

## **INTRODUCCIÓN**

- Orientación a objetos
- Análisis de requisitos. Enunciado ampliado y Casos de Uso
- Especificación. Modelo conceptual
- Diseño de la Arquitectura en 3 capas
- Implementación en Java
- Documentación
- Prueba de Programas
- Diseño e implementación de interfícies

$$NF = 0'8 * \text{NotaProyecto} + 0'2 * \text{NotaExamen}$$

$$\text{NotaProyecto} = 0'1 * \text{entrega1} + 0'2 * \text{entrega2} + 0'6 * \text{entrega3}$$

## **CICLO DE VIDA CLÁSICO DE UNA APLICACIÓN INFORMÁTICA**

Análisis de Requisitos (identificar QUÉ tiene que hacer el sistema)

└ Especificación (describir QUÉ tiene que hacer el sistema)

└ Diseño (describir CÓMO tiene que hacerlo el sistema)

└ Implementación (implementarlo)

└ Pruebas

└ Mantenimiento



## **ORIENTACIÓN A OBJETOS**

- Escala muy bien
- Recuerda a la manera de resolver problemas que usamos los humanos
- Multitud de librerías

ejemplo FLORISTERÍA

## ELEMENTOS DE LA ORIENTACION A OBJETOS

- Todo es un objeto
- Cada objeto es una instancia de una clase
- Una clase es una agrupación de objetos similares: es el repositorio del comportamiento y responsabilidad
  - Cada objeto tiene su propia memoria (datos)
  - Todos los objetos de la misma clase pueden hacer las mismas acciones
  - Las clases se organizan en una estructura jerárquica (clases y subclases)
- La comunicación entre objetos es mediante mensajes

## METODOLOGÍA GENERAL DE LA ORIENTACION A OBJETOS

Objetivos:

- Facilitar la construcción de programas
- Facilitar el mantenimiento de programas

La orientación a objetos intenta 3 cosas para conseguirlo:

- 1.- encapsular la información y el comportamiento (ocultar la implementación):  
Que la información quede encapsulada dentro de las clases, de modo que cada clase tenga las atribuciones que le toca.
- 2.- simular la realidad
- 3.- reutilizar el código

-----CLASE EXTRA-----

## PILARES DE LA ORIENTACIÓN A OBJETOS

Encapsulamiento/Ocultación

Abstracción/Clasificación

Herencia/Polimorfismo

- Reutilización de código
- Encapsulamiento
- (simular la realidad)

Concepto clave: envío de mensajes entre objetos.

Atributos (estado interno)

Definición de objetos → **Clase** (un objeto siempre es instancia de una clase) <

→ basados en prototipos

comportamiento (métodos)

## ENCAPSULAMIENTO/OCULTACIÓN:

El estado es irrelevante para el que usa el objeto (cliente), es privado.

Ej:

```
public class Ejemplo {
    public int estado;

    public Ejemplo (int estado) {
        this.estado = (estado < 0)?:estado;
    }
    .
    .
    .
}
```

Ejemplo **e** = new Ejemplo(3);           → estado = 3  
e.estado = -42;           //porque lo hemos declarado como **public** → perdemos el encapsulamiento

Ej2:

```
public class Ejemplo {
    private int estado;

    public Ejemplo (int estado) {
        this.estado = (estado < 0)?:estado;
    }
    int public getEstado() { return this.estado}
    void public setEstado(int e) {
        this.estado = (e<0)?this.estado:e;
    }
    .
    .
    .
}
```

Ejemplo **e** = new Ejemplo(3);           → estado = 3  
e.setEstado(2);  
e.setEstado(-3);

Ej3:

```
d : Date
d := Date(2, 2, 1974);
intercambiarDate(d1, d2);
```

```
Date d = new Date(2, 2, 1974)
d1.intercambiarDate(d2);
```

## **HERENCIA:**

```
public class Persona {
    private string dni;
    private int edad;
    private double altura;
    .
    .
    .
    public Persona (...) {
        .
        .
        .
    }
}

public class Estudiante extends Persona { //HEREDA de Persona (métodos y atributos)
    private int numMatricula;
    .
    .
    .
}

public class EstudianteUniversitario extends Estudiante {
    private double notaBachillerato;
    .
    .
    .
}
```

## **en UML**

Persona

-----

dni:string

edad: int

altura: double

-----

Persona(...)

Estudiante

-----

numMatricula: int

-----

-----

EstudianteUniversitario

-----

notaBachillerato: double

-----

- Una clase C2 subclase de C1, hereda de C1 los atributos y los métodos (TODO).
- C2 puede sobrescribir métodos de C1.
- La herencia es transitiva (“ser subclase de”)
- Decimos que C2 es una subclase de C1, si EN CUALQUIER CONTEXTO en el que aparece un objeto instancia de C1, puedo utilizar un objeto instancia de C2.

```
Public class A {
    .
    .
    .
    public void m () {
        system.out.println("A");
    }
}
```

```
public class B extends A {
    .
    .
    .
    public void m() {
        system.out.println("B");
    }
}
```

```
A a;    //a es instancia de A
a = ...;
a.m();    → no podemos saber si se escribirá A o B.
```

En Java, tenemos herencia SIMPLE:  
tenemos una clase sin padre (clase Object), que tiene subclases → tenemos un árbol

C++ tiene herencia múltiple:  
una clase puede heredar de varias clases → tiene un grafo.

### ABSTRACCIÓN/CLASIFICACIÓN

Clase ABSTRACTA: es aquella con **algún** (o todos los) método abstracto. → **No se puede instanciar!**

Un método abstracto es aquel en el que no defines el código, sólo la cabecera.

*Numeric* → se pone en cursiva

-----  
*suma*(Numeric, Numeric): Numeric  
-----

Enteros                      Complejos                      Reales                      → para que dejen de ser abstractas, cada uno (subclase) ha de implementar la suma

```

        > vector de Trabajador
void calcularSueldosTotales (Trabajador[] t) {    //Trabajador siendo abstracta con método sueldo
    double sueldoTotal = 0;
    for (int i = 0; i < t.size(); ++i) {
        sueldoTotal += t[i].sueldo();
    }
    return sueldoTotal;
}

```

Trabajador **no** tendrá ninguna instancia de Trabajadores, sino de Administrativos, Ejecutivos, Comerciales, etc... porque todos ellos tienen un método “sueldo”.

### HERENCIA + ABSTRACCIÓN = POLIMORFISMO

(en el código no podemos preguntar la clase de un objeto, lo estaríamos haciendo MAL!) Es importante saber quién tiene la responsabilidad de qué.

```

Public class Nodo <T> {
    private T date;
    private Nodo <T> next;
    public Nodo (T date, Nodo<T> next) {
        this.date = date;
        this.next = next;
    }
    public T getDate() {return date}
}

```

### en UML

```

-----
Nodo    |    T    |
-----
date: T
next: Nodo<T>
-----
...

```

```

public class NaturalNumbers<T extends Integer> {
    ...
}

```

### en UML

```

-----
NaturalNumbers    |    Integer → T    |
-----
...

```

-----CLASE EXTRA-----

- Un mismo nombre hace referencia a entradas diferentes
- El mismo objeto puede tener varios tipos
- Se puede usar la misma interficie con objetos de diferentes tipos

Se puede aplicar a:

- Métodos
- Objetos
- Clases

Hay 3 tipos de polimorfismos:

Polimorfismo paramétrico (Objetos y Clases)

Polimorfismo ad-hoc (Método)

Polimorfismo de subtipo (Objetos)

**Polimorfismo paramétrico:** //parámetros

└> es un parámetro

```
class List <T> {                               //clase GENÉRICA
    ...
    T.getValue();
    add(elemT) {
        ...
    }
}
```

List<Persona> **lp** = new List<Persona> //lp es polimórfico

**Polimorfismo ad-hoc:**

Un mismo nombre denota métodos diferentes.

- Sobrecarga de operadores:

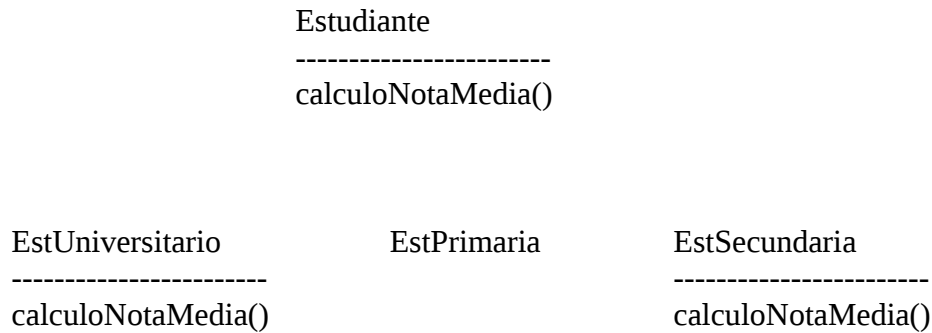
int a, b, c;

c = a + b; //c = suma(a, b);

- Sobrecarga de métodos en la misma clase:

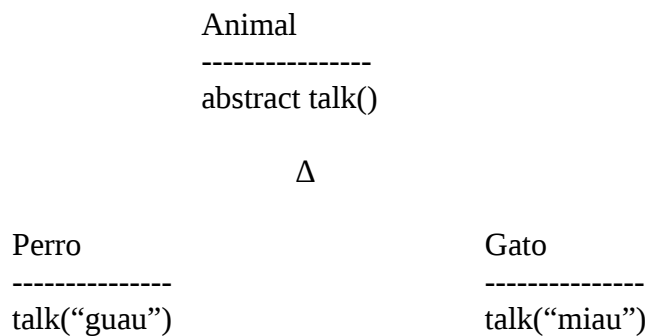
```
class MiClase ... {
    int n;
    float m;
    void incrementar (int dn) {
        n = n + dn;
    }
    void incrementar (float dm) {
        m = n + dm;
    }
    void incrementar (int dn, float dm) {
        incrementar(dn);
        incrementar(dm);
    }
}
```

- Redefinición de métodos en una jerarquía de clases:



### **Polimorfismo de subtipo:**

Con el mismo nombre de variable, se hace referencia a objetos de clases diferentes (en la misma jerarquía).



```
Animal a;
Gato g;
Perro p;

a = g;
a.talk();    //miau

a = p;
a.talk();    //guau

Gato g, Perro p;
Animal [] a = new Animal[2];
n[0] = g;
n[1] = p;
for (int i = 0; i < 2; ++i) a.talk();    //miau, guau
```

Donde aparezca un objeto de clase Animal se puede referenciar uin objeto de clase Perro/Gato, pero NO al revés:

```
a = p;    //✓
p = a;    //X
```



## REGLAS DE VISIBILIDAD

### Entre clases independientes:

- 1.- En una clase se puede ver todo lo que está definido en ella (atributos, métodos).
- 2.- Dentro de una clase se pueden ver cosas de otras clases independientes, pero hay que **importarlas**.
- 3.- Se puede cualificar cómo se ven las cosas (public, private, protected, package, ...).

### Dentro de una clase:

- 1.- Si se usa un identificador que se ha declarado varias veces, se usa el identificador declarado en el bloque de ámbito más pequeño.

```
Class MiClase ... {  
    int n;  
    void metodo (...) {  
        n = 99;  
        for (int n = 0; n < 10, ++n) system.out.println(n);    //0 ... 9  
    }  
}
```

- 2.- Si dentro de una clase no hay ninguna declaración del identificador, se usa la declaración del identificador de otra clase que se haya importado y sea visible.

### En una jerarquía de clases:

- 1.- Toda entidad (variable o método) no privada, es visible dentro de la clase y todas sus subclases.
- 2.- Si tenemos dos entidades visibles con el mismo identificador, se considera que se refiere a la más específica.

C1 ----- ----- void f(...) Δ	C1 v1; C2 v2, x; C3 v3; C4 v4;
C2 ----- ----- Δ	x = v1; //NO puedo asignar a una clase una superclase x.f(); //se ejecuta f de C1 x = v2; //Correcto x.f(); //se ejecuta f de C1 x = v3; //Correcto
C3 ----- ----- void f(...) Δ	x.f(); //se ejecuta f de C3 x = v4; //Correcto x.f(); //se ejecuta f de C3
C4 ----- -----	

## **ESPECIFICACIONES DE LA 1a ENTREGA:**

### Análisis de requisitos + Especificación:

Enunciado ampliado	(análisis de requisitos)
Diagrama de Casos de Uso	(análisis de requisitos)
Diagrama de Clases del Dominio	(especificación)

### Análisis de requisitos:

#### **Resultado:**

requisitos del sistema

requisitos del software:

#### · Funcionales:

- describen las entradas y salidas de los datos y procesos
- funcionalidades

#### · NO funcionales:

estructurales:

interficies externas

restricciones de diseño (ej: arquitectura en 3 capas)

restricciones de hardware

eficiencia (**¡IMPORTANTE!**)

mantenibilidad

portabilidad

usabilidad

reusabilidad

documento explicativo: contiene el enunciado ampliado

casos de uso: contiene el diagrama de casos de uso)

### **Enunciado Ampliado:**

Documento explicativo (texto técnico, escrito en 3a persona, ...):

introducción: se explica cuál es el problema

justificación: se explica la problemática y se estudian las alternativas

**conclusiones: descripción del proyecto (funcionalidades y restricciones)**

/ \  
obligatorias opcionales

Cuando se escribe un texto técnico, es recomendable:

- Usar frases cortas, ordenadas entre sí y claras
- No abusar de terminología profesional

Los 7 pecados más comunes:

- Ruido: información innecesaria o redundante
- Silencio: falta de información
- Sobre-especificación: detallar más de lo necesario
- Contradicciones
- Ambigüedad y falta de precisión
- Referencias hacia adelante: no usar cosas aún no definidas
- Remordimiento: hacer precisiones o introducir restricciones sobre lo dicho anteriormente

### Diagrama de Casos de Uso:

Casos de uso: Es un documento que describe la secuencia de eventos que realiza el actor al usar el sistema para llevar a cabo un proceso:  
Relacionado con las funcionalidades del sistema

Actor: Entidad que participa en la secuencia de eventos:  
Admite herencia

En PROP tenemos que:

Identificar las funcionalidades del enunciado y asociarlas a los actores.

┌-----CLASE EXTRA-----┐

### MODELO CONCEPTUAL DE LOS DATOS

Es el diagrama de Clases del Dominio de la fase de diseño que más tarde, en la fase de implementación, implementaremos.

#### Fase de especificación:

Especificación de los Datos

- Modelo Conceptual de Datos :

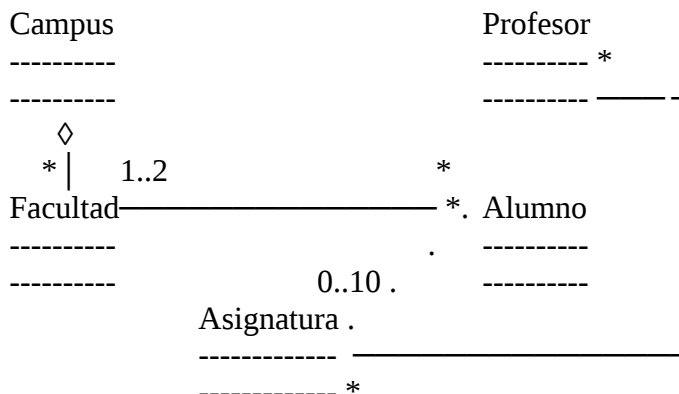
Es la representación que voy a tener del Dominio para resolver el problema. Es la representación de los conceptos significativos en el diagrama del problema.

Para representarlos con Orientación a Objetos, usaremos Clases que tendrán Atributos y Métodos. Usaremos **Relaciones entre las Clases** y Restricciones.

Especificación de los Procesos

- Diagramas de secuencia del sistema  
- Contratos de las operaciones del sistema  
- Modelo de estado

#### Relaciones entre Clases:



## Tipos de Relaciones (básicas)

### Generalización/Especificación:

Semántica: “es un”.

Está relacionado con la herencia.

Las clases están a diferentes niveles de abstracción.

Pueden tener diferente comportamiento.

### De Instancia:

Las instancias en la Clase están relacionadas de manera permanente con instancias de otras clases:

Asociación: “A tiene B”, “A conoce a B” (línea)

Agregación: “A forma parte de B”, “B contiene a A” (rombo)

Composición: Es como una agregación donde el contenido no tiene sentido sin el contenedor (rombo negro)

### De Dependencia/De Uso:

Son relaciones puntuales o temporales entre clases: “A usa B” (línea puntos)

## Clases de Dominio de la 1a Entrega

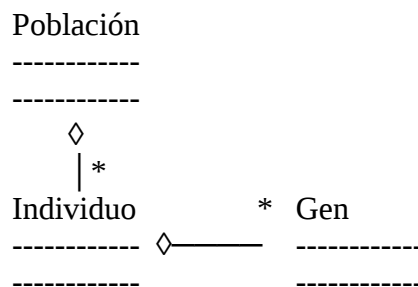
Clases: Datos necesarios para resolver el problema INDEPENDIENTEMENTE de cómo se resuelva.

Atributos: SÍ

Métodos: NO

Relaciones: SÍ (sin relaciones de uso)

Restricciones: SÍ

**Ejemplo algoritmo genético:****2ª Entrega:**

Fases de Diseño:

- Arquitectura de la aplicación (arquitectura en 3 capas: cómo vamos a estructurar el software)
- Diagrama de Clases de Diseño
- Diagramas de secuencia de las operaciones
- Contratos de las operaciones de las clases
- Lenguaje de programación (JAVA)
- Algoritmos
- Estructuras de Datos

**Arquitectura en 3 capas:**

Hay 3 cosas que considerar de manera independiente en una aplicación:

- |                                                                 |                |
|-----------------------------------------------------------------|----------------|
| -Interacción con el usuario                                     | → PRESENTACIÓN |
| -Dominio de la aplicación (lo que hay en el Diagrama de Clases) | → DOMINIO      |
| -Persistencia de los datos                                      | → PERSISTENCIA |

**CAPA DE PRESENTACIÓN:**

Se relaciona con los usuarios capturando las peticiones y presentando los resultados

Se relaciona con la capa de dominio, pasándole las peticiones y recibiendo los resultados del cálculo

**CAPA DE DOMINIO:**

Se relaciona con la capa de Presentación

Se relaciona con la capa de Persistencia a través de las peticiones de consulta y modificación de los datos persistentes

**CAPA DE PERSISTENCIA:**

Se relaciona con la capa de Dominio

Se relaciona con el sistema de gestión de Bases de Datos o el sistema de ficheros

Las comunicaciones entre las capas deberían ser vía Tipos Simples (vector de Strings, etc...)

## **LOS CONTROLADORES:**

Presentación:

Interacción con el Usuario	→ VISTAS
Interacción con el Dominio	→ CONTROLADORES

Dominio:

Interacción con Presentación/Persistencia	→ CONTROLADORES
Separar los datos de los procesos específicos del proyecto	→ CLASES DEL DOMINIO + CONTROLADORES

## **IMPLEMENTACIÓN:**

Herencia:

Dependencia/De Uso:

- 1.- Crear un objeto de la clase
- 2.- Usar sus métodos

Agregación:

```
class A {  
    int m; //definimos los atributos antes que los métodos  
    C agregacionC = new C(); //si tiene cardinalidad 1: variable interna de la clase  
    C[] agregacionC = new C[] //si tiene cardinalidad *: array  
    Vector <C> agregacionC = new Vector<C>(); //otra forma  
    void calculo1(...) {  
        ...  
    }  
}
```

Asociación:

También es una relación estructural permanente (al igual que la agregación), pero no hay una relación de contingencia del uno con el otro.  
Usar identificadores.

```
Class C {  
    Vector<X2> agregacionA2 ...  
}  
  
class A2 {  
    C agregacionC1 = ...  
}
```

# **JAVA:**

## Entorno de Trabajo:

Desarrollo → JDK

Ejecución → JRE

Versión que funcione en los PC instalados en la FIB (java -version) yo tengo la 1.6.0\_30

## Fuente:

Ficheros con extensión .java → para compilar: “javac \*.java” → obtengo un .class

El .class al pasar por la JVM (JavaVirtualMashine) se puede ejecutar → “java MainClass”

para ejecutar un fichero que no sea una clase “java -jar fichero.jar”

!!!!MIRAR los comandos “tar” y “jar”!!!!

## Entornos integrados (IDE):

Eclipse

Netbeans

Solo podemos usar librerías ESTÁNDAR de Java! Todo lo demás hay que consultarlo.

Estructura general de una clase:

```
[import ...]                                //para librerías/clases externas

    ⌈> visibilidad de la clase    ⌈> atributos de la clase
[public/...]    class MiClase [...] {
    [Atributos]                                //tipo1 nombre1, tipo2 nombre2, ...
    [Métodos]                                //void Calculo (int n, char c)
    [Clases Privadas]
}
```

ejemplo HelloWorld:

```
public class HelloWorld {

    public static void main (String[], args) {
        System.out.println(“Hello World!”);
    }
}
```

## Métodos:

Parecido a C:

```
void nombreP(parámetros) {
    ...           \_____ ; Paso de parámetros SIEMPRE por valor! (no hay '&', '*', ...)
}                /
tipo nombreF(parámetros) {
    ...
    return expresión;
}
```

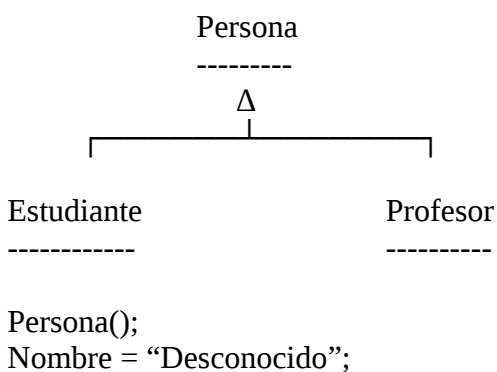
## Métodos constructores/destructores:

```
class Persona {
    string Nombre
    public Persona() {
        ...
    }
    public Persona(String n) {
        Nombre = n;
    }
    public void setNombre(String n) {
        Nombre = n;
    }
}
```

```
Persona p1, p2;
p1 = new Persona("Pepe");
p2 = new Persona();
```

```
p1 = new Persona(); → se cargaría el anterior p1
p2.setNombre("Pepe");
```

```
String s = new String("texto");
String s = "texto";
```



```
Estudiante e = new Estudiante();
//invoca al constructor de la Superclase → le daría
nombre "Desconocido"
```

```
public Estudiante(String n) {
    Super(n);    //llama al Padre
    ...
}
```



## La clase Object:

Toda clase hereda de Object.

Métodos:

```
Object clone();           //crea una copia del objeto;
Object equals(Object o);  //compara a los objetos;
class setClass();
String toString();
```

**Visibilidad:** desde el exterior de la clase (Clases, Atributos, Métodos)

Public: todo el mundo lo ve.

private: nadie más lo ve.

Package: por defecto.

Protected: visible en si jerarquía de clases → para que puedan verla los descendientes

## Las cláusulas STATIC y FINAL:

STATIC:

Atributos: Mismo atributo para TODOS los objetos de la clase.

Métodos: Mismo método para TODOS los objetos de la clase.

Se pueden acceder SIN instanciar:

```
class Persona {
    public Static int NPersonas;
    public Static int ContarPersonas() {
        return NPersonas;
    }
}
```

Persona p = new Persona();

```
if (p.NPersonas > 2) {
```

...

```
}
```

```
if (p.ContarPersonas() > 2) {
```

...

```
}
```

```
if (Persona.NPersonas > 2) {
```

...

```
}
```

```
if (Persona.ContarPersonas() > 2) {
```

...

```
}
```

FINAL:

class: no puede tener descendientes.

Método: no se puede redefinir en las subclases

Atributos: no se pueden modificar

## Implementación de conceptos de Orientación a Objetos:


### Encapsulamiento/Ocultación:

Cláusulas de visibilidad:

Atributos (→ **private**):

Consulta → **getAtributo**

Modificación → **setAtributo**

```
public boolean setAtributo (...) {  
     filto para comprobar que se pasan los parámetros correctos  
    if (!parametrosCorrectos) return false;  
    return true;  
}
```

### Herencia/Polimorfismo:

```
public class Persona {  
    public/protected/package  
    ...  
}
```

```
public class Estudiante extends Persona  
    super ...     hace referencia al padre      //constructores, atributos y métodos  
    this ...      hace referencia a sí mismo
```

## Compatibilidad de tipos en JAVA → cast:

La conversión de tipos es automática en JAVA:

entre tipos simples

en la herencia de padres a hijos //padre = hijo;

En el resto de casos hay que hacerla explícita:

```
class2 y;
```

```
class1 x =(class1) y;
```


### Clases abstractas:

```
public abstract class ClaseAbstracta {  
    public abstract void metodoAbstracto();  
    public void metodoConcreto(...) {  
        ...  
    }  
}
```

### Interfaces: clases completamente abstractas.

```
Public class MiCmase extends MiClasePadre implements Interface1, InterfaceN;
```

## Clases Parametrizadas/Genericidades:

 *>etiqueta (string, clase, ...): <E, m, ... >, <E extends A, m>*

```
public class MiClaseGenerica <E> {  
    E objectE;  
    public void calculo (E objectE) {  
        if (objectE.calculo() > 3) ...  
    }  
}  
  
Vector <integer> v = new Vector<integer> ();  
for (int i = 0; i < 10; ++i)    v.add(new integer(i*i));  
integer n = v.get(3);
```

-----01/04/2014

## Documentación de los proyectos:

2 tipos:

Documentación técnica: (enunciado ampliado, diagr. casos de uso, diagr. de clases, ...)

-carpeta del programa

-normas de programación:

**normas de codificación**

//2ª entrega

**política de comentarios (documentación del código)**

//2ª entrega

Documentación del usuario:

-manual de Referencia

**-manual del Usuario**

//3ª entrega

-manual de Aprendizaje/Tutorial

## **Manual de usuario:**

Es la descripción superficial de TODAS las funcionalidades del sistema (desde el punto de vista del usuario) y la secuencia de acciones que debe seguir el usuario para ejecutar las funcionalidades (con ayuda de Pantallazos, links, etc).

## **Normas de codificación:**

¿Qué pinta espero ver cuando lea el código? → a nivel de clúster (clases compartidas).

Decisiones del estilo:

- nombre de los identificadores (guiones bajos, mayúsculas, etc...)
- tabuladores entre un IF y su final, etc...

### **Identificadores:**

- usar nombres mnemotécnicos (nombres lógicos)
- permitimos ñ, ç, etc...?
- mayúsculas/separadores?
  - tres\_palabras\_separadas vs. TresPalabrasSeparadas vs. Tres\_Palabras\_Separadas
- indentación (espacios entre un IF y la siguiente línea, límite del tamaño de las líneas, ...)
- dónde ponemos los comentarios, cómo, cuándo?

## **PRUEBA DE PROGRAMAS:**

Para comprobar que nuestro programa funcione.

Fases del proceso de pruebas:

- 1.- **Detección de un error/disfunción**
- 2.- Localización del error
- 3.- Determinación de la fase del proyecto donde se ha producido
- 4.- Evaluación de la gravedad y el coste del arreglo
- 5.- Corrección
- 6.- Comprobación de que se haya resuelto y de que no se haya estropeado nada más

### **Detección de errores:**

-Verificación formal (program proving):

- Trata de demostrar (matemáticamente/lógicamente) que el programa funciona.
- Se requiere ser un especialista para poder aplicarlo → poca gente puede → caro
- Hay problemas que no se pueden verificar formalmente.
- Se suele usar en procesos especialmente críticos (software de una central nuclear).

-**Verificación experimental** (program testing)

- Trata de comprobar experimentalmente que el programa funciona.
- Se plantean una serie de situaciones iniciales, ejecutamos el programa y comprobamos que las salidas sean exactamente lo que esperábamos que fuesen.
- Es imposible probar todos los casos.
- Hay que conseguir un conjunto adecuado de situaciones a probar.
- No tenemos garantía de tener ausencia de errores.
- Paramos cuando el programa funcione relativamente bien.

## PROGRAM TESTING:

Hay dos estrategias:

### -Caja Blanca:

El probador conoce el software por dentro (implementación, código fuente, ...) y sabe cuales son los puntos críticos.

Los juegos de pruebas se diseñan para estudiar los casos críticos y para pasar por todas las líneas del código (en la medida de lo posible).

### -Caja Negra:

El probador NO conoce el software por dentro.

Entradas (probador) – Salidas (programa)

Situaciones representativas.

### -Caja Gris:

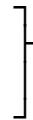
El probador conoce el software por dentro pero las pruebas se hacen en formato

Entrada – Salida.

## PROGRAM TESTING – NIVELES:

### 1.- PRUEBA DE COMPONENTES:

Son las pruebas de cada clase de manera aislada.



2ª entrega

### 2.- PRUEBAS DE INTEGRACIÓN:

Son las pruebas de un conjunto de clases relacionadas entre sí por una funcionalidad.

### 3.- PRUEBAS DEL SISTEMA:

Son las pruebas de todas las clases.

3ª entrega

## 1.- PRUEBA DE COMPONENTES: DRIVERS Y STABS

claseAProbar

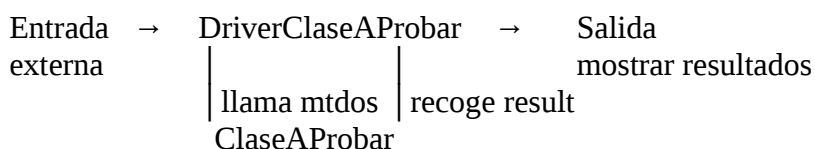
-----

atributos

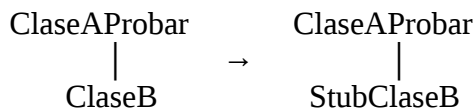
-----

métodos (get y set)

Estrategia: creo una clase: crea objetos de la clase a probar, llama métodos y recoge resultados.



Para saber dónde se ha producido el error (de la clase a probar), se crean los Stubs de las clases que están relacionadas con la clase que quiero probar, de forma muy simple.



Para **cada clase** ha de haber un Driver y sus Stubs (por cada una de las clases que utilice ésta).

```

Public class Clase A {
    private String Atrib1;
    private ClaseB Atrib2;

    public Clase A();
    public boolean setAtrib1(String S) {...}
    public boolean SetAtrib2(ClaseB O) {...}
    public String getAtrib1() {...}
    public ClaseB getAtrib2() {...}
    public int calculoComplejo() {
        int x = Atrib2.calculo();
    }
}

```

DriverClaseA

```

-----
testConstructor()
testSetAtrib1()
testSetAtrib2()
testGetAtrib1()
testGetAtrib2()
testCalculoComplejo()

```

1.-pedir datos  
2.-llamar metodo de ClaseA  
3.-mostrar/comprobar resultados

StubClaseB

```

-----
int calculoComplejo(...)

```

Si la clase a probar es una clase ABSTRACTA, no la puedo instanciar, pero si la puedo probar:

ClaseAbstracta

```

-----
-----

```

Δ  
|

StubClaseAbstractaNoAbstracta

```

-----
abstracta → implementado

```

## **2.- PRUEBA DE INTEGRACIÓN:**

¿Por dónde empezar?

- Top-Down:
- Bottom-Up:
- Híbrida:

## **COSAS QUE TENEMOS EN JAVA PARA DEPURAR PROGRAMAS:**

### 1.- JUnit:

Software libre (plugin) que permite gestionar los JDP (juegos de pruebas)

### 2.- Assert:

Es una instrucción que podemos poner en el programa para comprobar si pasa algo.

```
Assert expresionBooleana [:String] {  
    ...  
}
```

Cuando la máquina virtual pase por allí, comprobará si es cierta o no. En caso de ser falsa, se lanza una excepción.

Podemos decirle que si no se cumple algo, no pase de ésta línea de código.

EJECUCIÓN:        java -ea        (java -enableassertion)

### 3.- Excepciones:

Se lanza una excepción cuando ha pasado algo, el método no continúa y el llamador del método ha de hacer algo con esa excepción.

Hay de 2 tipos:

-java lang Error (subclase de Throwable):

    No se suele querer continua la ejecución del programa.

-java lang Exception (subclase de Throwable):

    IndexOutOfBounds: posicion de un vector que no tiene espacio

    NullPointerException: vector no inicializado

    FileNotFoundException

Si un método lanza una excepción, el método que la ha llamado puede hacer dos cosas:

    1.- NO tratarlo: pasarle la pelota a quién me ha llamado poniendo en la declaración:

```
public void método.... throws Exception  
public void método.... throws IndexOutOfBoundsException
```

    2.- tratarlo:

```
try {  
    métodoDeLaLlamada  
}  
catch (Exception e) {  
    ....  
}  
finally {}        //se ejecuta haya excepción o no
```

**AWT:**

**Componentes:**

Componentes primarios: widgets

Componentes contenedores: controladores principales

**Eventos:**

Listener

Eventos

**Threads y Event-dispatching Threads:**

...

**SWING:**