TEMA 3: CONCEPTOS FUNDAMENTALES DE ORIENTACIÓN A OBJETOS

1.- INTRODUCCIÓN

1.1. Evolución histórica

- ❑ Antes de los 60
  - El diseño y programación en aplicaciones era considerado como un arte.
  - No había método para el diseño ni la implementación de software.
  - No había métodos para mantener los programas.
  - Si algo iba mal, era mejor sustituirlo que revisarlo.
  - La mayoría del software escrito en esta época tuvo que ser posteriormente reescrito.
  - Esto llevo a la ***crisis del software***.

- ❑ Años 60 -. Metodologías
  - Surge la programación procedural.
    - *Aumentar la legibilidad.*
    - *Bajar costes de mantenimiento.*
    - *Acelerar el proceso de desarrollo.*
  - Programación estructurada.
    - *Utiliza un número limitado de estructuras de control que minimizan la complejidad de los problemas y por tanto reducen los errores. Se basa en el* ***teorema de estructura*** *que afirma que cualquier programa, por complejo que sea, puede escribirse utilizando tan sólo estas tres estructuras de control: secuencial, selectiva y repetitiva. Un programa estructurado por tanto, no contendrá instrucciones de salto.*
    - *Utiliza el diseño descendente o top-down que consiste en dividir el problema a resolver en subproblemas, a continuación éstos se descomponen en subproblemas más simples, y así sucesivamente hasta llegar a los problemas más pequeños y fáciles de resolver.*
    - *Abstracciones de datos.*
    - *Abstracciones de control.*
  - Programación modular.
    - *Software en módulos..*
    - *Reutilización de módulos.*
    - *Una aplicación es un conjunto de módulos.*
    - *La programación modular introduce el concepto de Tipo Abstracto de Datos. Los datos mpiezan a jugar un papel importante.*

- ❑ Los proyectos iban creciendo en tamaño.

- ❑ Abarcaban más y más campos.

- ❑ Un proyecto se realizaba por un equipo cada vez mayor de personas.
  - Cuanto mayor era el equipo, menor era el rendimiento individual.
  - La razón: la cantidad de esfuerzo en coordinación.

La **POO** (Programación Orientada a Objetos) nace con la idea de que el trabajo pueda ser dividido de forma coherente en pequeñas unidades independientes de manera que la necesidad de coordinación sea mínima.

Por ejemplo, si un diseñador hardware necesita construir una placa con concierta funcionalidad, lo primero que hace es estudiar los componentes que necesita y a continuación, si para cierta acción no existe un componente tiene dos alternativas: proponer a un equipo el diseño de un componente que resuelva el problema o producir el efecto deseado por otros medios. En cualquier caso, el trabajo del diseñador no es construir todos los componentes.

La POO divide a la comunidad de programadores en dos tipos:

- Los productores de componentes.
- Los consumidores de componentes.

Los equipos que crean componentes las someten a pruebas de depuración de manera que al usuario le llegan componentes en perfecto estado.

Los creadores de componentes ofrecen al usuario un interfaz por el que se pueden comunicar con el componente. Este interfaz impide conocer el interior del componente. Este permite que cambios internos en el componente no afecten a los consumidores de dichas componentes.

La misión del programador es conocer los componentes que existen en el mercado, y combinarlos adecuadamente para conseguir el fin.

1.2. ¿Qué es Orientado a objetos?

El software mantiene:

- ✓ Estructuras de datos (I)
- ✓ Comportamientos (II).

El término "Orientado a Objetos" significa que el software se organiza como una colección de objetos. Los objetos representan abstracciones del mundo real. Liga de forma robusta I y II, en oposición a la programación convencional que liga de forma débil I y II.

En este módulo nos limitaremos al campo de la programación, pero es también posible hablar de sistemas de administración de bases de datos orientados a objetos, sistemas operativos orientados a objetos, interfaces de usuario orientadas a objetos, etc.

Hoy en día no sólo la programación es orientada a objetos sino que el diseño, las bases de datos, etc., son también orientadas a objetos.

1.3. La consecuencia

Como resultado de usar la orientación a objetos nos encontramos con:

- Posibilidad de representar directamente las entidades del mundo real en los entornos informáticos, sin necesidad de deformarlos ni descomponerlos.
- Posibilidad de reutilizar y extender las aplicaciones existentes, por ejemplo, a partir de librerías especializadas y fácilmente modificables.
- Trabajo con entornos de desarrollo potentes, por ejemplo para la depuración y el seguimiento de ejecuciones.
- Disponibilidad de herramientas de comunicación hombre-máquina visuales de gran calidad.
- Facilidades de prototipado rápido de aplicaciones.
- Facilidad de utilización de paralelismo en el momento de la implementación.

2. FACTORES CRUCIALES QUE MIDEN LA CALIDAD DEL SOFTWARE

Antes de seguir avanzando, vamos a hacer un pequeño paréntesis para repasar técnicas que nos permiten mejorar significativamente la ***calidad*** de los productos software. Todos deseamos que nuestros sistemas de software sean rápidos, fiables, fáciles de usar, legibles, modulares, estructurados y así sucesivamente. Pero estos adjetivos describen dos tipos de cualidades diferentes:

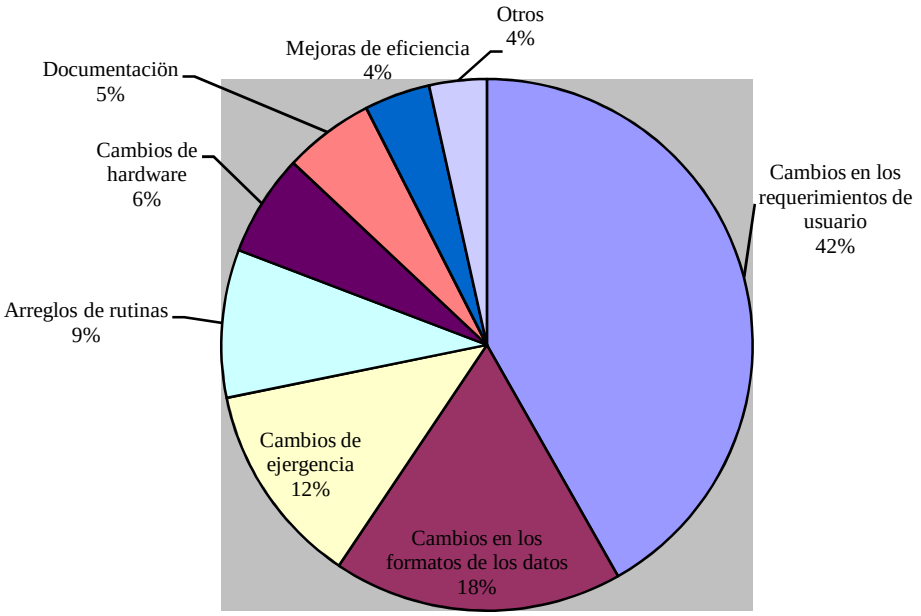
- **Factores externos**: perceptibles por el usuario (de cualquier nivel).
  - Corrección: habilidad de ajustarse a los requerimientos.
  - Robustez: habilidad de funcionar aún bajo condiciones anormales.
  - Extensibilidad: facilidad de adaptarse a cambios en los requerimientos.
  - Reutilidad: habilidad para ser reutilizado todo, o en parte, en nuevos desarrollos.
  - Eficiencia: habilidad en el uso óptimo de los recursos hardware.

- **Factores internos**: perceptibles por los desarrollardores del producto (profesionales de la informática que tienen acceso al código fuente).
  - Formalidad [Corrección + Robustez): debe ser fácil crear software que funcione correctamente y fácil garantizar que esto es así.
  - Modularidad [Reutilización + Extensibilidad]: debe crearse el menor software posible. El software debe ser fácil de modificar.

El método orientado a objetos, puede mejorar significativamente estos factores de calidad, por eso es tan atractivo. Sin embargo, hay que tener presente que el método orientado a objetos no es una panacea y que muchos de los problemas habituales de la ingeniería de software se mantienen. Ayudar a señalar un problema no es lo mismo que resolverlo.

Los factores anteriores no incluyen una cualidad que se menciona con frecuencia: *facilidad de mantenimiento*. Mantenimiento es lo que sucede después de que se ha distribuido un producto de software. Se estima que el 70% del coste del software se dedica al mantenimiento, distinguiendo por un lado la modificación en las especificaciones de los sistemas informáticos (siendo esta la parte más noble), y por otro lado (y no tan noble) la depuración a posteriori: quitar los errores que en principio nunca deberían haber existido.

La siguiente figura muestra el porcentaje de los costes de mantenimiento, destacar, que más de las dos quintas partes está dedicado a extensiones y modificaciones solicitadas por los usuarios (parte noble).



3. ¿QUÉ ES LA PROGRAMACIÓN ORIENTADA A OBJETOS?

Según Grady Booch, fabricante de la herramienta Rose para ingeniería de software orientada a objetos e impulsor del lenguaje unificado de modelado UML, define la progra*maci6n orientada a objetos (POO)* como:

> *«un método de implementación en el que los programas se organizan como colecciones cooperativas de objetos. cada uno de los cuales representan una instancia de alguna clase, y cuyas clases son todas miembros de una jerarquía de clases unidas mediante relaciones de herencia».*

Existen tres importantes partes en la definición: la programaci6n orientada a objetos
1) utiliza ob*jetos*, no algoritmos, como bloques de construcci6n 1ógicos *(jerarquía de objetos);*
2) cada objeto es una instancia de una clase, y
3) las clases se relacionan unas con otras par media de relaciones de herencia.

Un programa puede parecer orientado a objetos, pero si cualquiera de estos elementos no existe, no es un programa orientado a objetos. Específicamente la programaci6n sin herencia es distinta de la programaci6n orientada a objetos: se denomina *programación con tipos abstractos de datos o programación basada en objetos.*

El concepto de objeto, al igual que los tipos abstractos de datos o tipos definidos por el usuario, es una colección de elementos de datos, junto con las funciones asociadas para operar sobre esos datos. Sin embargo, la potencia real de los objetos reside en el modo en que los objetos pueden definir otros objetos. Este proceso, se denomina *herencia* y es el mecanismo que ayuda a construir programas que se modifican fácilmente y se adaptan a aplicaciones diferentes.

Los conceptos fundamentales de programación orientada a objetos son: ***objetos, clases***, ***herencia, mensajes*** *y* ***polimorfismo***.

4.- OBJETOS

Un objeto puede ser un número, una cola, un diccionario, un compilador, una ventana, fruta, carne, casas, cuadros de diálogo... Es decir un objeto pertenece al dominio del problema. Los objetos representan a los datos del problema real.

Representa un elemento individual e identificable, real o abstracto, con un comportamiento bien definido en el dominio del problema. Luego un objeto puede ser:

OBJETO = ESTADO + COMPORTAMIENTO + IDENTIDAD

El ***estado*** se refiere a las *propiedades* o *atributos* que caracterizan al objeto. Cada atributo debe tener un *valor* en algún dominio. Los valores de los atributos pueden variar a lo largo de la vida del objeto. Los atributos de un objeto son tenidos en cuenta según el dominio del problema. Los atributos de un objeto deben ser privados al objeto.

Por ejemplo, si quiero vender un coche, los atributos que interesaran son el precio, color, potencia, terminación,... pero si lo que quiero es participar en un rally, lo que interesa es aceleración, potencia, velocidad, anchura de ruedas,...

El ***comportamiento*** de un objeto viene determinado por la forma en la que el objeto interactúa con el sistema. La forma de actuar sobre un objeto es enviándole un *mensaje*. El mensaje activará un comportamiento del objeto (*método*) que será el que determine su forma de actuar. Los métodos son los comportamientos del objeto. Pueden generarse métodos para permitir consultas sobre aspectos internos del objeto, modificar atributos del objeto, envío de mensajes a otros objetos,...

La ***identidad*** del objeto es la propiedad característica que los distingue del resto de los objetos. Dos objetos con los mismos atributos y los mismos valores en sus atributos son iguales pero no idénticos. Para distinguir objetos se utilizan varias técnicas: utilizar dirección de memoria o referencia, utilizar nombres definidos por el usuario, utilizar claves identificadoras internas o externas. La identidad de un objeto permanece con él durante toda su vida. La identidad no se la proporciona el usuario, simplemente la tiene.

La implementación de un objeto:

- Mantiene una memoria privada que describe las propiedades del objeto (su estado). Sus atributos (variables de instancias, slot o datos miembro).
- Disponen de un conjunto de operaciones que actúan sobre dicha memoria privada. Son las que definen su comportamiento (métodos o funciones miembro).
- Un objeto tiene una identidad. (Normalmente una dirección).

La forma en la que un método actúa sobre un objeto es a través del envío de mensajes. En un mensaje intervienen el receptor (es el objeto que recibe el mensaje) y el selector (es el comportamiento que está involucrado en el mensaje).

El resultado esperado del envío de un mensaje dependerá:
- Del estado en que se encuentre dicho objeto.
- Del método involucrado en el mensaje.
- De la transformación que este mensaje pueda soportar.

Un método tiene total visibilidad sobre los atributos del objeto al cual le ha enviado el mensaje. Cuando a un objeto se le envía un mensaje, el método activado puede utilizar como variables los atributos del objeto receptor pudiendo incluso modificarlos.

El único camino para acceder al estado de un objeto debe ser por la activación de algún método, es decir, por el envío de un mensaje.

El conjunto de métodos que un objeto es capaz de responder es su protocolo o interfaz y define su conducta.

Ejemplo. Se desea realizar el siguiente experimento: de una urna que contiene inicialmente un número determinado de bolas blancas y otro número determinado de bolas negras, se pretende realizar lo siguiente:

```
Mientras en la urna quede más de una bola
    Sacar dos bolas de la misma
    Si ambas son del mismo color
        Introducir una bola negra en la urna
    Si ambas son de distinto color
        Introducir una bola blanca en la urna
Extraer la bola que quede y determinar su color.
```

Objeto: *Una urna*
Memoria privada del objeto: *Número de bolas blancas y Número de bolas negras.*
Interfaz (métodos):
- *sacaBola(): devolverá el color de la bola sacada. Decrementa en 1 el nº de bolas de ese color.*
- *meteBola(Color): incrementa en 1 el número de bolas del color dado.*
- *quedanBolas(): devuelve cierto si hay bolas en la urna.*
- *quedaMasDeUnaBola(): devuelve cierto si hay más de una bola en la urna.*
- *Int totalBolas(): devuelve el número total de bolas (privado).*

Crear un objeto es como crear una variable de tipo, por tanto tienen las mismas propiedades que los datos de cualquier tipo. Destruirlo debe ser igual. Además hay que inicializarlo. Todo esto es algo muy dependiente del lenguaje, ya se verá como hacerlo.

Mientras no se profundice en el estudio de Java, supongamos:

- Que todo se puede escribir en "Java".
- Que el envío de un mensaje a un objeto **u** con selector **m** se escribe por **u.m**.
- Que la manera de crear e inicializar un objeto de algún tipo, es simplemente creando una variable de ese tipo y que en la creación se le proporciona el valor inicial de sus variables de instancias como argumentos.
- Para crear el objeto urna **u** con 34 bolas blancas y 56 bolas se escribirá **Urna u(34,56).**

El ejemplo resuelto sería:

```
main()
{
   /* u es un objeto urna con 34 bolas blancas y 56 bolas negras */
   Urna u(34,56);
   char a,b;
   while (u.quedaMasdeUnaBola())
   {
        // en cada pasada, el número de bolas disminuye en uno
        a=u.sacaBola();
        b=u.sacaBola();
        if (a==b)
            u.meteBola('n');
        else
            u.meteBola('b');
   }
   System.out.println("La bola final es de color " + u.sacaBola());
}
```

5.- CLASES

Una clase describe el comportamiento de una familia de objetos Puede verse como un tipo (que además define métodos). Es una plantilla para crear objetos que tienen:

- ✓ Los mismos atributos.
- ✓ Responden a los mismos mensajes.

A los objetos de una clase se les suele llamar ***instancias de clase***.

A los atributos que describen la memoria privada de un objeto se les suele llamar ***datos miembro***, ***variables de instancia*** o ***slot***.

A los métodos o comportamientos se les suele llamar ***funciones miembro*** o ***métodos de instancia***.

En el ejemplo anterior, la definición de urna no define el comportamiento de una urna, sino de todas las urnas (se ha definido una clase). Si por ejemplo definimos una urna ***u*(34,56), *u*** es solo un ejemplo de ese comportamiento (un objeto Urna o una instancia de la clase Urna). El número de bolas blancas y el número de bolas negras son dos variables de instancia definidas en la clase, los valores 34 y 56 inicialmente son los valores de las variables de instancia que definen el estado de la urna *u*. Es evidente que podemos crear varias urnas.

Es evidente que podemos crear varias urnas, por ejemplo:

```
main()
{
    Urna u(34,67), v(89,23);

    u.meteBola('n');
    // u tiene ahora 34 blancas y 68 negras
```

```
    v.meteBola('b');
    // v tiene ahora 90 blancas y 23 negras
    ....
}
```

Ambas responden a los mismos mensajes con los mismos métodos pero su actuación depende de su estado que es particular.

La descripción de una clase podría ser:

```
Clase Urna
    Variables de instancia
        Número de bolas blancas (blancas)
        Número de bolas negras (negras)
    Métodos públicos
        sacaBola()
        meteBola(color)
        quedanBolas()
        quedaMasDeunaBola()
    Métodos privados
        totalBolas()
Fin Clase
```

Un objeto puede enviarse un mensaje a sí mismo, mediante la palabra ***this***, en algunos casos es posible suprimirlo, como se verá en temas posteriores.

```
int totalBolas()
{
    return (blancas + negras)
}

int quedaMasDeUnaBola()
{
    return (1 < this.totalBolas());
}
```

Ventajas de la utilización de clases:

- Cada clase puede ser creada de modo independiente.
- Cada clase puede probarse de modo independiente.
- Asegura la consistencia de los datos pues ofrece un interfaz para su manejo.
- La implementación queda escondida al usuario de la clase (lo mismo que la implementación de los enteros queda oculta a los que los usan).
- Puede variarse la implementación sin tener que cambiar los programas que las utilizan.
- Es altamente reutilizable.

En resumen:

- Todas las instancias de una clase responden al mismo conjunto de mensajes con los mismos métodos.
- Todas las instancias de una clase tienen las mismas variables de instancias pero cada una con sus valores.

6.- ECUACIÓN FUNDAMENTAL DE LA POO

PROGRAMACIÓN ORIENTADA A OBJETOS
=
TIPOS ABSTRACTOS DE DATOS + HERENCIA + POLIMORFISMO

6.1. Tipos abstractos de datos

Los tipos abstractos de datos permiten ***encapsulación*** (guardar conjuntamente datos y operaciones que actúan sobre ellos) y ***ocultación*** (proteger los datos de manera que se asegura del uso de las funciones definidas para su manejo).

Las ventajas que aportan son:
- ✓ Implementación escondida al cliente.
- ✓ Los TADs (Tipos Abstractos de Datos) se generan independientemente.
- ✓ Se pueden probar independientemente.
- ✓ Aseguran la consistencia de los datos.
- ✓ Aumentan la reutilización de código.

Las posibilidades de ocultación de la información dependen del lenguaje en el que se implemente. Hay lenguajes que ponen diferentes niveles de privacidad y otros que no pueden hacer las variables de instancia privadas. ***Una clase se implementa como un tipo abstracto de datos.***

Ejercicio 1: Definir el problema anterior con urnas de tres colores (blancas, negras y rojas) teniendo en cuenta que si saca dos blancas, mete una roja, si son iguales (menos dos blancas) se mete una igual y si son distintas se mete una blanca.

```
Clase Urna3
    Variables de instancia
        Número de bolas blancas
        Número de bolas negras
        Número de bolas rojas
    Métodos públicos
        sacaBola()
        meteBola(color)
        quedanBolas()
        quedaMasDeunaBola()
    Métodos privados
        totalBolas()
Fin Clase
```

El programa podría ser parecido a las siguientes líneas:

```
main() {
  Urna v(23,34,56);
  char a,b;
  while (v.quedaMasdeUnaBola())
   {
    // en cada pasada, el número de bolas disminuye en uno
    a=v.sacaBola();
    b=v.sacaBola();
    if (a==b)
        if (a==´b’)
            v.meteBola(‘r’)
        else
            v.meteBola(a);
    else
        v.meteBola(‘b’);
   }
   System.out.println(“La bola final es de color “ + v.sacaBola());

 }
```

6.2. Herencia

Es una técnica que permite incrementar la reusabilidad.

- Maneja eficientemente relaciones "..es como un ...".
  Ejemplos:
  - ✓ La clase Urna3 es como una Urna pero puede contener además bolas rojas.
  - ✓ Una cebra es como un caballo pero que tiene rayas.
- Crea nuevas clases a partir de generalizar o especializar otras clases ya existentes.

Para ello a la hora de crear una clase puede reutilizar parte de la conducta de otra clase. Esto se hace por medio de añadir, suprimir o modificar métodos, y añadir o suprimir variables de instancia. La clase así resultante sería una clase que es heredera de la inicial.

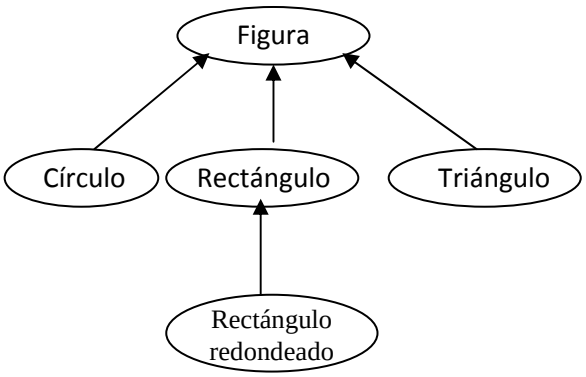
Si A hereda de B,

- A es hija de B, A es subclase de B, A es derivada de B.
- B es padre de A, B es superclase de A, B es ancestro de A y de sus subclases.

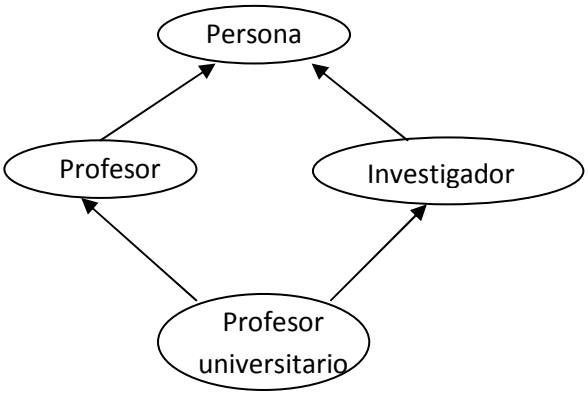
La herencia puede ser:

- **Simple:** Es aquel tipo de herencia en la cual un objeto (clase) puede tener sólo un ascendiente, o dicho de otro modo, una subclase puede heredar datos y métodos de una única clase, así como añadir o quitar comportamientos de la clase base.
- **Múltiple:** Es aquel tipo de herencia en la cual una clase puede tener más de un ascendiente inmediato, o lo que es igual, adquirir datos, y métodos de más de una clase.

Object Pascal, Smalltalk, Java y ***C#***, entre otros, sólo admiten herencia simple, mientras que ***Eiffel*** y ***C++*** admiten herencia simple y múltiple.



Herencia simple



Herencia múltiple

A primera vista, se pude suponer que la herencia múltiple es mejor que la herencia simple, sin embargo, no siempre será así. En general, prácticamente todo lo que se puede hacer con herencia múltiple, se puede hacer con herencia simple, aunque a veces resulta más difícil. Una dificultad surge con la herencia múltiple cuando se combinan diferentes tipos de objetos, cada uno de los cuales define métodos o campos iguales.

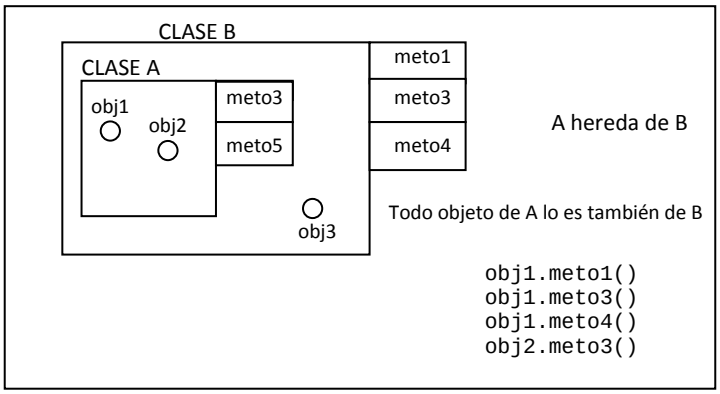
Supongamos dos tipos de objetos pertenecientes a las clases ***Graficos*** y ***Sonidos***, y se crea un nuevo objeto denominado ***Multimedia*** a partir de ellos.

- ***Graficos*** tiene tres campos de datos: *tamaño*, *color* y *mapaDeBits*, y los métodos *dibujar*, *cargar*, *almacenar* y *escala.*
- ***Sonidos*** tiene tres campos de datos: duración, voz y tono, y los métodos *reproducir*, *cargar*, *escala* y *almacenar*.

Así para un objeto ***Multimedia***, el método *escala* significa poner el sonido en diferentes tonalidades, o bien aumentar/reducir el tamaño de la escala del gráfico. Lo mismo ocurriría con los métodos almacenar y cargar.

El problema que se produce es la ambigüedad, y se tendrá que resolver con una operación de prioridad que el correspondiente lenguaje deberá soportar y entender en cada caso. En realidad, ni la herencia simple ni la herencia múltiple son perfectas en todos los casos, y ambas pueden requerir un poco más de código extra que represente bien la diferencias en el modo de trabajo.

La herencia clasifica a las clases en una jerarquía



Ejemplo: Sea Urna3 una clase que hereda las propiedades de la clase Urna. La clase Urna 3 *es como una* Urna pero que puede contener además bolas rojas.

```
Clase Urna3 hereda Urna
   Variables de instancia privadas
      Número de bolas rojas
Fin clase
```

- ✓ No se indican los métodos que hereda de la clase urna.
- ✓ Una vez definida, se usa como una clase más
- ✓ Una clase puede redefinir los métodos de la clase de la que hereda. La nueva definición puede apoyarse en la antigua.

Se puede redefinir el ejercicio que veíamos en apartado de clases basándonos en esta nueva definición de la clase Urna3.

Teníamos:

```
int totalBolas()
{
    return (blancas + negras)
}
int quedaMasDeUnaBola()
{
    return (1 < this.totalBolas());
}
```

Definimos totalBolas() en Urna3

```
int Urna3::totalBolas()
{
    return rojas + super.totalBolas();
}
```

El método quedaMasDeUnaBola no es necesario modificarlo. Hay que modificar los métodos sacaBola() y meteBola(). Su modificación puede hacerse haciendo referencia al padre (*super*).

Una **clase abstracta** es una clase que proporciona un interfaz **común** y **básico** a sus herederas. De ella no se obtendrá ninguna instancia. Definirá métodos con el cuerpo vacío o métodos con comportamientos generales a todas las subclases. Por ejemplo, supongamos que se define una clase Jugador que define las características generales de cualquier jugador:

6.3. Polimorfismo

El polimorfismo, en su expresión más simple, es el uso de un nombre o un símbolo, para representar o significar más de un acción. Así por ejemplo, en C el símbolo * se dice que está sobrecargado.

Dicho de otra forma es la capacidad de una entidad de referenciar distintos elementos en distintos instantes. Hay dos tipos de polimorfismo:

- Por sobrecarga de funciones.

  Dos funciones con el mismo nombre y distintos argumentos son funciones distintas. Igualmente dos funciones con el mismo nombre y con los mismos argumentos pero definidas en distintas clases son funciones distintas.

  Este tipo de polimorfismo es resuelto en tiempo de compilación. Permite especificar un mismo nombre a funciones que realizan una misma actividad en distintas clases. Objetos de distintas clases, pueden recibir el mismo mensaje, cada clase dispone de un método para aplicar a dicho mensaje.

- Por vinculación dinámica: paramétrico.

  En la vinculación estática, en tiempo de compilación se conoce el tipo de todas sus variables y expresiones. En la vinculación dinámica:

  - En tiempo de compilación no puede determinarse el tipo de algunas variables o expresiones
  - El compilador puede que no sepa a qué clase pertenece un objeto antes de la ejecución.
  - Por tanto hasta tanto no se determine la clase a la que pertenece el objeto, no podrá sustituir el cuerpo. Por tanto, esta vinculación tiene que hacerse en tiempo de ejecución.

7.- MÉTODOS Y VARIABLES DE CLASE

Puede que nos interese disponer de una información común a todos los objetos de una clase.

- ✓ **Variable de clase:** una variable común compartida por todos los objetos de una clase.
- ✓ **Métodos de clase:** métodos definidos para la clase que controlan el acceso a las variables de clase.

8.- LENGUAJES

Los lenguajes orientados a objetos lo podemos clasificar en tres grandes grupos:

- Puros (Smalltalk, Eifell, Java, Actor, etc.).
- Extensiones (C++, Object Pascal, Cobol, etc.).
- Códigos portables (Java).

Criterios de la POO

Nivel 1: Estructura modular basada en objetos. El sistema está modularizado en función de las estructuras de datos.

Nivel 2: Abstracción de datos. Los objetos están descritos como tipos abstractos de datos.

Nivel 3: Recolección automática de basuras. Los objetos no referenciados deben ser recolectados automáticamente por el sistema.

Nivel 4: Clases y módulos. Coherencia sintáctica.

Nivel 5: Herencia. Una clase puede ser definida como extensión o restricción de otra.

Nivel 6: Polimorfismo. Debe permitirse que las entidades se refieran a objetos de más de una clase.

Nivel 7: Herencia múltiple. Debe permitirse declarar una clase como heredera de varias clases.

	1	2	3	4	5	6	7
ADA	X	X	X				
C++	X	X		X	X	X	X
SMALLTALK	X	X	X	X	X	X	
EIFELL	X	X	X	X	X	X	X
JAVA	X	X	X	X	X	X	(*)

(*) Incluye el concepto de interfaz

ACTIVIDADES

1. ¿Qué metodologías surgen en los años 60? ¿Qué aporta cada una? ¿Por qué surgen?
2. ¿Qué ocurría antes de que surgiera la POO cuando un proyecto era realizado por un gran número de personas?
3. ¿Con qué idea nace la POO?
4. ¿Qué aporta que los programadores se dividan en productores y consumidores?
5. La POO liga de forma robusta o débil a las estructuras de datos y a los comportamientos de estos?
6. ¿Cuáles son las consecuencias de utilizar la POO?
7. Factores internos y externos que miden la calidad del software.
8. ¿Cuánto se suele dedicar al mantenimiento del software?
9. ¿Cómo define Grady Booch la POO?
10. ¿Cuáles son los conceptos fundamentales de la POO?
11. Un objeto nos dice que es un ***estado*** más un ***comportamiento*** más una ***identidad***. Explica que significan estos tres conceptos.
12. Explica que es una Clase y una Instancia de clase.
13. Ventajas de la utilización de clases.
14. Ecuación fundamental de la POO
15. ¿Qué nos aportan los TAD's? ¿Es lo mismo un TAD que una clase?
16. ¿En qué consiste la herencia? Tipos de herencia, explícalos.
17. ¿Qué es una clase abstracta?
18. ¿Qué has entendido que es el Polimorfismo?
19. Explica que es una variable de clase y un método de clase.