

---

# **Práctica 1. Composición y jUnit**

Metodología y Desarrollo de Programas

---

Para la elaboración de esta presentación se ha utilizado algunos de los ejemplos expuestos en:

- ❖ Blog de <http://arodm.blogspot.com/search/label/Dise%C3%B1o> (Blog
- ❖ Apuntes de la asignatura de ingeniería del software de Mari Carmen Otero Vidal de la Universidad del País Vasco-Euskal Herriko Unibertsitatea  
<http://www.vc.ehu.es/jiwotvim/ISOFT2010-2011/ingSoftware1011.htm>

- 2.1. Relaciones entre clases
- 2.2. Asociación, composición y agregación
- 2.3 jUnit. Batería de test en Java

## 2.1. Introducción a las relaciones en OO

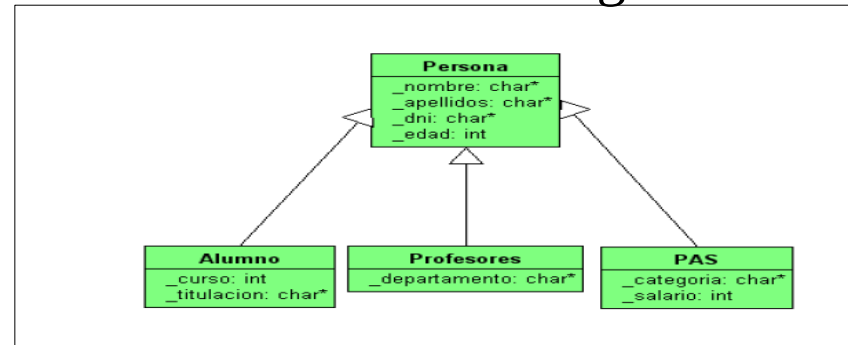
### ➤ Relaciones

#### ❖ Dependencia

- ✓ se representa mediante una línea discontinua. Utilizada fundamentalmente para la relación entre paquetes. P.ej: Clase Persona tiene un atributo String nombre

#### ❖ Herencia

- ✓ donde un objeto adquiere las propiedades y métodos de un objeto padre. Se representa mediante un triángulo



## 2.1. Introducción a las relaciones en OO

### ➤ Asociación, Composición y Agregación

- ❖ Los atributos de una clase se crean a partir de clases ya existentes que actúan como atributos de la nueva clase
- ❖ Esta nueva característica permite la creación de código a partir de uno ya existente --> **Reutilización**
- ❖ Estas relaciones generalmente se representa con la frase “*tiene un*”
- ❖ La diferencia entre ellas es una cuestión semántica. La implementación a nivel de código son similares

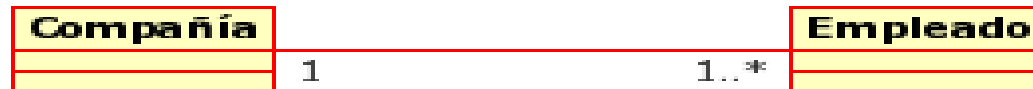
```
class Linea {  
    private Punto _origen;  
    private Punto _destino;  
    .....
```

¿Es asociación?  
¿Es composición?  
¿Es agregación?

## 2.1. Introducción a las relaciones en OO

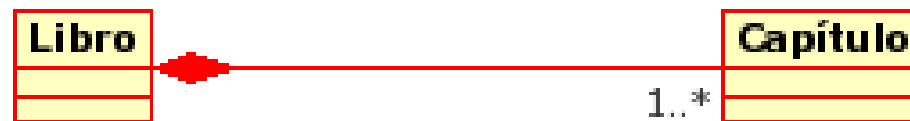
### ➤ Asociación, composición y agregación: **relación TIENE**

- ❖ Asociación: Representa una relación entre clases que colaboran para llevar algo a cabo.



### ❖ Composición

- ✓ Dos objetos están estrictamente limitados por una relación complementaria. Uno no se entiende sin el otro, esto es, cada uno por separado no tiene sentido. El tiempo de vida es dependiente.



### ❖ Agregación

- ✓ Dos objetos se consideran usualmente como independientes y aun así están ligados. Se puede decir que es de tipo todo / parte. El tiempo de vida de cada uno de los objetos es independiente.



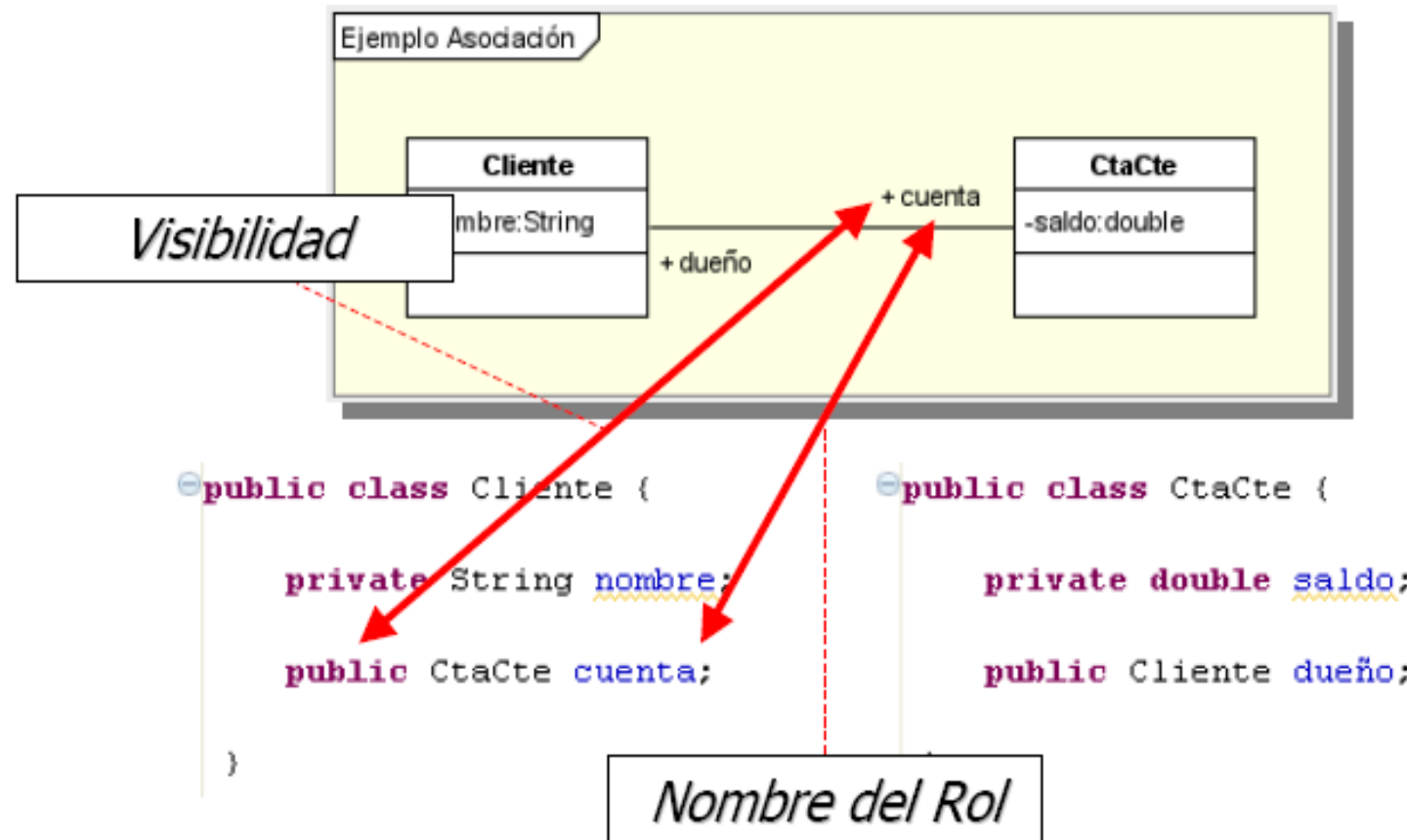
## 2.2. Asociación, composición y agregación

---

- Asociación:
  - ❖ Se define como una “relación semántica entre dos o más clases que especifica conexiones entre las instancias de estas clases”.
  - ❖ Existen varias opciones de asociación dependiendo de:
    - ✓ multiplicidad
      - 0..1 o 1
      - 1..N
    - ✓ Direccionales
      - Unidireccionales
      - Bidireccionales

## 2.2. Asociación, composición y agregación

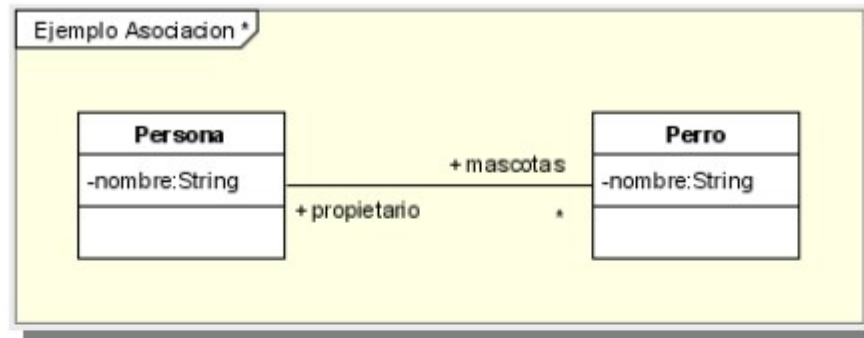
### ➤ Asociación Bidireccional con multiplicidad 0..1 o 1





## 2.2. Asociación, composición y agregación

### ➤ Asociación Bidireccional con multiplicidad \*

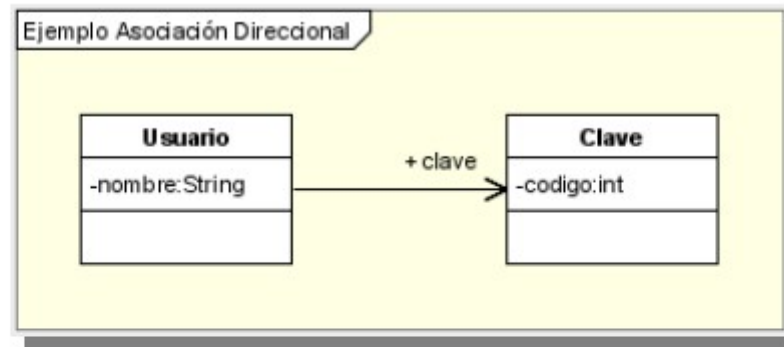


```
public class Persona {  
    private String nombre;  
    public java.util.Collection mascotas = new java.util.TreeSet();  
}  
  
public class Perro {  
    private String nombre;  
    public Persona propietario;  
}
```

*Decisión de Implementación*

## 2.2. Asociación, composición y agregación

### ➤ Asociación Direccional con multiplicidad 0..1 o 1

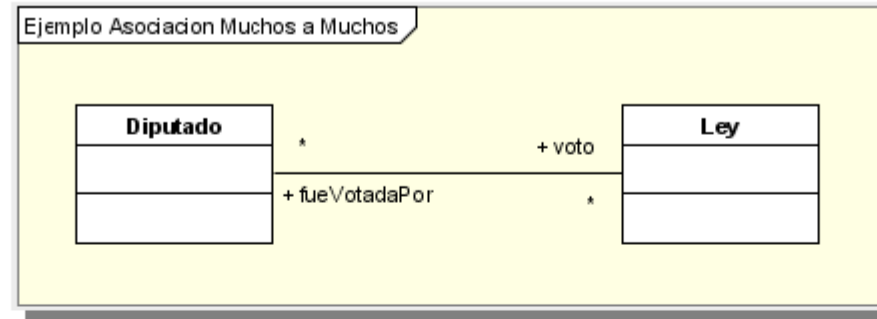


```
public class Usuario {  
    private String nombre;  
    public Clave clave;  
}
```

```
public class Clave {  
    private int codigo;  
}
```

## 2.2. Asociación, composición y agregación

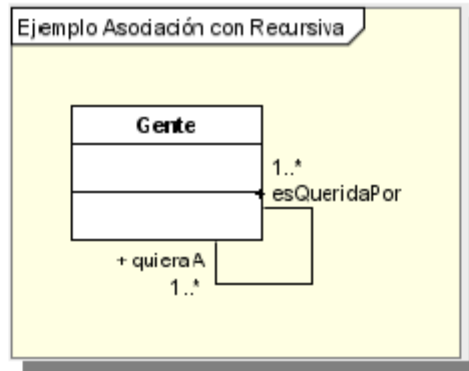
### ➤ Asociación Bidireccional con multiplicidad \*



```
public class Diputado {  
    public java.util.Collection voto = new java.util.TreeSet();  
}  
  
public class Ley {  
    public java.util.Collection fueVotadaPor = new java.util.TreeSet();  
}
```

## 2.2. Asociación, composición y agregación

### ➤ Asociación

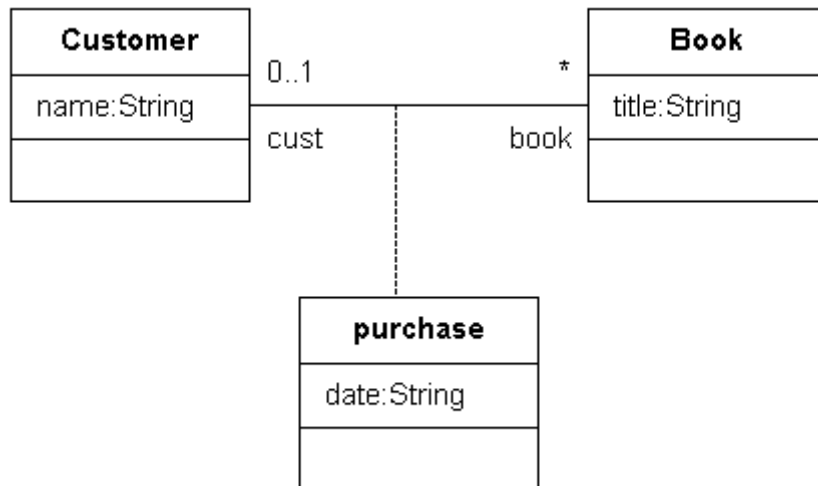


```
public class Gente {  
  
    public java.util.Collection quieriaA = new java.util.TreeSet();  
  
    public java.util.Collection esQueridaPor = new java.util.TreeSet();  
  
}
```

## 2.2. Asociación, composición y agregación

### ➤ Asociación ternaria:

- ❖ Se usa clase de asociación para representar la compra (purchase) de un libro (Book) por un comprador (Customer).



```
public class Book {
    // Data attributes
    private String title;
    // Association attributes
    public purchase purchaseOfCust;
    .....
} // class Book

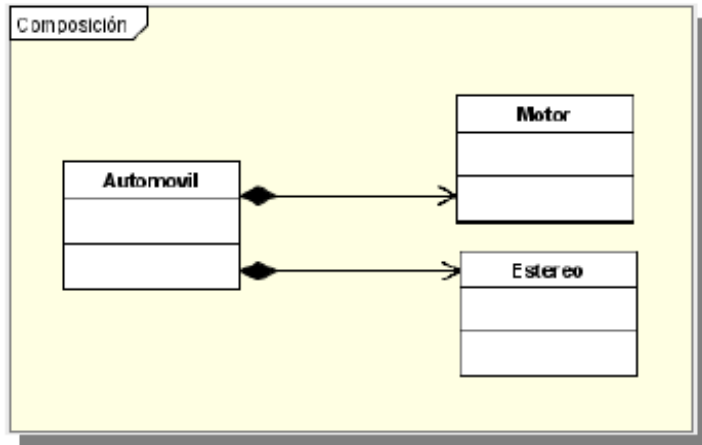
public class Customer {
    // Data attributes
    private String name;
    // Association attributes
    public Collection purchaseOfBookSet;

    .....
} // class Customer

public class purchase {
    // Data attributes
    private String date;
    // Association attributes
    public Customer cust;
    public Book book;
    ....
} // class purchase
```

## 2.2. Asociación, composición y agregación

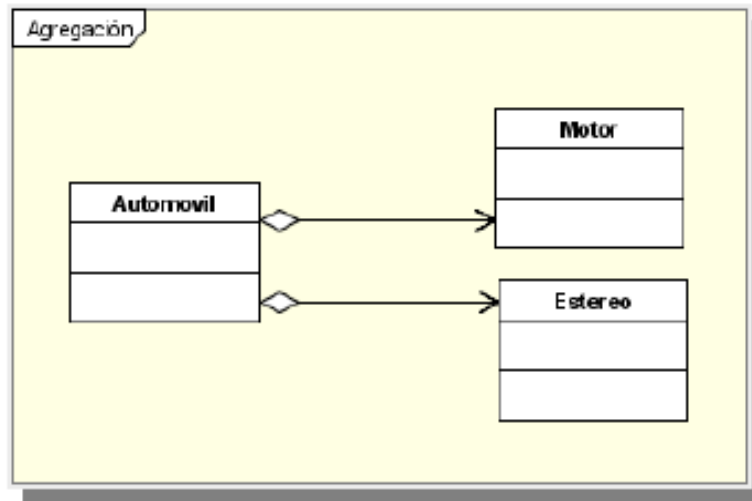
### ➤ Composición



```
public class Automovil {  
  
    public Estereo estereo;  
    public Motor motor;  
  
    public Automovil() {  
        estereo = new Estereo();  
        motor = new Motor();  
    }  
}
```

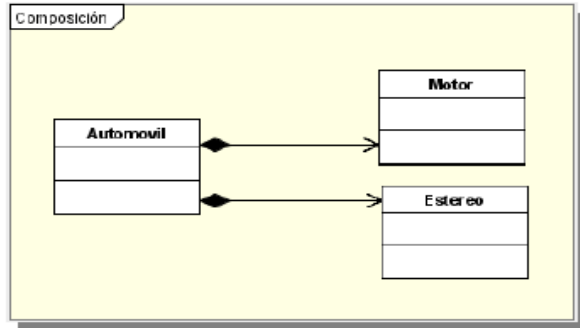
## 2.2. Asociación, composición y agregación

### ➤ Agregación

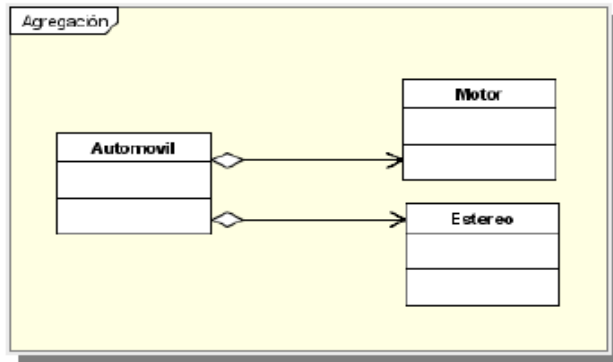


```
public class Automovil {  
  
    public Estereo estereo;  
    public Motor motor;  
  
    public Automovil() {  
        estereo = new Estereo();  
        motor = new Motor();  
    }  
}
```

## 2.2. Asociación, composición y agregación



```
public class Automovil {  
  
    public Estereo estereo;  
    public Motor motor;  
  
    public Automovil() {  
        estereo = new Estereo();  
        motor = new Motor();  
    }  
}
```



```
public class Automovil {  
  
    public Estereo estereo;  
    public Motor motor;  
  
    public Automovil() {  
        estereo = new Estereo();  
        motor = new Motor();  
    }  
}
```

¿Cuál es la diferencia entre composición y agregación?



## 2.2. Asociación, composición y agregación

### ➤ Asociación

- ❖ Se define como una “relación semántica entre dos o más clases que especifica conexiones entre las instancias de estas clases”.

### ➤ Composición

- ❖ Dos objetos están estrictamente limitados por una relación complementaria. Uno no se entiende sin el otro, esto es, cada uno por separado no tiene sentido. El tiempo de vida es dependiente.

### ➤ Agregación

- ❖ Dos objetos se consideran usualmente como independientes y aun así están ligados. Se puede decir que es de tipo todo / parte. El tiempo de vida de cada uno de los objetos es independiente.

¿Cuál es la diferencia entre asociación  
composición y agregación a nivel de código?

## 2.2. Asociación, composición y agregación

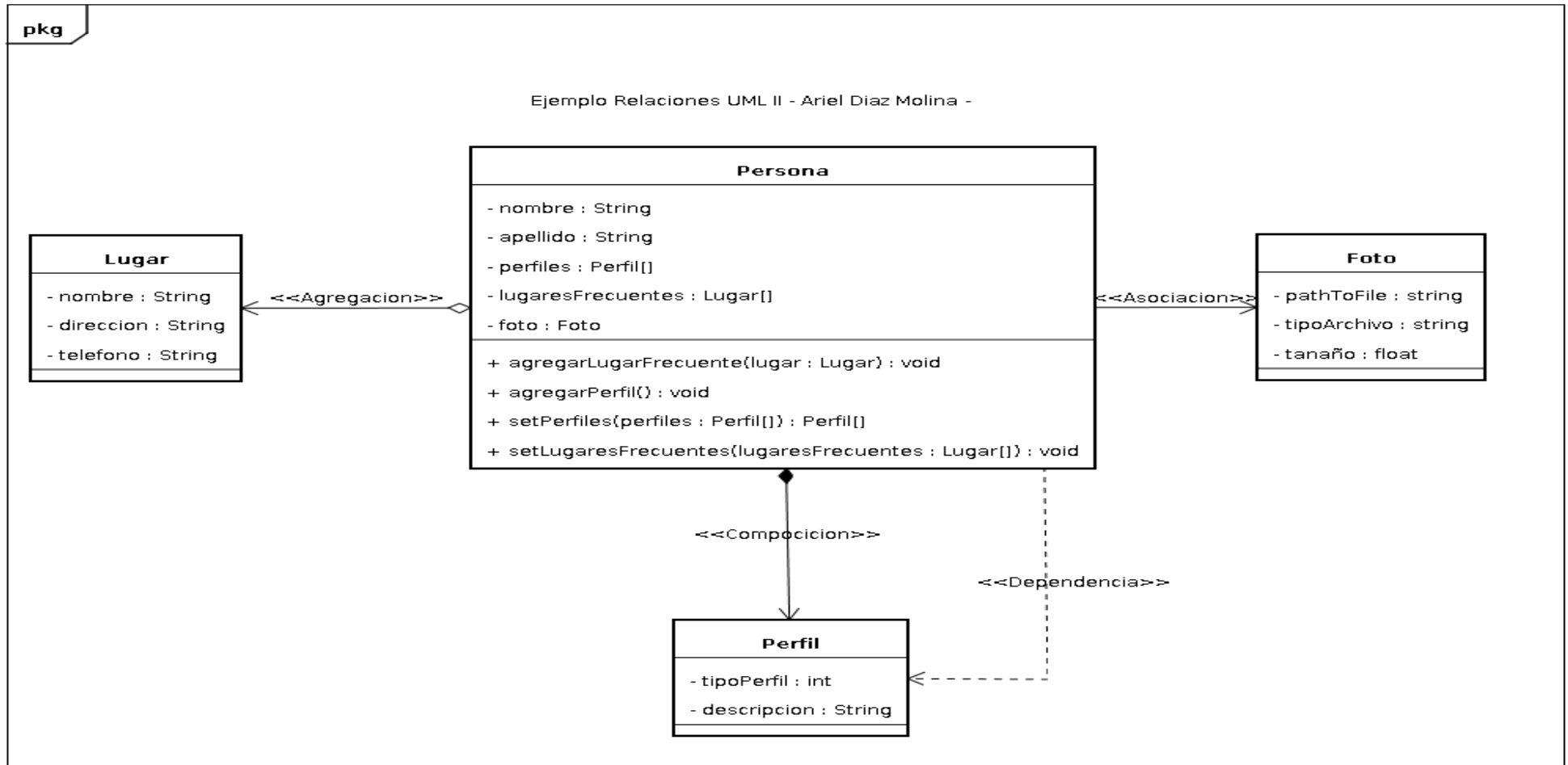
### ➤ Diferencias

- ❖ La relación de **asociación** se suele usar cuando únicamente se tiene **un objeto de otra clase**, que se recibe en el constructor o con el método set.
- ❖ La **agregación** puede, como no, tener los métodos **setter y getter**, mientras que la **Asociación siempre los tiene**, que ponen y obtienen una variable de referencia del mismo tipo de la variable de instancia o de clase.
- ❖ Agregación vs composición: contiene dos métodos uno que **agrega** elementos a la colección y otro que los **elimina** de ella.
  - ✓ **Agregación:** los objetos son pasados por parametro, no han sido instanciados dentro del método, es decir no hemos realizado el new del objeto. **Existen cada uno de forma independiente.**
  - ✓ **Composición:**
    - "el new del objeto se realiza dentro del método agregar"
    - De este modo, el objeto compuesto debe dejar de existir una vez que el método se termine de ejecutar, y el ciclo de vida de esa variable va a existir mientras el objeto exista (debería implementarse el método finalize()).

## 2.2. Asociación, composición y agregación

➤ Ejemplos de este apartado obtenido de:

<http://arodm.blogspot.com/search/label/Dise%C3%B1o>



## 2.2. Asociación, composición y agregación

```
public class Persona {  
  
    private String nombre;  
    private String apellido;  
    private List perfiles = new LinkedList();  
    private List lugaresFrecuentes = new LinkedList();  
  
    //Setters and Getters  
    public String getNombre()  
        {return nombre;}  
    public void setNombre(String nombre)  
        {this.nombre = nombre;}  
    public String getApellido()  
        {return apellido;}  
    public void setApellido(String apellido)  
        {this.apellido = apellido;}  
  
    // OJO. Estos son solo setters y getters de las propiedades  
    public List getPerfiles() {  
        return perfiles;  
    }  
    public void setPerfiles(List perfiles) {  
        this.perfiles = perfiles;  
    }  
    public List getLugaresFrecuentes() {  
        return lugaresFrecuentes;  
    }  
    public void setLugaresFrecuentes(List lugaresFrecuentes) {  
        this.lugaresFrecuentes = lugaresFrecuentes;  
    }  
}
```

## 2.2. Asociación, composición y agregación

```
//Agregación: Añadir y eliminar
public void agregarLugarFrecuenta(Lugar lugar) {
    if (!lugaresFrecuentes.contains(lugar)) {
        lugaresFrecuentes.add(lugar);
    }
}

public void removerLugarFrecuenta(Lugar lugar) {
    if (lugaresFrecuentes.contains(lugar)) {
        lugaresFrecuentes.remove(lugar);
    }
}

//Composición: Añadir y eliminar
public void agregarPerfil() {
    Perfil p = new Perfil();
    perfiles.add(p);
}

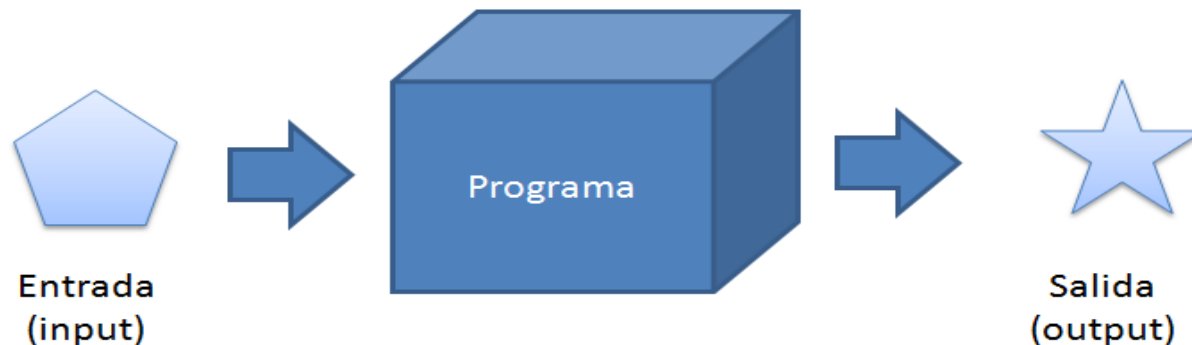
//sobrecarga
public void agregarPerfil(String nombre) {
    Perfil p = new Perfil(nombre);
    perfiles.add(p);
}

public void removerPerfil(int index) {
    perfiles.remove(index);
}
```

```
}
```

# Introducción a junit

- El objetivo de la práctica 0 fue repasar el ciclo de vida un objeto con su instanciación así como el desarrollo de un programa con array de objetos.
- Se realizó una entrega usando un evaluador automático que comprobaba si la ejecución del programa era correcta.
- Pero ¿cómo funciona?
  - ❖ La ejecución recibe unos datos de entrada y con ellos espera unos datos de salida
  - ❖ Si la salida obtenida no es igual a la esperada → Programa funciona incorrectamente

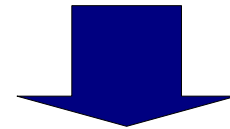


- Java dispone de una herramienta propia para automatizar estas pruebas → <http://junit.org/junit4/>
- Open Source, disponible en <http://www.junit.org>
- Adecuado para el Desarrollo dirigido por las pruebas (*Test-driven development*)
- Consta de un conjunto de clases que el programador puede utilizar para construir sus casos de prueba y ejecutarlos automáticamente.
- Los casos de prueba son **realmente programas Java**.  
Quedan archivados y pueden ser reejecutados tantas veces como sea necesario

- ¿Cómo pruebo manualmente el método equals de Persona?
  - ❖ Poniendo un caso de uso correcto e incorrecto que verifiquen su funcionamiento → **Importante:** Automático y repetible
  - ❖ Mediante el uso de instrucciones **if**.

```
//Prueba1
Persona p1= new Persona("Paco","1223",2);
Persona p2= new Persona("Paco","1223",2);
boolean expected=true;
boolean result=p1.equals(P2));
if (expected==result)
    System.out.println("OK");
else
    System.out.println("Fallo");
```

- Muy artesano
- Arduo
- Salida



jUnit



## El framework JUnit

---

- El ejemplo anterior (*result* frente a *expected*) es una idea fundamental de JUnit
  - ❖ Siempre se va a establecer el resultado y se va a comparar con el resultado obtenido
- JUnit:
  - ❖ Permite separar los casos de prueba frente al código fuente
  - ❖ Permite ejecutarlos automáticamente
  - ❖ Nos va a permitir programar conjuntos de test (*suites*)

- JUnit tiene un conjunto de métodos para comprobar lo esperado con lo obtenido

```
public class Tests extends TestCase {  
    public Tests () { }  
    public void testSumas () {  
        assertEquals(4, 2+2);  
    }  
    public void testDivisiones () {  
        assertTrue(1==2/2);  
    }  
}
```

## El framework JUnit

- JUnit incluye una serie de métodos para probar que las cosas son como esperamos.

<code>assertEquals (X esperado, X obtenido)</code>	compara un resultado esperado con el resultado obtenido, determinando que la prueba pasa si son iguales, y que la prueba falla si son diferentes.
<code>assertFalse (boolean resultado)</code>	verifica que el resultado es FALSE
<code>assertTrue (boolean resultado)</code>	verifica que el resultado es TRUE
<code>assertNull (Object resultado)</code>	verifica que el resultado es "null"
<code>assertNotNull (Object resultado)</code>	verifica que el resultado no es "null" fail sirve para detectar que estamos en un sitio del programa donde NO deberíamos estar

## Ejemplo con JUnit

---

- Crear un nuevo proyecto “Práctica 01”
- Crear un paquete “es.unex.cum.mdp.practica01”
- Crear una clase Calculator sin atributos e implementar
  - ❖ add(double a, double b)
  - ❖ subtract(double a, double b)
  - ❖ multiply(double a, double b)
  - ❖ divide(double a, double b)
- Para finalizar es necesario incluir la librería jUnit en el proyecto. **¿Cómo se añade una librería?**
  - ❖ Botón Derecho → Properties → BuildPath → Libraries → AddLibrary → jUnit → jUnit 4.

# Ejemplo con JUnit

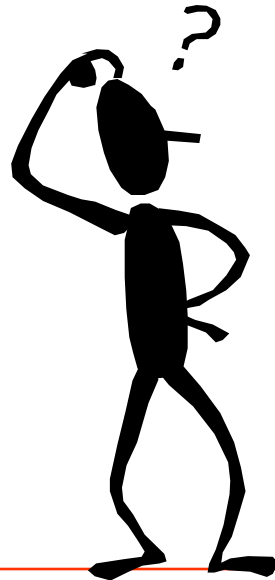
## ➤ Ejemplo 1. Calculadora

❖ Dada la siguiente clase, implementar la batería de prueba

```
public class Calculator {  
  
    public double add(double a, double b) {  
        return a + b;  
    }  
  
    public double subtract(double a, double b) {  
        return a - b;  
    }  
  
    public double multiply(double a, double b) {  
        return a * b;  
    }  
  
    public double divide(double a, double b) {  
        if (b == 0) {  
            throw new ArithmeticException();  
        }  
        return a / b;  
    }  
}
```

## Ejemplo con JUnit

- Entonces ahora por cada uno de los métodos, debemos **programar** una ejecución para comprobar si el código funciona correctamente → debe generar siempre una misma salida
  - ❖ ¿Cómo sabemos que un constructor funciona correctamente? ¿Qué hace un constructor? ¿Si no hay constructor se programa la batería)
  - ❖ ¿Cómo sabemos que un método getter funciona correctamente?
  - ❖ ¿Cómo sabemos que un método equals funciona correctamente?
  - ❖ ...



## Ejemplo con JUnit

- ¿Cómo podría saber si la operación de la suma funciona correctamente?
  - ❖ Dándoles unos valores de entrada y comprobado que el resultado es el esperado

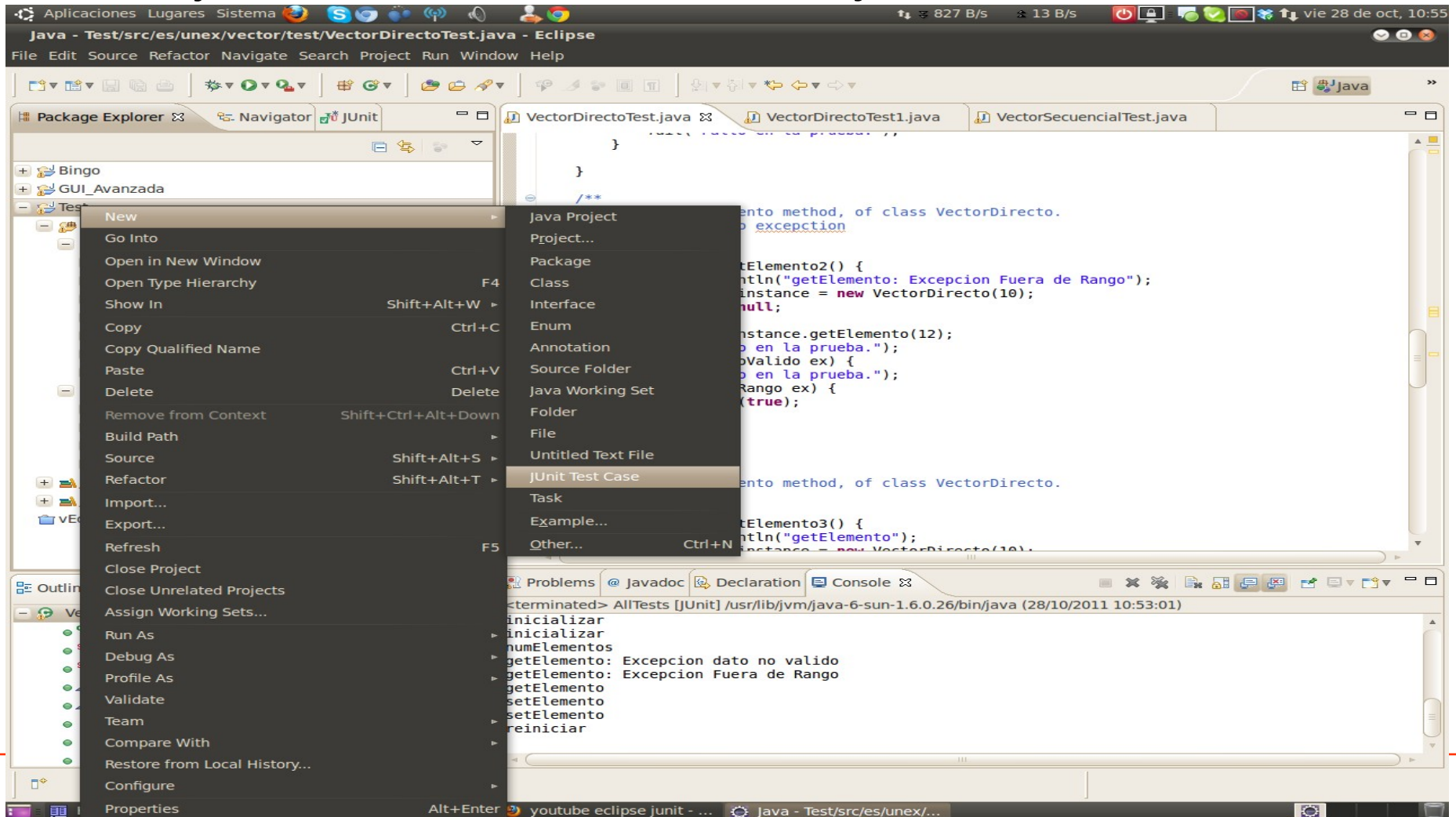
```
public class TestCalculator {  
  
    @Test  
    public void testAdd() {  
        Calculator calc= new Calculator(); //Se instancia el objeto  
        double result = calc.add(10.5, 20.32); //Se hace la op.  
        assertEquals(30.82, result,0); //Se comprueba el resultado  
        //assertTrue(30.82==result); //Otra forma  
    }  
    ...  
    ...  
}
```

- ❖ ¿Qué métodos tengo que programar? ¿Con que nombre? ¿Qué devuelven?...

# Ejemplo con JUnit

➤ Eclipse facilita el trabajo con jUnit → genera el esqueleto y sólo es necesario rellenarlo correctamente

❖ Proyecto → Botón Derecho → new → JUnit Test Case

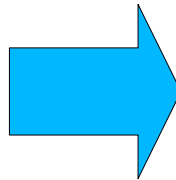




## Ejemplo con JUnit

- Por cada método de la clase ha generado un método TestXXXX con su esqueleto. Cuidado:
  - ❖ Por defecto devuelve fail: hay que quitarlo pues la ejecución de la batería daría un resultado negativo
  - ❖ Hay que adaptar el código a la funcionalidad propia del método.

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



```
@Test
public void testSubtract() {
    Calculator calc = new Calculator();
    double result = calc.subtract(100.5, 100);
    assertEquals(0.5, result);
}
```

Generado

Correcto

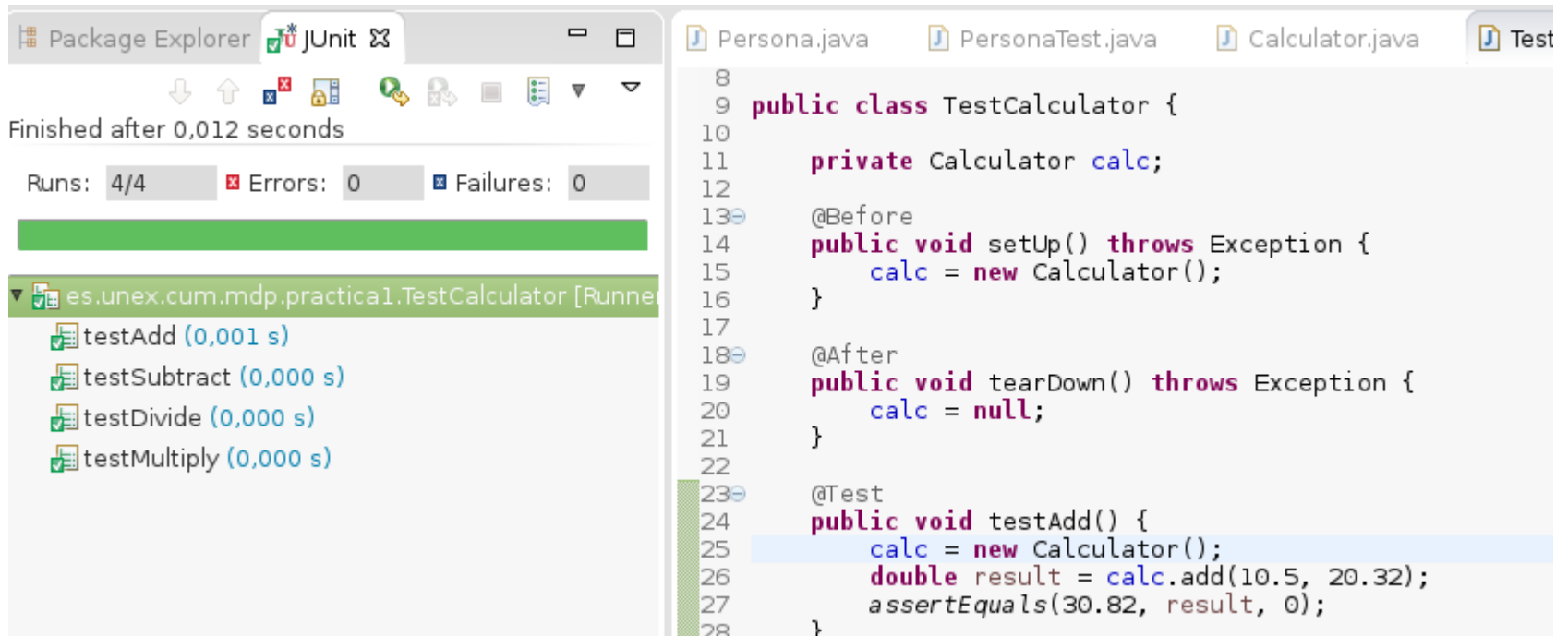
# Ejemplo con JUnit

➤ Para probar los test → Ejecutar el fichero de Test

```
public class TestCalculator {  
    private Calculator calc;  
    @Before  
    public void setUp() throws Exception { //Lo ejecuta antes de cada test  
        calc = new Calculator();  
    }  
    @After  
    public void tearDown() throws Exception { //Lo ejecuta después de cada test  
        calc = null;  
    }  
    @Test  
    public void testMultiply() {  
        double result = calc.multiply(2.5, 100);  
        assertEquals(250, result);  
    }  
    @Test  
    public void testDivide() {  
        double result = calc.divide(100, 10);  
        assertEquals(10, result);  
        try {  
            calc.divide(100.5, 0);  
            fail();  
        } catch (ArithmeticException e) {  
            assertTrue(true);  
        }  
    }  
}
```

# Ejemplo con JUnit

➤ Para probar los test → Ejecutar el fichero de Test



The screenshot displays an IDE interface. On the left, the 'JUnit' tab shows the test results for 'es.unex.cum.mdp.practical1.TestCalculator'. The tests passed successfully, with a total time of 0.012 seconds. The results list includes: testAdd (0,001 s), testSubtract (0,000 s), testDivide (0,000 s), and testMultiply (0,000 s). On the right, the 'Test' tab shows the source code of 'TestCalculator.java'. The code defines a 'TestCalculator' class with a private 'Calculator' field 'calc'. It includes '@Before' and '@After' methods for setup and teardown, and a '@Test' method 'testAdd()' that creates a 'Calculator' instance, calls 'add(10.5, 20.32)', and asserts the result is 30.82.

```
8
9 public class TestCalculator {
10
11     private Calculator calc;
12
13     @Before
14     public void setUp() throws Exception {
15         calc = new Calculator();
16     }
17
18     @After
19     public void tearDown() throws Exception {
20         calc = null;
21     }
22
23     @Test
24     public void testAdd() {
25         calc = new Calculator();
26         double result = calc.add(10.5, 20.32);
27         assertEquals(30.82, result, 0);
28     }
```

## Ejemplo2 con JUnit

---

- Crear una clase Pieza con los siguientes atributos:
  - ❖ `private String id;`
  - ❖ `private String nombre;`
  - ❖ `private int stock;`
  - ❖ Implementar los métodos `CD`, `CP`, `Getter`, `Setter`, `equals`, `toString`. Se puede utilizar el generador automático de código.
  
- ¿Qué métodos deben probarse con jUnit? → **TODOS**

## Ejemplo2 con JUnit

- ¿Cómo se realiza la batería entonces? → Se define una nueva clase Java por cada clase implementada y para cada método se debe realizar un test

```
public class TestPieza {  
  
    @Test  
    public void testConstructorPorDefecto() {  
        Pieza p = new Pieza();  
        assertNotNull(p);  
    }  
  
    @Test  
    public void testConstructorParametrizado() {  
        Pieza p = new Pieza("1", "Freno", 2);  
        assertNotNull(p);  
    }  
  
    @Test  
    public void testConstructorCopia() {  
        Pieza p = new Pieza("1", "Freno", 2);  
        Pieza p2 = new Pieza(p);  
        assertNotNull(p2);  
        assertEquals(p, p2);  
    }  
}
```

# Ejemplo con JUnit

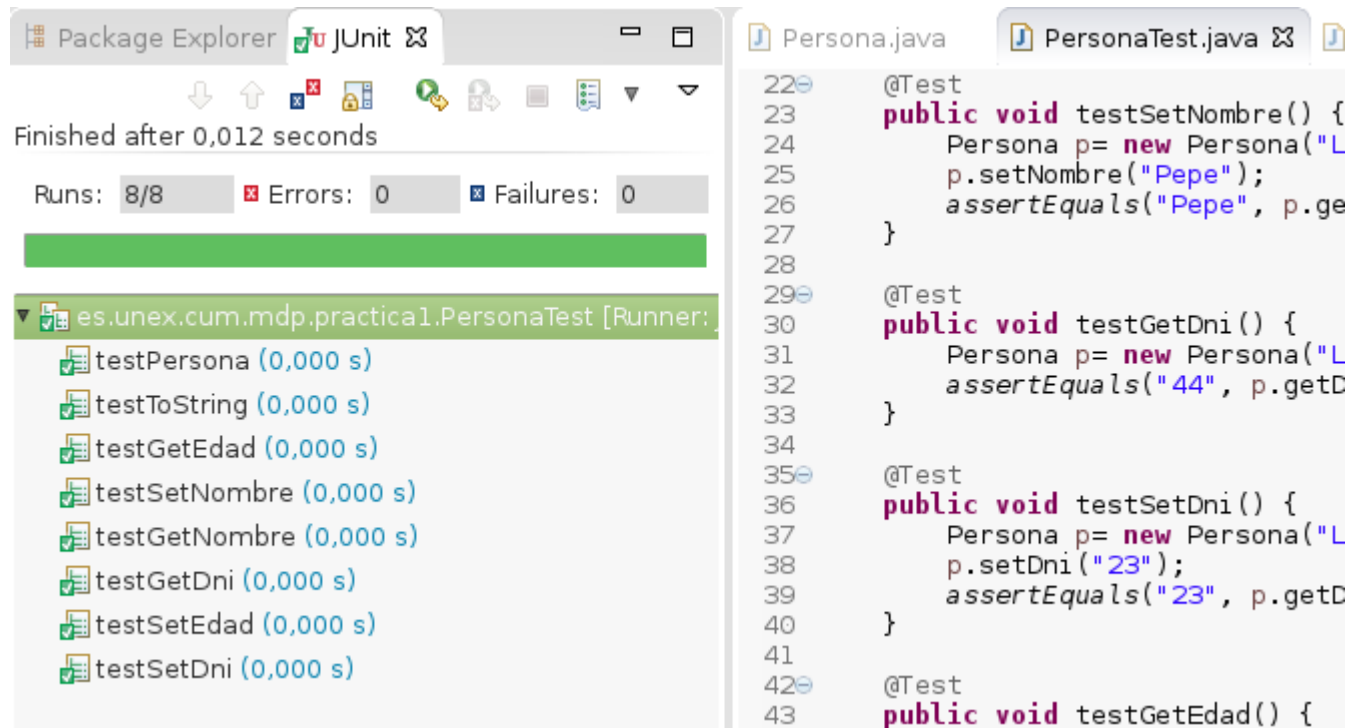
## ➤ Ejemplo de Pieza

```
public class TestPieza {
    private Pieza p1, p2, p3;
    @Before
    public void setUp() throws Exception {
        p1 = new Pieza();
        p2 = new Pieza("1", "Freno", 2);
        p3 = new Pieza(p2);
    }
    @After
    public void tearDown() throws Exception {
        p1=null;
        p2=null;
        p3=null;
    }
    @Test
    public void testPieza() {
        assertNotNull(p1);
        assertNotNull(p2);
        assertNotNull(p3);
        assertEquals(p2, p3);
    }
    @Test
    public void testGetNombre() {
        String nombre = p2.getNombre();
        assertEquals("Freno", nombre);
    }
    @Test
    public void testSetNombre() {
        p2.setNombre("Motor");
        assertEquals("Motor", p2.getNombre());
    }
}
```

```
@Test
public void testGetId() {
    String Id = p2.getId();
    assertEquals("1", Id);
}
@Test
public void testSetId() {
    p2.setId("1234");
    assertEquals("1234", p2.getId());
}
Contador @Test
public void testGetStock() {
    int Contador = p2.getContador();
    assertEquals(2, Contador);
}
@Test
public void testSetStock() {
    p2.setContador(35);
    assertEquals(35, p2.getContador());
}
@Test
public void testToString() {
    assertEquals(p2.toString(), "Pieza [id=1,
nombre=Freno, contador=2]");
}
@Test
public void testEquals() {
    assertEquals(p2, p3);
    assertNotEquals(p1, p3);
}
}
```

# Ejemplo con JUnit

➤ Para probar los test → Ejecutar el fichero de Test



The screenshot displays an IDE interface with two main panels. The left panel shows the 'JUnit' runner window, indicating that the tests were 'Finished after 0,012 seconds'. It reports 'Runs: 8/8', 'Errors: 0', and 'Failures: 0'. Below this, a list of test methods is shown, all marked with green checkmarks and a duration of '(0,000 s)':

- testPersona (0,000 s)
- testToString (0,000 s)
- testGetEdad (0,000 s)
- testSetNombre (0,000 s)
- testGetNombre (0,000 s)
- testGetDni (0,000 s)
- testSetEdad (0,000 s)
- testSetDni (0,000 s)

The right panel shows the source code for 'PersonaTest.java'. The code includes several test methods annotated with '@Test':

```
22 @Test
23 public void testSetNombre() {
24     Persona p= new Persona("L
25     p.setNombre("Pepe");
26     assertEquals("Pepe", p.ge
27 }
28
29 @Test
30 public void testGetDni() {
31     Persona p= new Persona("L
32     assertEquals("44", p.getD
33 }
34
35 @Test
36 public void testSetDni() {
37     Persona p= new Persona("L
38     p.setDni("23");
39     assertEquals("23", p.getD
40 }
41
42 @Test
43 public void testGetEdad() {
```

## Ejercicio a realizar

- Implementar la clase Vehículo con las siguientes características:
  - ❖ marca (String)
  - ❖ modelo (String)
  - ❖ propietario (Persona): Composición unaria (getter y setter)
  - ❖ piezas (Array de Piezas): Composición nAria (addPiezaV, getPiezaV)
  - ❖ cont (para usar un vector más contador)
  - ❖ Debe existir un método que permita añadir piezas a un vehículo public void addPiezaV(Pieza p). Se debe controlar los posibles errores.
  - ❖ Debe existir un método que permita obtener una pieza: public Pieza getPiezaV(int pos). Se debe controlar los posibles errores.
- Importante: Debe cumplir con la batería de test proporcionada

