

## **TEMA 1: Introducción al Paradigma Orientado a Objetos**

**Abstracción:** supresión intencionada u ocultación de algunos detalles de un proceso o artefacto, con el fin de destacar más claramente otros aspectos, detalles o estructuras. Crea modelos de la realidad.

Los diferentes niveles dependen de los mecanismos proporcionados por el lenguaje elegido:

- Perspectiva funcional: ensamblador, procedimientos y módulos.
- Perspectiva de datos: Paquetes y TAD
- Perspectiva de servicios: Objetos: TAD, paso de mensajes, herencia y polimorfismo

Los lenguajes de programación proporcionan abstracciones:

- Espacio del problema -> Lenguajes orientados a objetos (LOO): Smalltalk, Eiffel
- Espacio de la solución -> Ensamblador, imperativos (C, Fortran, BASIC), específicos (LISP, PROLOG)
- LOO híbridos (multiparadigma): C++, Object Pascal, Java,...

**Paradigma:** forma de entender y representar la realidad. Conjunto de teorías, estándares y métodos que, juntos, representan un modo de organizar el pensamiento.

**Ocultación de información:** omisión intencionada de detalles de implementación tras una interfaz simple.

**Encapsulación:** cuando existe una división entre la vista interna (implementación, cómo lo hace) de un objeto y su vista externa (interfaz, qué sabe hacer el objeto). Favorece la intercambiabilidad y, la comunicación del equipo de desarrollo y la interconexión del trabajo de cada miembro.

**Paradigma orientado a objetos:** metodología de desarrollo de aplicaciones en la cual estas se organizan como colecciones cooperativas de objetos, cada uno de los cuales representan una instancia de alguna clase, y cuyas clases son miembros de jerarquías de clases unidas mediante relaciones de herencia.

Herramienta para resolver la llamada crisis del software. Escala muy bien, proporciona un modelo de abstracción que razona con metáforas y, gran desarrollo de herramientas orientadas a objetos en todos los dominios.

Mundo estructurado en:

**Agentes y comunidades:** un programa OO se estructura como una comunidad de agentes que interaccionan (objetos). Cada uno juega un rol en la solución del problema. Proporciona un servicio o realiza una acción que es utilizada por otros miembros de la comunidad.

**Paso de mensajes:** a un objeto se le envían mensajes para que realice una acción, y éste selecciona un método para realizarla. unJuego.mostrarCarta(laCarta,42,47)

Se diferencia de un procedimiento/llamada a función en:

- En un mensaje siempre hay un receptor, en una llamada a procedimiento no.
- La interpretación de un mismo mensaje puede variar en función del receptor del mismo.

**Organización:** en clases (datos + operaciones de datos)

**Responsabilidades:** comportamiento de un objeto. Incrementa el nivel de abstracción y permite mayor independencia entre objetos.

**Protocolo:** conjunto de responsabilidades de un objeto.

**Objeto:** encapsulación de un estado (valores de los datos) y comportamiento (operaciones).

**Clase:** agrupación de objetos, donde el objeto es una instancia de una clase.

**Herencia:** jerarquía de clases: una clase puede heredar propiedades de otra clase que está en una escala más alta de la jerarquía.

**Enlace de métodos:** instante en el cual una llamada a un método es asociada al código que se debe ejecutar.

- Estático: en tiempo de compilación
- Dinámico: en tiempo de ejecución

Características básicas de un LOO:

- Todo es un objeto
- Cada objeto es construido a partir de otros objetos.
- Todo objeto es instancia de una clase.
- En la clase se define el comportamiento de los objetos y su estructura interna.
- Las clases se organizan en una estructura arbórea de raíz única (jerarquía de herencia)
- Un programa es un conjunto de objetos que se comunican mediante el paso de mensajes.

Opcionales:

**Polimorfismo:** capacidad de una entidad de referenciar elementos de distinto tipo en distintos instantes.

**Genericidad:** definición de clases parametrizadas (templates en C++, generics en Java) que definen tipos genéricos. Lista <tipo>

**Gestión de errores:** tratamiento de condiciones de error mediante excepciones.

**Aserciones:** especifican qué hace el software.

- **Precondiciones:** propiedades que deben ser satisfechas cada vez que se invoca un servicio.
- **Postcondiciones:** deben ser satisfechas al finalizar la ejecución.
- **Invariantes:** expresan restricciones para la consistencia global de sus instancias.

**Tipado estático:** imposición de un tipo a un objeto. Se asegura en tiempo de compilación que un objeto entiende los mensajes que se le envían y evita errores en tiempo de ejecución.

**Recogida de basura** (garbage collection): permite liberar automáticamente la memoria de aquellos objetos que ya no se utilizan.

**Concurrencia:** permite que diferentes objetos actúen al mismo tiempo, usando diferentes threads.

**Persistencia:** propiedad por la cual la existencia de un objeto trasciende la ejecución del programa.

**Reflexión:** capacidad de un programa de manipular su propio estado, estructura y comportamiento.

Parámetros de calidad extrínsecos: fiabilidad

**Corrección:** capacidad para realizar con exactitud sus tareas.

**Robustez:** capacidad de reaccionar ante condiciones excepcionales.

Parámetros de calidad intrínsecos: modularidad

**Extensibilidad:** facilidad de adaptar a los cambios de especificación. Simplicidad de diseño

**Reutilización:** capacidad de servir para la construcción de muchas aplicaciones diferentes

**Mantenibilidad:** producir aplicaciones + fáciles de cambiar.

## TEMA 2: Conceptos Básicos de la Programación Orientada a Objetos

Objeto:

**Estado**: conjunto de propiedades y sus valores actuales.

**Comportamiento**: modo en que actúa y reacciona ante los mensajes que se le envían (con posibles cambios en su estado). Determinado por su clase.

**Identidad**: propiedad que lo distingue de otros (nombre único de variable).

Clase: conjunto de objetos que comparten una estructura y un comportamiento comunes.

- **Id. de clase**: nombre.
- **Propiedades**: atributos, roles, operaciones, métodos, servicios.

Atributo: (dato miembro o variable de instancia) porción de información que un objeto posee. Suelen ser a su vez objetos y se 'declaran' como 'campos' de la clase.

- **Visibilidad**: indica desde dónde se puede acceder a él.
  - +Pública (interfaz) -> desde cualquier lugar
  - Privada (implementación) -> sólo desde la propia clase
  - #Protegida (implementación) -> desde clases derivadas
  - ~De paquete (en Java) -> desde clases definidas en el mismo paquete
- **De instancia**: atributos o características de los objetos representados por la clase. Se guarda espacio para una copia de él por cada objeto creado.
- **De clase**: características de una clase comunes a todos los objetos de dicha clase. Reserva una zona de memoria para todos los objetos.

Operación: definen el comportamiento del objeto. Tienen asociada una visibilidad.

- **De instancia**:
  - Operaciones que pueden realizar los objetos de la clase.
  - Pueden acceder directamente a atributos tanto de instancia como de clase.
  - Normalmente actúan sobre el objeto receptor del mensaje.

```
Circulo c=new Circulo();
c.setRadio(3);
double r=c.getRadio();
c.pintar();
```

```
Void setRadio(double r) {
    If(r>0.0) radio=r;
    else r=0.0; }
```

- **De clase**:
  - Operaciones que acceden exclusivamente a atributos de clase.
  - No existe receptor del mensaje (a menos que se pase explícitamente como parámetro).
  - Se pueden ejecutar sin necesidad de que exista ninguna instancia.

```
class Circulo {
    private static final double pi =3.141592;
    public static double getRadioPerimetro()
    {return 2*pi;}
}
```

- **Órdenes**: pueden modificar el estado del objeto receptor. c.setRadio(3);
- **Consultas**: no modifican al objeto receptor. c.getRadio();

- **Operaciones sobrecargadas:** existencia, dentro de un mismo ámbito, de más de una operación con el mismo nombre, pero con diferente número y/o tipo de argumentos.

Constructor: crea e inicializa objetos.

- Se invoca siempre que se crea un objeto (con new en Java).
- Tienen el mismo nombre que la clase y no devuelven nada.
- Si no se define ninguno, el compilador genera uno con visibilidad pública, llamado constructor de oficio. `super();`

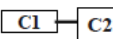


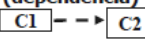

Copia de objetos:

- **Shallow copy:** modo de copia por defecto. Copia bit a bit los atributos de un objeto.
- **Deep copy:** copia completa. En C++ un constructor de copia. En java constructor de copia o el método `clone()`.

Destrucción de objetos: (en Java método `finalize()`, en C++ destructores). Se usan para liberar recursos que el objeto haya podido adquirir (cerrar ficheros abiertos, conexiones con la bd,...)

Forma canónica de una clase: conjunto de métodos que toda clase debería definir. El compilador proporciona una definición 'de oficio'.

- C++: constructor por defecto y de copia, destructor y operador de asignación.
- Java: representación del objeto en forma de cadena (`toString()`), comparación de objetos (`equals(objeto)`), id numérico de un objeto (`hashCode()`).

	Persistente	No persist.
Entre objetos	<ul style="list-style-type: none"> <li>• <b>Asociación</b> </li> <li>• <b>Todo-Parte</b> <ul style="list-style-type: none"> <li>• Agregación </li> <li>• Composición </li> </ul> </li> </ul>	<ul style="list-style-type: none"> <li>• <b>Uso (dependencia)</b> </li> </ul>
Entre clases	<ul style="list-style-type: none"> <li>• <b>Generalización (Herencia)</b> </li> </ul>	

Relaciones: el conocimiento entre objetos se realiza mediante el establecimiento de ellas. Se pueden establecer entre clases o entre objetos.

- **Persistentes:** recogen caminos de comunicación entre objetos que se guardan y pueden reutilizarse.
- **No persistentes:** recogen caminos de comunicación entre objetos que desaparecen tras ser utilizado.

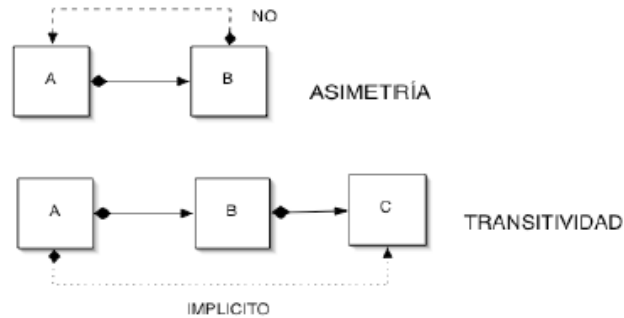
Asociación: expresa una relación entre los objetos instanciados a partir de las clases conectadas.

- **Navegabilidad:** sentido en que se recorre la asociación.
- **Rol:** papel que juega el objeto
- **Multiplicidad:** número de objetos mínimo y máximo que pueden relacionarse con un objeto.

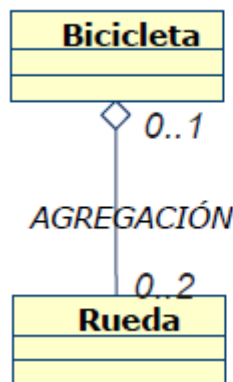
Los objetos asociados son independientes, si uno se crea o desaparece, sólo se creará o desaparecerá la relación.

- **Implementación:** una sola referencia o un vector de referencias del tamaño indicado por la cardinalidad máxima.

Todo-parte: relación en la que un objeto forma parte de la naturaleza de otro. A diferencia de la asociación, presenta simetría y transitividad.



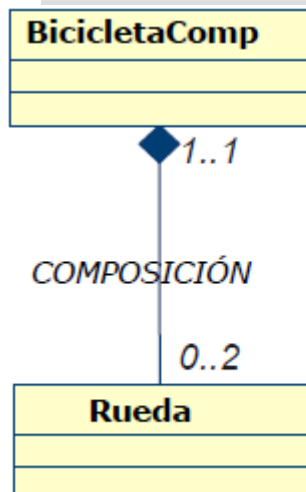
- **Agregación:** asociación binaria ('tiene-un', 'pertenece-a'); si A desaparece, B no. Se implementa como una asociación unidireccional.
- **Composición:** agregación 'fuerte'. Una instancia 'parte' está relacionada, como máximo, con una instancia 'todo' en un instante dado. Si desaparece A, B también.



```
class Rueda {
    private String nombre;
    public Rueda(String n){nombre=n;}
}

class Bicicleta {
    private Vector<Rueda> r;
    private static final int MAXR=2;
    public Bicicleta(Rueda r1, Rueda r2){
        r.add(r1);
        r.add(r2);
    }
    public void cambiarRueda(int pos, Rueda raux){
        if (pos>=0 && pos<MAXR)
            r.set(pos,raux);
    }

    public static final void main(String[] args)
    {
        Rueda r1=new Rueda("1");
        Rueda r2=new Rueda("2");
        Rueda r3=new Rueda("3");
        Bicicleta b(r1,r2);
        b1.cambiarRueda(0,r3);
    }
}
```



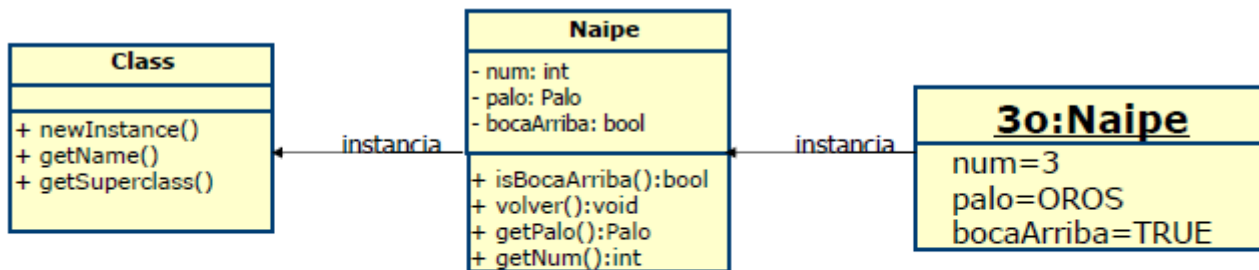
```
class BicicletaComp{
    private static const int MAXR=2;
    private Vector<Rueda> r;
    public BicicletaComp(String p,String s){
        r.add(new Rueda(p));
        r.add(new Rueda(s));
    }
    public static final void main(String[] args) {
        BicicletaComp b2 = new BicicletaComp("1","2");
        BicicletaComp b3 = new BicicletaComp("1","2");

        //son ruedas distintas aunque con el mismo nombre
    }
}
```

Dependencia: una clase A usa una clase B cuando no contiene datos miembros del tipo especificado por la clase B pero:

- Utiliza alguna instancia de la clase B como parámetro en alguno de sus métodos para realizar una operación.
- Accede a sus variables privadas (clases amigas)
- Usa algún método de clase de B.

Metaclases: métodos que se asocian con clases (new, delete; métodos estáticos). En Java, una clase es una instancia de otra clase, llamada Metaclase, por lo que, las clases pueden responder a ciertos mensajes (creación de objetos new).



P. ej. en Java:

```
Naipes n1 = new Naipes();
Class c = n1.getClass();
Naipes n2 = (Naipes) c.newInstance();
```

124

### **TEMA 3: Introducción al Diseño Orientado a Objetos**

**Pequeños proyectos** (programming in the small): Pocos programadores. El mayor problema es el diseño y desarrollo de algoritmos para resolver el problema actual.

**Grandes proyectos** (programming in the large): gran equipo de programadores. El mayor problema en el proceso de desarrollo es el manejo de detalles y la comunicación entre los programadores.

Interfaz e implementación:

- **Principio de Parnas**: el desarrollador proporciona la información necesaria al usuario para utilizar el software. El desarrollador recibe la información necesaria para hacer el software.

Métricas de calidad:

- **Acoplamiento**: relación entre componentes software. Interesa bajo acoplamiento. Se consigue dando las tareas al que ya tiene habilidad para hacerlas.
- **Cohesión**: grado en que las responsabilidades de un solo componente forman una unidad significativa. Interesa alta cohesión.

**Diagrama de clases UML**: define las clases, sus propiedades y cómo se relacionan unas con otras.

Diseño dirigido por responsabilidades (RDD): proporciona técnicas para el modelado de roles, responsabilidades y colaboración de objetos.

- **Principios**: maximizar la abstracción, distribuir el comportamiento, crear objetos 'inteligentes', preservar la flexibilidad.

- **Artefactos:**
  - Aplicación: conjunto de objetos interactivos.
  - Objeto: implementación de uno o más roles.
  - Rol: conjunto de responsabilidades.
  - Responsabilidad: obligación de realizar una tarea o conocer cierta información.
  - Colaboración: interacción entre objetos y/o roles.

## **TEMA 4: Gestión de Errores**

**Código espagueti:** 1º lleva a cabo una tarea. 2º si se produce error, lleva a cabo las tareas de errores. Mezcla la lógica del programa con el tratamiento de errores. El código cliente (llamador) no está obligado a tratar el error. Los 'códigos de error' no son consistentes.

**Excepción:** objeto que contiene información del error, y que interrumpe el flujo normal de las sentencias durante la ejecución de un programa. No se puede ignorar, aborta el programa.

- Comportamiento: se lanzan (throw) por un método cuando se detecta una condición de error. Se capturan (try/catch) por el código cliente (llamador), si éste no lo captura, la excepción 'saldrá' a un ámbito más externo hasta que sea capturada, si no el programa abortará.

En C++ y Java try/catch. Sólo en Java finally para cerrar ficheros o liberar otros recursos del sistema. Se puede ejecutar tras el bloque try o después de los catch.

- Funcionamiento: 1º Ejecuta instrucciones try, si hay error va al catch correspondiente. 2º Continuar la ejecución después de los bloques catch.
- Especificación de excepción: indica qué excepciones puede lanzar el método. `Public int f() throws E1, E2 {...}, f()` puede lanzar excepciones de tipo E1 y E2.
- Uso correcto: un método que lanza una excepción señala el error pero no debe tratarlo, lo hará quien invoque a dicho método.

**Excepción estándar:** tienen al menos 2 constructores, uno por defecto y otro que acepta un string.

- Throwable: clase de la que derivan todas las excepciones lanzadas por componentes de la API de Java. Definida en `java.lang`
- Exception: indican errores de ejecución del API de Java o nuestros.
  - Verificadas (checked): si son lanzadas por un método, Java obliga a que lo declare en su especificación de excepciones, en tiempo de compilación. (excepto las de tipo `RuntimeException`.)
  - No verificadas (unchecked): todas las que derivan (extends) de `RuntimeException`. Si un método puede lanzarlas, no es obligatorio incluirlas en su especificación de excepciones.
- Error: indican errores de compilación o del sistema. No suelen capturarse.
- RuntimeException: errores de programación, no del usuario. No suelen capturarse.

**Excepción de usuario:** podemos incluir información extra al lanzarla y agruparlas en jerarquías de clases. Se crean tomando como base la clase `Exception`. `class miExcep extends Exception{...}`

**Orden de captura:** la colocación de los catch sí importa. Los bloques try/catch se pueden anidar.

**Excepción en constructor:** si se produce una excepción en un constructor, y no se captura dentro, el objeto no se creará.

**Regenerar una excepción:** tratar parte del problema que generó la excepción y regenerarla para tratarlo a un nivel superior. 1º Volver a llamar a la misma excepción dentro del catch. 2º Lanzar una nueva excepción dentro del catch.

Ventajas: separar el manejo de errores del código normal. Agrupar los errores y diferenciarlos. Obliga a tratar o ignorar las condiciones de error.

Inconvenientes: sobrecarga del sistema para gestionar los flujos de control de excepciones.

JAVA

```
void Func() {  
  
    if(detecto_error1) throw new Tipo1();  
    ...  
    if(detecto_error2) throw new Tipo2();  
    ...  
}
```

constructor

```
try  
{  
    // Código de ejecución normal  
    Func(); // puede lanzar excepciones  
}  
catch (Tipo1 ex)  
{  
    // Gestión de excep tipo 1  
}  
catch (Tipo2 ex)  
{  
    // Gestión de excep tipo 2  
}  
finally {  
    // se ejecuta siempre  
}  
//continuación del código
```

C++

```
void Func() {  
  
    if (detecto_error1) throw Tipo1();  
    ...  
    if (detecto_error2) throw Tipo2();  
    ...  
}
```

constructor

```
try  
{  
    // Código de ejecución normal  
    Func(); // puede lanzar excepciones  
    ...  
}  
catch (Tipo1 &ex)  
{  
    // Gestión de excep tipo 1  
}  
catch (Tipo2 &ex)  
{  
    // Gestión de excep tipo 2  
}  
catch (...)  
{  
    /* Gestión de cualquier excepción no  
       capturada mediante los catch  
       anteriores*/  
}  
//Continuación del código
```



## Tema 5: Herencia

Mecanismo de implementación en el que elementos más específicos incorporan la estructura y comportamiento de elementos más generales. Gracias a ella es posible especializar o extender la funcionalidad de una clase.

Siempre es transitiva: una clase puede heredar características de superclase que están varios niveles arriba.

- De implementación: la implementación de los métodos es heredada. Puede sobrescribirse en las clases derivadas.
- De interfaz: sólo se hereda la interfaz, no hay implementación a nivel de clase base
- Simple: única clase base
- Múltiple: más de una clase.

### Herencia de Implementación

**Simple**: las propiedades definidas en una clase base son heredadas por la derivada, ésta puede añadir atributos, métodos o roles.

La parte privada de una clase base no es directamente accesible desde la clase derivada

- Visibilidad protected: directamente accesibles desde la propia clase y sus derivadas. Privada para el resto de ámbitos (#).
- Pública: se hereda interfaz e implementación. Java sólo soporta esta. (class circ extends F2D{...})
- Protected y pública (C++): solo permite heredar la implementación. La interfaz de la clase base queda inaccesible desde objetos de clase derivada.

En una clase derivada se pueden añadir nuevos métodos/atributos y modificar los métodos heredados de la clase base.

- Refinamiento: se añade comportamiento nuevo antes y/o después del heredado.
  - This: referencia a objeto actual usando implementación de la clase actual.
  - Super: ídem this, en vez de clase actual, clase base.
- Reemplazo: el método heredado se redefine sustituyendo al de la clase base.
- **Constructor**: no se heredan, siempre son definidos para las clases derivadas. Primero se ejecuta el constructor de clase base y luego de la derivada. La derivada aplica una política de refinamiento.
  - Ejecución implícita del constructor por defecto de clase base al invocar a un constructor de clase privada.
  - Ejecución explícita de cualquier otro tipo de constructor en la zona de inicialización (constructor de copia).
  - El orden de construcción va de la clase base a la derivada.
  - El orden de destrucción en Java es responsabilidad del programador. Se usa el método finalize() (no se sabe cuándo se ejecutará), o creamos propios.
- **Upcasting**: convertir un objeto de tipo derivado a tipo base. Cuando se convierten objetos en C++, se hace object slicing. Con referencias (en Java o C++) NO hay object slicing.
- Particularidades herencia:
  - En las jerarquías de herencia hay un refinamiento implícito de constructor por defecto
  - Los constructores sobrecargados se refinan explícitamente.
  - Las propiedades de clase definidas en la clase base también son compartidas (heredadas) por las clases derivadas.

**Múltiple**: se da al existir más de una clase base. Java no soporta herencia de implementación múltiple, pero sí de interfaz. C++ sí, se heredan interfaces e implementaciones de la clase base.

## Herencia de Interfaz:

No hereda código, sólo interfaz (a veces con implementación parcial o por defecto).

Objetivos: separar la interfaz de la implementación y garantizar la sustitución.

- **Sustitución:** poder usar objetos de una subclase en vez de objetos de la superclase sin que afecte al código de la superclase.
  - Java lo hace directamente [Panad p = new Panad(); Depend d1=p //sustitución]
  - C++ a través de punteros o referencias [Panad p; Depend& d1=p; //sust. Depend\* d2=&p; //susti. Depend d3=p; //no sustit. Object slicing]
  - Tipado estático: lenguajes fuertemente tipados. Caracterizan los objetos por su clase.
  - Tipado dinámico: lenguajes débilmente tipados. Caracterizan los objetos por su comportamiento.
- **Tiempo de enlace:** momento en el que se identifica el trozo de código donde se llama a un método o el objeto concreto asociado a una variable.
  - **Estático de objeto:** el tipo de objeto que tiene una variable se determina en tiempo de compilación.
  - **Dinámico de objeto:** el sistema gestionará la variable en función de la naturaleza real del objeto que reference durante la ejecución.
    - Java usa dinámico con objetos y estático con los escalares [Fig2D f=new Circ()]
    - C++ con punteros o referencias y solo en jerarquía de herencia [Fig2D \*f=new Circ()]
  - **Estático de métodos:** en tiempo de compilación se elige el método que responderá al mensaje.
  - **Dinámico de métodos:** ídem anterior pero en tiempo de ejecución.
- **Clases abstractas:** alguno de sus métodos no está definido. Tienen enlace dinámico. No se crean objetos de estas clases. Sus referencias apuntan a objetos de clases derivadas. Las derivadas deben implementar todos los métodos abstractos. Implementa el interfaz de la abstracta. Garantiza el principio de sustitución.
- **Interfaces:** declaración de un conjunto de métodos abstractos. Separación total de interfaz e implementación. En Java/C# las clases pueden implementar más de un interfaz (herencia múltiple de interfaz).

Herencia de implementación

Habilidad para que una clase herede parte o toda su implementación de otra clase.

Uso seguro:

Beneficios:

Costes:

Técnicas de rehúso:

Tema 6: Polimorfismo

## **Tema 7: Reflexión**

Infraestructura del lenguaje que permite a un programa conocerse y manipularse a sí mismo en tiempo de ejecución. Consiste en metadatos más operaciones que los manipulan

Se usa para construir nuevos métodos y arrays, acceder y modificar atributos de instancia o de clase, invocar métodos de instancia y estáticos, acceder y modificar elementos de arrays, etc.

Todo ello sin necesidad de conocer el nombre de las clases en tiempo de compilación. Esta información puede ser proporcionada en forma de datos en tiempo de ejecución.

API de reflexión de Java: proporciona dos formas de obtener información sobre los tipos en tiempo de ejecución.

- **RTTI tradicional** (upcasting, downcasting): cuando el tipo de un objeto está disponible en tiempo de compilación y ejecución.
- **Reflexión**: el tipo de un objeto puede no estar disponible en tiempo de compilación y/o ejecución.

Clases del API de reflexión:

- `Java.lang.reflect`: paquete de reflexión
- `Java.lang.reflect.Array`: proporciona métodos estáticos para crear y acceder dinámicamente a los arrays.
- `Java.lang.reflect.Member`: interfaz que refleja información sobre un miembro de una clase.
- `Java.lang.reflect.Constructor`: proporciona información y acceso sobre un constructor
- `Java.lang.reflect.Field`: proporciona información y acceso dinámico a un atributo de instancia o de una clase static.
- `Java.lang.reflect.Method`: proporciona información y acceso a un método.
- `Java.lang.Class`: representa clases e interfaces
- `Java.lang.Package`: proporciona información sobre un paquete
- `Java.lang.ClassLoader`: clase abstracta. Proporciona servicios para cargar clases dinámicamente.

**Objeto Class**: para toda clase que se carga en JVM tiene asociado un objeto de tipo class. Contiene información sobre una clase (métodos, campos, superclase, interfaces, etc)

- `Class.forName(string)`: obtiene el objeto class que tiene el mismo nombre que el string.
- `String getName()`: devuelve el nombre de la clase del objeto.
- `Boolean isInterface()`: devuelve cierto si la clase es una interfaz
- `Boolean isArray()`: true si es un array
- `Class getSuperclass()`: devuelve el objeto que representa la clase base actual.
- `Class[] getInterfaces()`: devuelve un array de objetos class que representa las interfaces de esta clase.
- `Object newInstance()`: crea una instancia de esta clase.
- `Constructor[] getConstructors()`: devuelve un array con todos los constructores públicos de la clase
- `Method[] getDeclaredMethods()`: devuelve un array con todos los métodos privados y públicos de la clase
- `Method[] getMethods()`: devuelve un array con todos los métodos públicos en la clase, los de la clase base o interfaces implementadas por la clase.

- Method `getMethod(String methodName, Class[] paramTypes)`: devuelve un objeto Method que representa el método publico identificado por su nombre y tipo de parámetros, declarado en esta clase o heredado de una clase base.
- Method `getDeclaredMethod(String methodName, Class[] paramTypes)`: idem anterior y el método puede ser privado. No heredado de una clase base.

**La clase Array:** 2 formas de declararla:

```
//1:
Perro perrera = new Perro[10];
//2:
Class c1 = Class.forName("Perro");
Perro perrera = (Perro[]) Array.newInstance(c1, 10);
```

**Interface Member:** representa a un miembro de una clase. Implementado por Constructor, Method y Field.

- Class `getDeclaringClass()`: devuelve el objeto class que representa a la clase donde se declara el miembro.
- Int `getModifiers()`: devuelve un entero que representa los modificadores aplicados a este miembro.
- String `getName()`: devuelve el nombre simple del miembro.

**Class Method:** con un objeto Method podemos obtener su nombre y sus parámetros e invocarlo, obtener un método o listado de los que tiene a partir de una signatura.

- String `name = meth.getName()`: obtener el nombre de un método.
- Class `parms[] = meth.getParameterTypes()`: obtener un array de tipos de los parámetros
- Class `retType = meth.getReturnType()`: obtener el tipo de retorno de un método.
- Method `invoke()` - public Object `invoke(Object obj, Object[] args)`: invoca a un método. El 1er argumento es el objeto receptor, el 2º es la lista de parámetros.

**Objetos Field:** listado de atributos que tiene un objeto. Podemos obtener el nombre del atributo (`getName()`), su tipo (`getType()`), su valor (`get()`), asignarle uno (`set(recep, valor)`), etc.

Ventajas:

- Técnica común entre LOO como SmallTalk, Eiffel, Java,...
- Ayuda a mantener software robusto
- Las aplicaciones son más flexibles, extensibles y uso de plugins
- Es fácil, mejora la reusabilidad del código
- Puede simplificar el código fuente y el diseño, aumentando el rendimiento de la aplicación
- Puede usarse con el estándar 100% Pure Java
- Eliminar código condicional

Inconvenientes:

- No se pueden saber las clases derivadas de una clase dada por culpa de la carga dinámica de clases en JVM
- No saber le método que se está ejecutando

## **Tema 8: Frameworks**

Es un esquema (esqueleto, patrón) para el desarrollo e implementación de una aplicación que podemos modificar, implementar interfaces, herencia de clases abstractas, ficheros de configuración. Llama a nuestras clases.

**Librería:** conjunto de clases que realizan funciones más o menos concretas. El usuario llama a las clases de las librerías. No requieren que implementemos ni heredemos nada.

- El fabricante nos proporciona una API (conjunto de clases y sus métodos).
- Podemos considerar librerías a la implementación de referencia del JCF incluido en el JDK: `java.util.ArrayList`, `java.util.Stack`, `java.util.TreeSet`

Los frameworks y las librerías implementan funcionalidades útiles que el propio lenguaje no incorpora.

**JCF** (Java Collection Framework): conjunto de clases incluido en el JDK representando tipos de datos abstractos de datos básicos: pilas, colas, vectores, mapas, etc.

- Tiene formato de framework porque está diseñado para que cualquiera pueda usarlo como base para realizar su propia implementación de cada tipo de dato (extendiendo los interfaces).
- Proporciona implementaciones de referencia para cada interface, si sólo usamos esas, JCF se usa como librería.

**JDBC** (Java database Connection): Framework para conectar Java con sistemas de bases de datos. Los desarrolladores usamos directamente las librerías de los fabricantes (drivers o conectores). Estructura: 1º crear conexión, 2º consultar/manipular la bd mediante la clase `Statement`, 3º cerrar conexión.

Tratamiento de XML: métodos para analizar XML:

- **DOM Parser:** lee todo el XML formando el árbol de elementos en memoria. Funciona como librería.
- **SAX Parser:** usado para grandes documentos. Funciona como Framework. Cada vez que se abre y cierra un elemento se invoca a un callback (puntero a función proporcionada por nosotros).

**Logging:** framework para la emisión eficiente de logs. Es configurable mediante código o con ficheros

**Hibernate:** Mapeo Objeto/Relacional (O/R). A partir de unos ficheros de configuración genera clases que gestionan las operaciones (create, retrieve, update, delete) que hacen persistentes los objetos en bd.

**Apache Commons:** conjunto de librerías incluidas en el proyecto Apache. Útiles:

- CLI: parámetros para las utilidades en línea de comando.
- Collections: añade más tipos abstractos de datos.
- Configuration: ficheros de configuración
- Email
- FileUpload: subir ficheros a nuestro servidor
- Math: para operaciones estadísticas.

Otros Frameworks:

- **GWT** (Google Web Toolkit): generación de aplicaciones ricas web mediante la generación de Javascript a partir de Java.
- **Spring**: generación de aplicaciones web empresariales
  - HTML5, REST, AJAX, soporte móviles
  - Mapeo O/R
  - Integración con redes sociales
  - Integración con sistemas basados en mensajería asíncrona.
- **Apache Struts**: framework para la generación de aplicaciones web
- **JUnit**: pruebas unitarias
- **JavaFX**: nueva estrategia de Oracle para GUI RIAs (rich interface applications)
- **Apache Lucene**: motor de búsqueda para recuperación de información.

## **Tema 9: Refactorización**

Proceso de mejora de la estructura interna de un sistema software tal que su comportamiento externo no varía. Consiste en realizar modificaciones como añadir argumentos a un método, mover atributos de una clase a otra, mover código hacia arriba o hacia abajo en una jerarquía de herencia, etc.

2 pasos:

- Introducir un cambio simple (refactorización)
- Probar el sistema tras el cambio.

2 reglas de oro:

- Antes de añadir una nueva funcionalidad a la aplicación, primero refactoriza el código para que el cambio sea fácil de introducir y luego añade la nueva funcionalidad.
- Antes de refactorizar, disponer de auto-pruebas (tipo JUnit)

Motivos para refactorizar:

- Mejorar el diseño del software.
- Hacer que el código sea más fácil de entender.
- Hacer que sea más sencillo de encontrar fallos
- Permite programar más rápidamente.
- Ahorra tiempo y aumenta la calidad del proyecto, y evita el duplicar código.

¿Cuándo?

- Al añadir un método o función
- Para arreglar fallos
- Al revisar código
- Cuando 'algo huele mal'
- Si vemos que hay código duplicado o métodos muy largos
- Si las clases son muy grandes
- Los métodos son muy largos o tienen muchos parámetros.
- Sentencias 'switch'

### ¿Cuándo NO?

- Si el código es muy malo
- Cuando el plazo se acaba

### Problemas

- Acoplamiento con bases de datos.
- Implica a menudo cambios en la interfaz de clase

**Test o pruebas unitarias:** son una forma de probar el correcto funcionamiento de un módulo de código. Resulta más útil escribirlas antes de escribir el código a probar, sirve para tener más claro el comportamiento de un método. Requisitos:

- Automatizable: no debería requerirse una intervención manual.
- Completas: deben cubrir la mayor cantidad de código.
- Reutilizables: no se deben crear pruebas que sólo puedan ser ejecutadas una sola vez. Útil para integración continua.
- Independientes: la ejecución de una no debe afectar a la ejecución de otra.
- Profesionales: deben ser consideradas con la misma profesionalidad, documentación, etc. que el código.

**Pruebas funcionales:** verifican una aplicación comprobando que su funcionalidad se ajusta a los requerimientos o a los documentos de diseño. Se trata a la aplicación como un todo.

### Técnicas de refactorización simples:

- Añadir un parámetro: un método necesita más información al ser invocado.
- Quitar un parámetro: un parámetro ya no es usado en un método.
- Cambiar el nombre de un método: el nombre del método no indica su propósito

### Técnicas de refactorización comunes:

- Mover un atributo: un atributo es o no debe ser usado por otra clase más que en la clase donde se define.
- Mover un método: un método es usado más en otro lugar que en la clase actual.
- Extraer una clase: una clase hace el trabajo que deberían hacer dos.
- Extraer un método: existe un fragmento de código, quizás duplicado, que se puede agrupar en una unidad lógica.
- Cambiar condicionales por polimorfismo: una estructura condicional elige entre diferentes comportamientos en función del tipo de un objeto.
- Cambiar código de error por excepciones: un método devuelve un valor especial para indicar un error.

### Técnicas de refactorización y herencia:

- Generalizar un método: existen dos o más métodos con idéntico comportamiento en las clases derivadas.
- Especializar un método: un comportamiento de la clase base sólo es relevante para algunas clases derivadas.
- Colapsar una jerarquía: apenas hay diferencias entre una clase base y una de sus derivadas.
- Extraer una subclase: algunas características de una clase son usadas sólo por un grupo de instancias.
- Extraer una superclase: hay dos o más clases con características similares.

- Convertir herencia en composición: el principio de sustitución no se cumple: una clase derivada usa sólo parte de la interfaz de la base o no desea heredar también los atributos.

## **Tema 10: Principios de Diseño Orientado a Objetos**

Forma de diseño para mejorar la calidad y el entendimiento del software.

**Objetivos:** reducir costes de desarrollo, plasmar adecuadamente lo que el cliente pide, acelerar el desarrollo y la calidad del software.

**Acoplamiento:** relación entre componentes del software. Interesa bajo acoplamiento, se hace asignando las tareas a quien sepa hacerlas.

**Cohesión:** grado en que las responsabilidades de un solo componente forman una unidad significativa. Interesa alta cohesión, asociando a un solo componente tareas que está relacionadas de cierta manera.

Componentes con bajo acoplamiento y alta cohesión facilitan su utilización e interconexión.

### **Principio Abierto-Cerrado:**

- Clases abstractas y sus derivadas están ‘cerradas’, pero tienen un conjunto ilimitado de comportamientos.
- El código cliente que usa la interfaz de la clase abstracta puede cerrarse a la modificación, pero abrirse a la extensión creando clases derivadas.
- Las aplicaciones son modificadas añadiendo nuevo código sin modificar el existente.
- Todos los atributos deben ser privados (encapsulación). No usar variables globales.
- Mejora reusabilidad y mantenimiento
- Requiere usar abstracción en los cambios.

### **Principio de sustitución (Liskov):**

- Los métodos que usan referencias a clase base deben poder usar objetos de clases derivadas sin saberlo.
- Si no se cumple Liskov, no se cumple el principio abierto-cerrado.
- Los métodos sobrescritos en las clases derivadas deben aceptar cualquier cosa que acepte el método base.

### **Principio de Inversión de Dependencias:**

- Los módulos de alto nivel no deben depender de los de bajo nivel. Ambos deben depender de abstracciones.
- Las abstracciones no deben depender de detalles específicos. Éstos deben depender de las abstracciones.
- Implica la separación de interfaz e implementación en distintos módulos.
- El diseño de frameworks está basado en este principio.

### **Principio de Segregación de Interfaz:**

- El código cliente no debe ser forzado a depender de interfaces que no usa.
- Los interfaces ‘gruesos’ conducen a acoplamientos entre clientes que de otra forma deberían estar aislados entre sí.
- Estos interfaces pueden segregarse en distintas clases abstractas o interfaces que rompen el acoplamiento no deseado entre los clientes.



### Principio de Responsabilidad Única:

- Nunca debe de existir más de una razón para que una clase cambie.
- Cada responsabilidad asociada a una clase es una razón para que la clase cambie
- Si una clase asume más de una responsabilidad, existe más de una razón para que la clase cambie.
- Si una clase asume 2 o más responsabilidades, éstas se acoplan (fragilidad)
- Éste principio es uno de los más simples y difíciles de cumplir correctamente.
- Unir responsabilidades es algo que hacemos de forma natural
- Encontrarlas y separarlas es el objetivo del diseño del software.