ca.func2();
    }
}
```

Simuladores. MOCKS

- La implementación pasa por varias etapas
  - **Creación del objeto mock**
    - Sólo necesitamos el esqueleto de la clase a simular
      - Es válido usar la clase total o parcialmente implementada pero declarando, al menos, todos los métodos a simular.
    - En todos los casos el mock sustituye al objeto original
      - No es un objeto de la clase simulada
      - Las respuestas a sus métodos son los que configuremos en el mock o por defecto 0 o nulls.

Ejemplo Test Mock.

```
@Test
public void test() {
    ClaseB cb = new ClaseB(); //SUT
    ClaseA ca = Mockito.mock(ClaseA.class);
    cb.setCa(ca);

    Mockito.when(ca.func3(Mockito.anyInt(),
    Mockito.anyInt())).thenReturn(Integer.valueOf(5));

    cb.met1();
    int result=ca.func3(5, 6);

    assertEquals(5, result);
    Mockito.verify(ca, Mockito.times(2)).func3(Mockito.anyInt(),
    Mockito.anyInt());
}
```

Simuladores. MOCKS

- La implementación pasa por varias etapas
  - Creación del objeto mock
  - **Descripción del comportamiento esperado**
    - CUANDO invocamos algún método con parámetros concretos o descritos por patrones ENTONCES el mock genera un resultado

Ejemplo Test Mock.

```
@Test
public void test() {
    ClaseB cb = new ClaseB(); //SUT
    ClaseA ca = Mockito.mock(ClaseA.class);
    cb.setCa(ca);

    Mockito.when(ca.func3(Mockito.anyInt(),
    Mockito.anyInt())).thenReturn(Integer.valueOf(5));

    cb.met1();
    int result=ca.func3(5, 6);

    assertEquals(5, result);
    Mockito.verify(ca, Mockito.times(2)).func3(Mockito.anyInt(),
    Mockito.anyInt());
}
```

Simuladores. MOCKS

- La implementación pasa por varias etapas
  - Creación del objeto mock
  - Descripción del comportamiento esperado
  - **Llamadas al SUT (Uso de los mocks)**
    - SUT (System Under Test) Es el objeto cuyo comportamiento y estado pretendemos probar.
    - El SUT debe invocar en sus clases de apoyo a los mock creados en el primer paso.
    - El SUT debe invocar el comportamiento del mock descrito en el segundo de los pasos.

Ejemplo Test Mock.

```
@Test
public void test() {
    ClaseB cb = new ClaseB(); //SUT
    ClaseA ca = Mockito.mock(ClaseA.class);
    cb.setCa(ca);

    Mockito.when(ca.func3(Mockito.anyInt(),
    Mockito.anyInt())).thenReturn(Integer.valueOf(5));

    cb.met1(); // OJO: llamada directa a la clase simulada
    int result=ca.func3(5, 6);

    assertEquals(5, result);
    Mockito.verify(ca, Mockito.times(2)).func3(Mockito.anyInt(),
    Mockito.anyInt());
}
```

Simuladores. MOCKS

- La implementación pasa por varias etapas
  - Creación del objeto mock
  - Descripción del comportamiento esperado
  - Llamadas al SUT (Uso de los mocks)
  - **Verificación que la interacción con los mocks es la esperada.**
    - Verificamos Estados
      - Comprobamos que el SUT entrega las salidas correctas.
    - Verificamos comportamientos
      - Comprobamos que se han realizado las interacciones

Ejemplo Test Mock.

```
@Test
public void test() {
    ClaseB cb = new ClaseB(); //SUT
    ClaseA ca = Mockito.mock(ClaseA.class);
    cb.setCa(ca);

    Mockito.when(ca.func3(Mockito.anyInt(),
    Mockito.anyInt())).thenReturn(Integer.valueOf(5));

    cb.met1();
    int result=ca.func3(5, 6);

    assertEquals(5, result);
    Mockito.verify(ca, Mockito.times(2)).func3(Mockito.anyInt(),
    Mockito.anyInt());
}
```

Ejemplo con Eclipse

- Crear un proyecto java (Calculadora)
- En un “Source Folder” code
  - Crear una clase Interface => ItfAritmetica
    - Definir métodos suma y resta con enteros.
  - Crear stub de una clase que implemente la Interface
    - A esta clase me referiré como Arit
- En un “Source Folder” test
  - Crear una JUnit Test Class (New/ JUnit Test Class)
    - Seleccionar Arit como clase bajo test.
    - Seleccionar el método suma enteros.
  - Ejecutar el Test (Run As/JUnit Test)
    - Construir un test verifique suma enteros.

Ejemplo con Eclipse

- En el “Source Folder” test
  - Crear otra Junit Test Class (TestExcepciones)
    - Añadirle un test que compruebe una excepción.
  - Crear otra Junit Test Class (TestOperaciones)
    - Añadir varios test con sendas operaciones distintas.
    - Incluir una verificación de rendimiento.
  - Ejecutar todos los test en bloque organizados con clases anidadas.

Ejemplo con Eclipse

- En el “Source Folder” test
  - Forzar que se ignoren los test fallidos.
  - Crear y ejecutar un test parametrizado.
- En el “Source Folder” de code
  - Codificar resta(int a, int b) en la clase Arit
- Añadir otra clase Test con las pruebas para verificar la resta. Utilizar Fixtures

Ejemplo con Eclipse

- Nuevos métodos de Calculadora
  - Crear una clase Geo
    - Implementarle el método prod(int a, int b) que multiplique
      - El método no puede usar el operador '*'.
      - El método se implementa con la función suma de la clase Arit
- Nuevos Test
  - Generar un test válido para prod(int a, int b).
  - Cambiar de sitio
  - Sin ver el código tratar de crear una prueba que falle.

Ejemplo con Eclipse

- Nuevos métodos de Calculadora
  - En la clase Geo
    - Añadirle el método div(int a, int b)
      - El método no puede usar el operador '/'.
      - El método se implementa con la función resta de la clase aritmética
- Nuevos Test
  - Generar un test válido para div(int a, int b)
  - Cambiar de sitio
  - Sin ver el código tratar de crear una prueba que falle.

Ejemplo con Eclipse

- Crear nuevas funciones a Calculadora
  - Añadir una clase MemoriFunc que realice funciones con memoria con la calculadora. La memoria es un solo número.
    - mS: Guarda el resultado en memoria
    - mC: Borra memoria
    - mA: Suma el resultado al contenido de la memoria
    - mR: Muestra memoria
- Crear pruebas que incluyan todas las posibilidades de assertXXX

Ejemplo con Eclipse

- Utilizar la clase Geo como SUT.
- Ejecutar las pruebas de forma que estas pasen
- Crear Mocks para la clase Arit y configurarlas de forma que se sigan pasando las pruebas .

Ejemplo Injectar Mock.

```
// Clase aritmetica que vamos a Mockear
Arit ca;

// Clase geométrica en la que se Injectará la aritmética
@InjectMocks
Geo cg;

@Before
public void setUp() throws Exception {
// Mockeamos la clase para injectar
ca = Mockito.mock(Arit.class);

// Inicializa los mocks anotados con su clase inyectadas.
MockitoAnnotations.initMocks(this);
}
```

Ejemplo Injectar Mock.

```
@Test
public void testMult() {
   // Generamos los valores que debe devolver la clase
   mockeada
   Mockito.when(ca.suma(Mockito.anyInt(),Mockito.anyInt()))
        .thenReturn(Integer.valueOf(5))
        .thenReturn(Integer.valueOf(10))
        .thenReturn(Integer.valueOf(15));

   int result = cg.mult(3, 5);
   assertEquals("Producto Incorrecto", 15, result);
   Mockito.verify(ca,
   Mockito.times(3)).suma(Mockito.anyInt(),
   Mockito.anyInt());
}
```