

Introducción a la Programación y los Lenguajes Orientados a Objetos

Objetivos de aprendizaje	1
Introducción y breve historia de la POO y los LPOO	1
Definición de POO y caracterización de los LPOO.....	2
1. La base de objetos y clases	3
2. El encapsulamiento y la ocultación de información.....	5
3. Las relaciones entre objetos: la agregación y la herencia.....	5
4. El ligamiento dinámico y el polimorfismo	10
5. Interacción basada en mensajes a objetos	12
Ejemplificación a través de un programa OO y ejercicios	13
Bibliografía recomendada para el alumno	13

Introducción a la Programación y los Lenguajes Orientados a Objetos

Objetivos de aprendizaje

Se pueden identificar tres objetivos fundamentales de esta lección para el alumno:

1. Introducir el paradigma de la POO como el principal en la actualidad.
2. Destacar sus características y profundizar en los conceptos básicos subyacentes.
3. Estudiar muestras de código en Java como el LPOO elegido para la realización de la práctica y de este modo conectar las partes práctica y teórica del tema.

Introducción y breve historia de la POO y los LPOO

En un sentido general se puede considerar la **orientación a objetos** como un marco para la ingeniería del software basado en objetos y clases. Abarca desde los principios del análisis de un problema hasta el final de su implementación y su dominio de aplicación también es muy amplio. Según varios autores, el interés por la OO surgió en el contexto de la crisis del software de los años 70 (la falta de reusabilidad de software). Al hablar de la OO, se suelen identificar las siguientes ventajas:

Ventajas de la OO

- Desarrollo rápido de sistemas.
- Mejora en la calidad y legibilidad del código.
- Facilidad de mantenimiento.
- Aprovechamiento del poder de los LPOO.
- Reusabilidad de software y diseños.
- Producción de sistemas más resistentes al cambio.

La POO y los LPOO juegan un papel importante dentro de las tecnologías OO. Según la literatura, el término **objeto** emergió paralelamente en varios campos de la Informática a principios de los años 70, para hacer referencia a nociones superficialmente distintas aunque relacionadas. La identificación de la importancia de la composición de sistemas en niveles de abstracción, la ocultación de información y el desarrollo de mecanismos de tipos de datos abstractos en los años 70 tuvieron una gran influencia en el desarrollo de la POO, aunque existe cierta polémica sobre como exactamente estos avances dieron lugar a lo que hoy en día se considera como POO. El LPOO Simula apareció en 1962 (y más tarde Simula67 en 1967) y, aunque no fue muy utilizado, ha sido reconocido como el primer LPOO, incorporando los conceptos de clase y objeto.

El concepto de **POO** propiamente dicho fue presentado por Alan Kay, uno de los inventores de Smalltalk (el primer LPOO popular), algunos años más tarde:

Caracterización de POO

- Todo es un objeto que almacena datos y al que se le puede hacer peticiones.
- Un programa es un conjunto de objetos que intercambian mensajes.
- Cada objeto tiene su propia memoria que está compuesta por otros objetos.
- Cada objeto tiene un tipo de mensajes que puede recibir y procesar.
- Todos los objetos de un cierto tipo pueden procesar los mismos mensajes.

Se trata de una caracterización muy general que no se puede aplicar a muchos de los LPOO más utilizados hoy en día. Smalltalk tenía estas características y fue concebido con el objetivo de ser un LPOO dinámico, que permitiera la adición de nuevas clases, objetos y comportamiento sobre la marcha. En las actas del congreso HOPL II editadas en 1993 por la ACM (*Association of Computing*

Machinery), Kay definió la POO en términos de una célula que permite el flujo de información en las dos direcciones, pero en la cual lo que está dentro está oculto desde fuera. En 1985 apareció el LPOO Eiffel, diseñado para mejorar la productividad y calidad de programas OO, pero no fue muy utilizado.

Para que la POO se estableciera como un paradigma era necesario que los programadores lo adoptaran. Por eso fue muy efectiva la modificación de un LP ya existente para incorporar los conceptos (y beneficios) de la POO, sin perder la posibilidad de reutilizar código fuente, como ocurrió con C++ (que es una extensión de C que incluye los conceptos OO). Otros LP han sido expandidos también para incorporar estos conceptos: Modula2 se convirtió en Modula3, Ada en Ada95, Lisp en CLOS (*Common Lisp Object System*) – vía Flavors, COBOL en Object COBOL, etc. Como ejemplos de LPOO de nueva creación se pueden destacar Python, Java y C#. Actualmente se pueden identificar unos 140 LPOO que se usan de alguna forma u otra.

Otra manera de ver la POO es como la evolución natural de la programación imperativa, desde la programación sin estructura, pasando por la programación procedimental y modular. En primer lugar, la **programación sin estructura** es la más sencilla y cada programa consiste en una secuencia de instrucciones que operan sobre datos globales o comunes a todas las partes del programa. Lo que ocurre es que, según va creciendo el programa, van surgiendo problemas. Por ejemplo, si se necesita la misma secuencia de instrucciones en varias partes del programa, hay que copiarla. Para evitar este problema, se empezaron a extraer estas secuencias, a darles un nombre, y a ofrecer una técnica para llamarlas y devolver el flujo de control desde ellas al programa principal junto con los resultados. Así aparecieron los procedimientos y funciones y la **programación procedimental**. Con la incorporación del paso de parámetros y procedimientos dentro de otros, se podían escribir programas con más estructura y menos probabilidades de errores. Así, en vez de ver un programa como una secuencia de instrucciones, se podía contemplar como una secuencia de llamadas a procedimientos. La extensión natural de este tipo de programación consistió en agrupar en módulos procedimientos comunes a varios programas, y así surgió la **programación modular**. En ella cada módulo tiene sus propios datos y estado, que se modifican con las llamadas al módulo. Por último, como se verá a continuación, la **POO** soluciona algunos de los problemas de la programación modular: puede haber simultáneamente múltiples versiones de un mismo objeto y cada una es responsable de su propia creación y destrucción.

Definición de POO y caracterización de los LPOO

Es muy difícil definir la POO y listar todos los LPOO existentes porque hay diferencias de opinión sobre lo que significa exactamente el término POO y sobre cuáles son las características de un lenguaje de este tipo. Sin entrar en este debate, basta proporcionar una definición general de POO y a continuación describir las características principales que se suelen relacionar con los LPOO, para comprender sendos conceptos¹:

Definición de POO

Método de implementación en el que los programas están organizados como colecciones de objetos, donde cada uno es una instancia de alguna clase, y donde todas las clases son miembros de una jerarquía de clases conectadas por relaciones de herencia.

Una vez definida la POO se pueden identificar las características principales que definen a los LPOO según la mayoría de los autores:

¹ El significado de los términos que se incluyen tanto en la definición como en la caracterización se explican en las secciones siguientes.

1. La base de objetos y clases
2. El encapsulamiento y la ocultación de información
3. Las relaciones entre objetos: la agregación y la herencia
4. El ligamiento dinámico y el polimorfismo
5. La interacción basada en el intercambio de mensajes entre objetos

A continuación se procede a explicar cada una de estas cinco propiedades.

1. La base de objetos y clases

Objetos

Los **objetos** son la base de la POO. El mundo está lleno de objetos: el perro, la mesa, la televisión, la bicicleta, etc. y todos ellos tienen dos características: **estado** y **comportamiento**. Por ejemplo, los perros tienen estado (raza, edad, sexo, color, nombre, etc.) y comportamiento (ladran, muerden, saltan, mueven la cola, etc.). También las bicicletas tienen estado (marca, número de marchas, etc.) y comportamiento (frenan, aceleran, etc.).

Definición de objetos de software

Modelos de los objetos en el mundo real, que también tienen estado, representado por las variables, y comportamiento, representado por los métodos (**método** es una función asociada a un objeto), además de identidad.

Además, se puede usar los objetos de software para representar conceptos abstractos, como por ejemplo, el evento generado por el sistema operativo cuando un usuario mueve el ratón o presiona una tecla del teclado.

Clases

Se puede pensar en una **clase** como una plantilla, y en un objeto, como una **instancia** de la clase.

Definición de clase

Representación de una estructura de datos abstracta junto con las operaciones que se pueden realizar con ella.

Dentro de lo que es el estado del objeto cabe distinguir entre las **variables de instancia** (datos distintos en cada instancia; por ejemplo, el color de la bicicleta) y las **variables de clase** (datos comunes a todas las instancias; por ejemplo, el nombre de la empresa que construye todas las bicicletas BH). También debe hacerse la distinción entre **métodos de instancia** (que trabajan con el estado de una instancia de la clase) y **métodos de clase** (que funcionan igual en todas las instancias).



La clase Bicicleta sería la plantilla para representar todos los tipos de bicicleta y una instancia sería la bicicleta de alguien, con marca, color, suspensión o no, y número de marchas específicas. Y aunque todas las instancias de las bicicletas proceden de la misma clase, son objetos independientes con estados distintos.

```
public class Bicicleta {
    private int no_ruedas = -1;
    private int marchas = -1;
    private static String marca = "Zipe";

    public Bicicleta(int nr, int nm){
        no_ruedas = nr;
        marchas = nm;
    }

    public int getNoRuedas(){
        return(no_ruedas);
    }


    public static String getMarca(){
        return(marca);
    }
}
```

Variables de instancia

Variable de clase

Método de instancia

Método de clase



De la misma manera que las empresas pueden aprovechar las características de las bicicletas a la hora de construir otras nuevas (no sería eficiente empezar desde cero cada vez que se quiere producir un nuevo modelo), en la POO se puede aprovechar el hecho de que los objetos son de una cierta clase para crear otros nuevos.

Como una clase describe un conjunto de objetos con características y comportamientos idénticos, se puede pensar en una clase como si fuera un tipo. La diferencia estriba en que un programador define una clase para un problema en concreto y no está obligado a usar un tipo existente, vinculado más bien con la estructura de la máquina que con el problema. Una vez definida una clase, debe representar una unidad de código útil que se pueda volver a usar en el futuro. No es fácil de conseguir al principio, pero con la experiencia se pueden producir clases así.

Objetos vs. clases

En algunos círculos se usa el término **objeto** solamente para hacer referencia a las instancias de una clase y en otros, a la clase en sí. Sin embargo, para que pueda existir una clase como instancia de otra, debe introducirse el concepto de **metaclass**.

Definición de metaclass

Clase especial que se instancia para producir una clase. Cada metaclass no tiene más que una instancia que es la propia clase.

Se pueden distinguir cuatro tipos de relaciones entre los términos objeto, clase y metaclass en los LPOO:

Comparación objeto/ clase/metaclass	Tipo	Elementos	Definición	LPOO
	1	Objeto	Se considera las clases como objetos	Self
	2	Objeto Clase	Los objetos son instancias de las clases pero no se puede acceder a las clases desde los objetos	C++
	3	Objeto Clase Metaclass	Los objetos son instancias de las clases y las clases de las metaclases. Se puede acceder a las clases desde los objetos	Java
	4	Objeto Clase Metaclass Metaclass clase	Los objetos son instancias de las clases y las clases de las metaclases. Se puede acceder a las clases desde los objetos. Aquí (a diferencia del tipo 3) la clase de una metaclass no es sí mismo.	Smalltalk

Info. y ejemplos Bernard (2000), capítulo 2 de Budd (1994) y Wegner (1987)

2. El encapsulamiento y la ocultación de información

Se suele confundir los conceptos de **ocultación de información** y **encapsulamiento**, en parte porque los autores más conocidos del mundo de la POO tienen definiciones e interpretaciones distintas y solapadas. Se proponen las siguientes:

Definición de encapsulamiento

Manera en que se acoplan los datos y comportamientos de un objeto dentro de él, mostrando una interfaz común.

Definición de ocultación de información

Principio según el cual fuera del objeto sólo se muestran los detalles mínimos necesarios para poder iniciarlo y acceder a él.

Como puede verse, si la ocultación de información fuera lo mismo que el encapsulamiento, entonces todos los datos y comportamientos encapsulados en un objeto estarían también ocultos, y normalmente no es así.

El encapsulamiento y la ocultación de información aportan dos ventajas a los desarrolladores de software:

Ventajas del encapsulamiento y la ocultación de información

- Modularidad: Se puede escribir y mantener el código para un objeto aparte del código para los demás.
- Encubrimiento de detalles de implementación: Como el objeto tiene una interfaz pública que otros objetos pueden usar para comunicarse con él, puede mantener información privada y proteger métodos cuyos detalles de implementación pueden cambiar.

Info. y ejemplos Capítulo 3 de Budd (1994) y capítulo 5 de Pratt y Zelkowitz (1998)

3. Las relaciones entre objetos: la agregación y la herencia

Un objeto solo no es muy útil, sino que lo es cuando está relacionado con otros objetos. En general puede haber cuatro tipos de relaciones entre los objetos: relaciones de **instanciación**, **asociación**, **agregación** y de **herencia**. Se ha visto antes que un objeto es una instancia de alguna clase y, por lo tanto, las relaciones de instancia son del tipo: "es un/a". Un objeto está asociado con otro si lo usa a través de mensajes que le envía; por eso se llaman relaciones de uso o asociación. Aquí se tratan los otros dos tipos de relaciones.

Definición de agregación

La manera en que se puede construir una clase u objeto en términos de otras clases u objetos ya existentes.

Las relaciones de agregación son relaciones del tipo: "tiene un/a" (en el caso de un coche: tiene un motor, suspensión, etc.). Se puede distinguir entre dos tipos de objetos formados por agregación: un objeto **monolítico** y uno **compuesto**:

Comparación entre objetos monolíticos y compuestos

- El **monolítico** no parece desde fuera que se ha construido a partir de otros objetos (p.ej., un botón de radio en una interfaz visual).
- El **compuesto** parece estar construido de otros objetos en el sentido de que se puede acceder a su estructura desde fuera. Se puede distinguir entre dos variantes:
 - El compuesto **heterogéneo**, que está compuesto por objetos distintos (p.ej., un objeto **fecha** puede tener un objeto **día**, otro **mes**, **año**, etc.)
 - El compuesto **homogéneo** esta compuesto por objetos iguales (p.ej., una lista de objetos de **dirección**)

Se puede saber bastante acerca de un objeto por su clase. Con el ejemplo de la bicicleta, aunque uno no sepa qué es una “penny-farthing”², si sabe que es un tipo de bicicleta, sabrá que tiene ruedas, manillar, sillín y pedales. De forma similar, la POO permite que se definan las clases en términos de otras clases. Por ejemplo, las bicicletas de montaña, de carrera y tándems, son todos tipos de bicicleta y, por lo tanto, según la terminología OO son **subclases** de la clase **Bicicleta**. Y **Bicicleta** es su **superclase**. La herencia representa relaciones de: “es un tipo de”.

Definición de herencia

La manera en que se puede definir una clase u objeto como extensión de otra clase u objeto.

Todas las subclases heredan el estado (en términos de las declaraciones de variables) y los métodos de la superclase. Aun así, las subclases no se limitan al estado y comportamiento dado por la superclase, sino que pueden añadir nuevas variables y métodos a los que hereden. Por ejemplo, un tándem tiene dos sillines y dos pares de pedales, y una bicicleta de montaña tiene neumáticos gruesos.

Ejemplo

```
public class PennyFarthing extends Bicicleta {
    private String FechaFabrica = "1903";

    public PennyFarthing(){
        super(2,0);
    }
    // etc.
}
```

Una relación de herencia se expresa en Java con la palabra clave **extends**

Se usa el método **super** para llamar al constructor de la superclase



Las subclases también pueden **sobrecargar** los métodos heredados para proporcionar implementaciones especializadas de estos métodos. Por ejemplo, en el caso de una bicicleta de montaña con un conjunto de marchas distinto del de la bicicleta de paseo, será necesario sobrecargar el método **cambioDeMarcha** para que el ciclista pueda utilizar las nuevas marchas.

No hay límite en cuanto a las capas de herencia; el **árbol de herencia** o **jerarquía de clases** puede ser tan profunda como sea necesaria. Y los métodos y variables son heredados en los niveles inferiores. Y en general, cuanto más abajo en la jerarquía esté la clase, más especializado será su comportamiento.

La mayoría de los LPOO ofrecen **herencia de clases**, pero hay otros, como Self y JavaScript, que permiten **herencia de objetos**, e incluso hay lenguajes como Python y Ruby que permiten las dos formas. Aquí se tratará la herencia de clases porque es la más común, pero cabe mencionar brevemente que la herencia de objetos permite extender el comportamiento de objetos en el tiempo de

² Modelo de bicicleta inventado entre 1870 y 1875; véase www.inet-shibata.or.jp/~HSbicycles/HISTORY1.htm.

ejecución. Dentro de la herencia de clases hay que distinguir entre la **herencia de implementación y de sub-tipo**:

Comparación entre las herencias de clase

- La **herencia de sub-tipos** (también conocida como herencia de interfaz) es la más común y en este modelo una sub-clase se considera como un sub-tipo de su superclase, lo que se suele llamar una **relación "ES-A"**. Aquí se considera un objeto como del tipo de su clase o de cualquier superclase suya. Por ejemplo, la clase **Círculo ES-A Forma**, así que en cualquier lugar del programa donde se use **Forma**, se puede usar **Círculo** también. Este criterio se puede aplicar a todos los LPOO con un sistema de tipos estático, porque es un rasgo que usa el compilador para comprobar que la utilización de tipos en un programa es correcta.
- La **herencia de implementación** permite que una clase herede una parte o toda su implementación de otra clase. Por ejemplo, se podría implementar una clase para una pila utilizando una matriz heredada de la clase **Matriz**. Así, la clase **Pila** podría utilizar cualquier rasgo de **Matriz** para su propia implementación. Con la **herencia de implementación pura**, el hecho de que la clase **Pila** herede su implementación de **Matriz** no sería visible al código en **Pila**, sino que sería solamente un detalle de implementación.

La distinción más importante que se puede mencionar entre la utilización de la herencia por los LPOO es la existente entre la **herencia simple** y la **múltiple**. El segundo tipo permite la herencia de más de una clase a la vez cuando se define una clase nueva. Por ejemplo, puede haber una clase que defina las características y comportamientos de los motores, así que se podría definir una nueva clase **Motocicleta**, que sería una subclase de la clase **Bicicleta** y también de la clase **Motor**.

La herencia múltiple puede parecer que sea un rasgo esencial para un LPOO para casos como el ejemplo que se acaba de presentar. Pero la herencia múltiple introduce algunas complicaciones en el LPOO que la ofrece: problemas como los conflictos de nombres de clases y las ambigüedades requieren mecanismos en un LPOO para evitarlos, lo que a su vez añade complejidad al lenguaje. Eiffel es un lenguaje que se conocido por la manera en que proporciona herencia múltiple en términos de un control estricto sobre la selección de los rasgos que se pueden heredar y los cambios de nombre necesarios. C++, en comparación, no proporciona herencia múltiple con la misma flexibilidad, lo que puede llevar a algunos a pensar que este tipo de herencia es peligrosa e innecesariamente compleja.

Ya se ha podido ver que el tema de la herencia múltiple es más sutil que la mera elección de si un LPOO la proporciona o no. Un lenguaje puede ofrecer diferentes formas de herencia simple, como por ejemplo, la de sub-tipos y la de implementación. Se ha visto que tanto C++ como Eiffel tienen los dos tipos, tanto en la forma simple como en la múltiple. Y Java, por ejemplo, no proporciona la herencia múltiple de ninguna forma pero sí la simple, de dos formas:

1. La **extensión** de clases (utilizando la palabra clave *extends*) que permite una combinación de la herencia de implementación y de sub-tipos.
2. La implementación de una **interfaz** (utilizando la palabra clave *interfaz*) que ofrece la herencia de sub-tipos.

Cabe mencionar que la herencia de sub-tipos tiene mucha menos importancia en los LPOO dinámicos porque el control de tipos no es generalmente relevante, así que se prefiere la herencia múltiple de implementación sobre la herencia múltiple de sub-tipos (aun así, la mayoría de estos lenguajes consideran que cualquier clase que hereda de otra es su sub-tipo). La herencia ofrece los siguientes beneficios:

Ventajas de la herencia

- Proporciona una manera natural de modelar el mundo y la relación de los objetos que existen en él.
- Las subclases exportan un comportamiento especializado a partir de una base común de elementos proporcionados por las superclases. A través de la herencia, se puede reutilizar el código de las superclases repetidas veces.
- Dependiendo del LPOO, se pueden implementar superclases que definen comportamiento genérico. Así se puede implementar parcialmente el comportamiento dejando la mayor parte sin definir para su especialización en las subclases.

Finalmente, hay que distinguir entre la **herencia estática**, que es lo que se suele encontrar en la mayoría de los lenguajes, y la **herencia dinámica**, que permite cambiar la **clase padre**³ en el tiempo de ejecución. Aunque no se suele ver ésta en muchos LPOO, puede ser útil cuando el conjunto de características de un objeto (o clase) cambie radicalmente durante su vida. Por ejemplo, un objeto ventana tendrá un conjunto de características cuando esté abierta, y otras distintas cuando está minimizada como un icono.

Clases genéricas o parametrizadas

La conversión de un objeto de una clase base a una más abajo en la jerarquía de clases (lo que se llama **especialización** [*downcasting*]), lo cual se hace con bastante frecuencia al sacar objetos de contenedores genéricos, es una operación costosa y también potencialmente peligrosa, porque incluso en los LPOO con comprobación estricta de tipos no se pueden garantizar los contenidos del contenedor. Una solución es utilizar las **clases genéricas o parametrizadas**, que se refieren a una manera de parametrizar una clase con tipos de datos específicos. Un ejemplo típico es la clase **Pila**, que es parametrizada por el tipo de los elementos que contiene. Así se puede comprobar que no hay problemas de tipo a la hora de compilar la clase **Pila** y que permanece suficientemente genérico para tratar cualquier tipo de elementos.

La ventaja de los tipos parametrizados es que permiten que los LPOO con un sistema de tipos estáticos mantengan su seguridad de tipos al compilar los programas, manteniendo casi la misma flexibilidad que los LPOO dinámicos.

Se puede decir que la ausencia de clases genéricas en Java representa una grave limitación en su sistema de tipos. En realidad, la mayoría de objetos en un programa existen dentro de algún tipo de contenedor, y como en Java los contenedores no tienen tipo, se puede decir que su sistema de tipos no ofrece ninguna ventaja sobre las alternativas dinámicas.

Evidentemente, los LPOO dinámicos no requieren tipos parametrizados para permitir la programación genérica. Los tipos se comprueban en el tiempo de ejecución.

Clases abstractas e interfaces o protocolos de interfaces

Las **clases abstractas** y las **interfaces** son dos maneras de especificar de antemano una parte del comportamiento de una clase.

Definición de clase abstracta

Clase que especifica algunos de los métodos (tanto los nombres como el tipo de resultado que devuelven) que hay que implementar para usarla.

³ La clase directamente superior en la jerarquía.

Definición de interfaz⁴

Zona (o dispositivo⁵) que entidades no relacionadas pueden usar para comunicarse o interactuar.

No se pueden implementar directamente las clases abstractas sino que hay que heredar de ellas. Son abstractas también las clases que hereden de las abstractas y no implementen todos los métodos declarados allí.

En cuanto a las interfaces, por ejemplo, un lenguaje natural es una interfaz entre seres humanos, las reglas de comportamiento entre dos equipos de fútbol es una interfaz que permite la realización de un partido y un mando a distancia es una interfaz entre una persona y una televisión.

En la POO un **protocolo de interfaz** (o simplemente **interfaz** en el caso del LPOO Java) es un mecanismo que objetos no relacionados pueden usar para interactuar entre ellos. Por ejemplo, la clase **Bicicleta** va a definir lo que puede hacer una bicicleta pero solamente en términos de su comportamiento habitual (es decir, como funcionan normalmente las bicicletas). Pero una bicicleta puede interactuar con el mundo de muchas otras maneras que no tienen nada que ver con su función principal.

Ejemplo

Se pueden almacenar bicicletas en un almacén y controlarlas a través de un programa de gestión. A dicho programa no le importa si gestiona bicicletas, lápices o portaaviones; lo único que quiere de cada objeto es que aporte cierta información (precio, número de serie, etc.).

Por lo tanto, en vez de imponer relaciones entre clases de objetos que son fundamentalmente distintos (y potencialmente no conocidos a la hora de escribir el programa de gestión), lo que se hace es establecer un protocolo en la forma de una interfaz que defina (pero no implemente) los métodos que necesita tener cualquier objeto para funcionar con el programa de gestión (por ejemplo, métodos para fijar y ver el precio, asignar y ver el número de serie, etc.). Por lo tanto, si se quiere usar una bicicleta como objeto de gestión para este programa, hay que implementar la interfaz relevante, es decir, tener métodos que controlen el precio y la asignación de un número de inventario, etc.

Las interfaces sirven para definir un protocolo de comportamiento para cualquier punto en la jerarquía de clases y, en general, se pueden ver las siguientes ventajas de las interfaces para un programa OO:

Ventajas de las interfaces

- Captan las similitudes entre clases no relacionadas sin forzar relaciones entre ellas.
- Declaran los métodos que las clases tienen que implementar para pertenecer a un cierto protocolo de comportamiento.
- Revelan la interfaz de programación de un objeto sin revelar su clase.

Se puede ver que las clases abstractas permiten la implementación del código para un método si se quieren controlar los aspectos de implementación, ya que una interfaz no puede incluir detalles de implementación.

⁴ Definición del Diccionario de la Real Academia Española.

⁵ Extensión de la definición por parte del autor de esta lección.

4. El ligamiento dinámico y el polimorfismo

Un método con **ligamiento dinámico** se denomina un **método virtual**. El principio del ligamiento dinámico se usa para implementar el **polimorfismo**. Véanse las definiciones de estos conceptos:

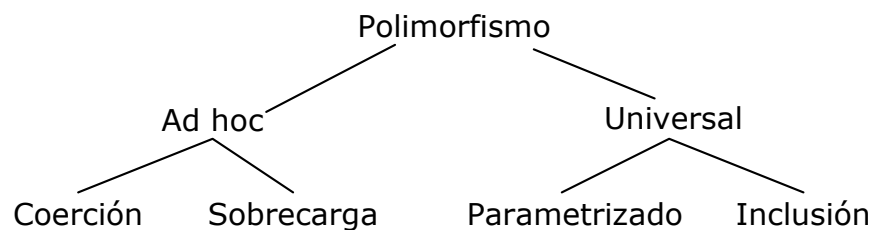
Definición de ligamiento dinámico

Resolución del método que se usa en el momento de ejecución basada en los tipos de los parámetros y la clase del objeto (contrasta con el **ligamiento estático**, en el que la resolución tiene lugar durante la compilación del programa basado en el tipo de la referencia).

Definición de polimorfismo

En términos generales, capacidad de una entidad de tener varias formas según el contexto. Específicamente en la POO, habilidad de definir un método en clases distintas según sus necesidades.

A la hora de trabajar con jerarquías de clases, muchas veces es deseable tratar un objeto de una clase específica como uno de la clase base (llamado **generalización** [*upcasting*]), porque así no es necesario escribir código que depende de una clase concreta. Es una técnica muy potente, porque nos permite usar el polimorfismo para llamar a un método (que subsume de alguna forma todas las clases en la jerarquía) y el objeto se comportará de una manera coherente con su estructura, sin que tengamos que comprobar la estructura explícitamente. Por ejemplo, un objeto que sepa dibujarse en una interfaz visual. Si no fuera por esta técnica, tendríamos que añadir código explícito para hacer la comprobación (por ejemplo: `if` o `case`) e implementar cada versión correspondiente, lo que no produciría código muy robusto porque estaría afectado por la adición de nuevas clases. A continuación se puede ver una clasificación del polimorfismo:



Comparación entre los tipos de polimorfismo

- El **polimorfismo ad hoc** hace referencia a métodos que funcionan con varios tipos distintos que no tienen que tener una estructura común, y que pueden producir resultados distintos para cada tipo. Se pueden identificar dos tipos:
 1. **Coerción:** operación semántica que convierte implícitamente un argumento en el tipo esperado en una ocasión que produciría un error. Se puede implementar con técnicas estáticas o dinámicas. Por ejemplo, un método que requiere un número entero y que recibe un número real, haría la conversión.
 2. **Sobrecarga:** técnica sintáctica según la cual se usa el mismo nombre de variable para denotar funciones distintas en las que se usa el contexto para distinguir entre ellas. Por ejemplo, el método **imprimir** tendría una implementación distinta para clases textuales y gráficas.
- El **polimorfismo universal** hace referencia a métodos que funcionan con todos los tipos que posean una estructura común dada. Se pueden identificar dos tipos:
 1. El **parametrizado** hace referencia a funciones que tienen un parámetro de tipo implícito o explícito para cada tipo de aplicación de esa función. Por ejemplo, el método **tamaño**, que devolvería el número de elementos presente, tanto para una lista de enteros como para una lista de cadenas.
 2. Con la **inclusión** se consigue la funcionalidad de los métodos a través de la herencia y los subtipos. Por ejemplo, se hereda el método **llenarDeposito** de la clase **Coche** en la subclase **CocheDeportivo**, y el método funcionaría igual en los dos casos.

En el LPOO Java el polimorfismo por inclusión y de sobrecarga de métodos son estándar. La sobrecarga de operadores no está permitida.

Definición de sobrecarga de operadores

Sobrecarga que permite a un programador definir operadores (como + o *) para sus propias clases.

Ejemplo

Para una clase **Matriz**, se podría definir la operación de multiplicación de dos matrices como: **matriz1 * matriz2**.

La utilización correcta de la sobrecarga de operadores puede resultar en código que se entiende mejor. Si no, puede producir programas crípticos. En los LPOO que incluyen la sobrecarga de operadores hace falta dos cosas para evitar la ofuscación:

1. Todas las operaciones tienen que tener la forma de mensajes a objetos, para que todos los operadores sean llamadas a métodos.
2. Los operadores tienen que tener una forma funcional equivalente, para que la utilización de una llamada al método se comporte igual, independientemente de si se trata de un infijo, prefijo o posfijo (por ejemplo, $1 + 2$ debe ser igual a $1. + (2)$).

Ejemplo

Una clase puede tener varios métodos con el nombre **imprimir**, uno para cada tipo de elemento que se quiere imprimir: cadenas de texto, números enteros, etc.

Las ventajas que ofrece el polimorfismo son las siguientes:

Ventajas del polimorfismo

- Proporciona una separación entre interfaz e implementación para permitir una organización clara del código.
- Extensibilidad: comportamientos que funcionaban más arriba en la jerarquía de clases van a seguir funcionando en clases nuevas que hereden de ellas.
- Permite referencias a/de cualquier clase en la jerarquía de clases como si fuera de la clase base; así no hay que introducir código que compruebe explícitamente la clase en cuestión para producir comportamientos distintos.

Tanto Java como C++ permiten el polimorfismo.

Ejemplo

```
public class GestorImpresion {  
  
    public GestorImpresion(){  
        // init dispositivo;  
    }  
  
    public boolean imprimir(String s){  
        // detalles aquí  
    }  
  
    public boolean imprimir(integer i){  
        // detalles aquí  
    }  
  
    public boolean imprimir(MiObjeto mo){  
        // detalles aquí  
    }  
}
```

Tres ejemplos de polimorfismo



Info. y ejemplos

Capítulo 6 de Ghezzi y Jazayeri (1998), sección 3.2.8 de Cerrada y otros (2000) y capítulo 7 de Budd (1994)

5. Interacción basada en mensajes a objetos

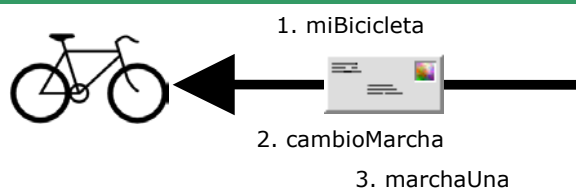
La funcionalidad compleja y útil de un programa OO no proviene de un sólo objeto, que no suele ser muy útil, sino de la interacción y comunicación entre varios objetos. Por ejemplo, una bicicleta colgada de una percha en el garaje no sirve de nada. Es solamente cuando se combina con otra entidad (un ser humano) e interacciona con él que ocurre un comportamiento interesante.

Los objetos de software interaccionan y se comunican entre ellos utilizando los **mensajes**. Cuando un objeto O_1 quiere realizar uno de los métodos del objeto O_2 , O_1 envía un mensaje a O_2 . Muchas veces el objeto recipiente necesita más información para poder realizar el método y esta información se envía con el mensaje como **parámetros**.

Definición de los tres componentes de un mensaje

- El **objeto** que es el recipiente del mensaje.
- El nombre del **método** que hay que realizar.
- Los **parámetros** que requiere el método.

Ejemplo



Los mensajes aportan dos beneficios:

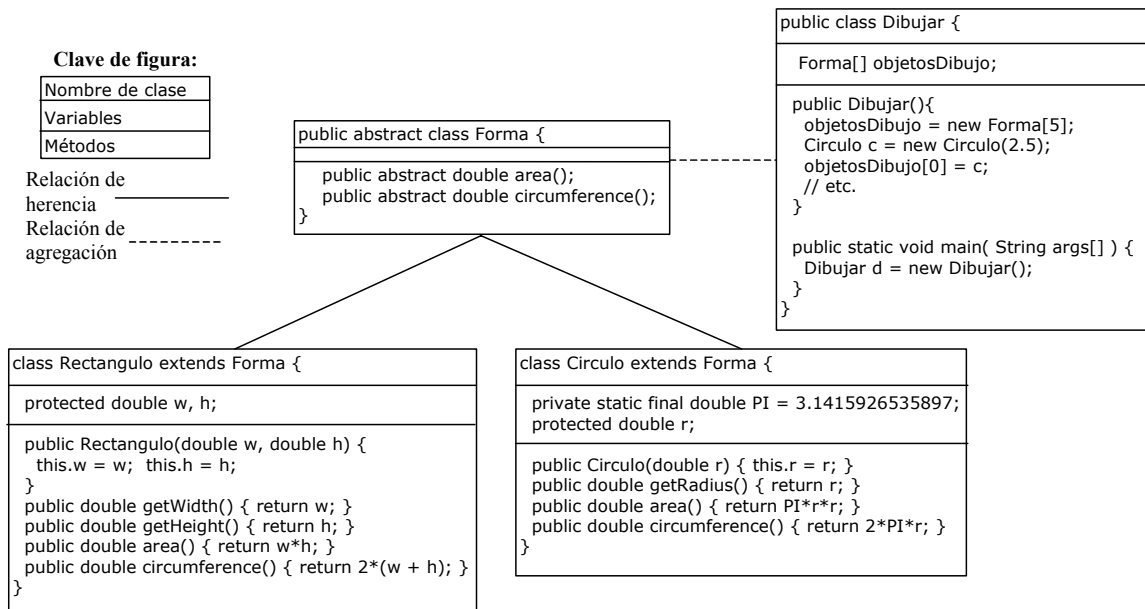
Ventajas de los mensajes

- Como el comportamiento de un objeto se expresa a través de sus métodos, el intercambio de mensajes permite todo tipo de interacciones posibles entre objetos.
- No es necesario que los objetos estén en el mismo proceso ni en la misma máquina para poder enviar y recibir mensajes.

Info. y ejemplos Capítulo 4 de Budd (1994)

Ejemplificación a través de un programa OO y ejercicios

Esta lección termina con un sencillo programa OO que muestra algunos de los principios presentados en este texto. Estúdielo y realice los ejercicios que aparecen a continuación:



1. Comente el programa ejemplo del apartado anterior identificando los aspectos de la POO destacados en este texto que se encuentran allí.
2. ¿En qué ocasiones se debería contemplar la utilización de instancias de clases en vez de heredarlas?
3. Desarrolle un sencillo programa OO (sin detallar el código de cada método) para el juego de cartas mus presentando el diseño como el ejemplo anterior.

Bibliografía recomendada para el alumno

Bernard, E.V. 2000. *Basic Object-Oriented Concepts* (www.toa.com/pub/oobasics/oobasics.htm).

Budd, T. 1994. *Introducción a la programación orientada a objetos*. Madrid: Addison-Wesley Iberoamericana.

Cerrada, J. A., Collado, M. Gómez, S.R. y Estívariz, J.F. 2000. *Introducción a la ingeniería del software*. Madrid: Ceura.

Eckel, B. 2001. *Pensando en Java*. Madrid: Pearson Educación.

- Ghezzi, C. y Jazayeri, M. 1998. *Programming Language Concepts*. Tercera edición. Nueva York: John Wiley & Sons.
- Pérez-Chirinos Sanz, C. 1995. "Objetos Contra Complejidad". *Novática* 114 (www.well.com/user/ritchie/cesar.html).
- Pratt, T.W. y Zelkowitz, M.V. 1998. *Lenguajes de Programación. Diseño e Implementación*. Tercera edición. México: Prentice Hall Hispanoamericana.
- Wegner, P. 1987. "Dimensions of Object-Based Language Design". En N.K. Meyrowitz (ed.) *Proceedings of OOPSLA'87*. ACM SIGPLAN Notices 22(12), 168-182 .