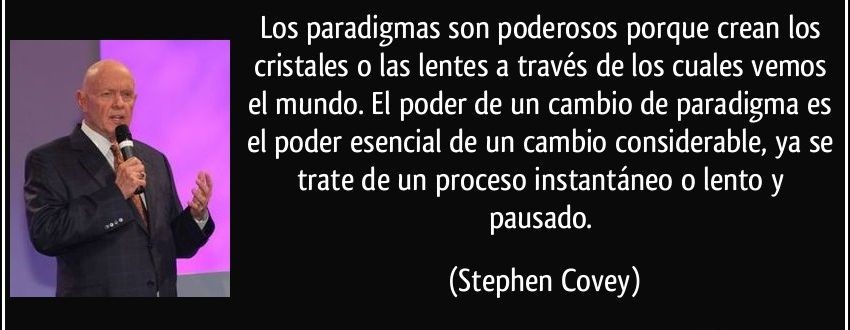


Programación Orientada Objetos

¿Qué son los paradigmas?



El concepto de paradigma se utiliza en la vida cotidiana como sinónimo de “marco teórico” o para hacer referencia a que algo se toma como “modelo a seguir”.

En términos generales, se puede definir al término paradigma como la forma de visualizar e interpretar los múltiples conceptos, esquemas o modelos del comportamiento en diversas disciplinas. A partir de la década del ’60, los alcances del concepto se ampliaron y ‘paradigma’ comenzó a ser un término común en el vocabulario científico y en expresiones epistemológicas cuando se hacía necesario hablar de modelos o patrones.

Una de las primeras figuras de la historia que abordaron el concepto que ahora nos ocupa fue el gran filósofo griego Platón que realizó su propia definición de lo que él consideraba que era un paradigma. En este sentido, el citado pensador expuso que esta palabra venía a determinar *qué son las ideas o los tipos de ejemplo de una cosa en cuestión*.

El estadounidense Thomas Kuhn, un experto en Filosofía y una figura destacada del mundo de las ciencias, fue quien se encargó de renovar la definición teórica de este término para otorgarle una acepción más acorde a los tiempos actuales, al adaptarlo para describir con él a *la serie de prácticas que trazan los lineamientos de una disciplina científica a lo largo de un cierto lapso temporal*.

De esta forma, un paradigma científico establece aquello que debe ser observado; la clase de interrogantes que deben desarrollarse para obtener respuestas en torno al propósito que se persigue; qué estructura deben poseer dichos interrogantes y marca pautas que indican el camino de interpretación para los resultados obtenidos de una investigación de carácter científico.

Cuando un paradigma ya no puede satisfacer los requerimientos de una ciencia (por ejemplo, ante nuevos hallazgos que invalidan conocimientos previos), es sucedido por otro. Se dice que un cambio de paradigma es algo dramático para la ciencia, ya que éstas se perciben como estables y maduras. En las ciencias sociales, el paradigma se encuentra relacionado al concepto de cosmovisión. El concepto se emplea para mencionar a todas aquellas experiencias, creencias, vivencias y valores que repercuten y condicionan el modo en que una persona ve la realidad y actúa en función de ello. Esto quiere decir que un **paradigma es también la forma en que se entiende el mundo**.

Paradigmas de Programación

Un paradigma de programación provee y determina la visión y métodos de un programador en la construcción de un programa o subprograma. Diferentes paradigmas resultan en diferentes estilos de programación y en diferentes formas de pensar la solución de problemas (con la solución de múltiples “problemas” se construye una aplicación o producto de software).

Los lenguajes de programación están basados en uno o más paradigmas, por ejemplo: Smalltalk y Java son lenguajes basados en el paradigma orientado a objetos. El lenguaje de programación Scheme, en cambio, soporta sólo programación funcional. Otros lenguajes, como C++ y Python soportan múltiples paradigmas.

**Clasificación de Paradigmas de Programación**

Un paradigma de programación representa un enfoque particular o filosofía para diseñar soluciones. Los paradigmas difieren unos de otros, en los conceptos y la forma de abstraer los elementos involucrados en un problema, así como en los pasos que integran su solución del problema, en otras palabras, el cómputo. Además, los paradigmas dependen del contexto y el tiempo en el que surgen, ya que nacen en base a una necesidad generalizada de la comunidad de desarrolladores de software, para resolver cierto tipo de problemas de la vida real. Un paradigma de programación está delimitado en el tiempo en cuanto a aceptación y uso, porque nuevos paradigmas aportan nuevas o mejores soluciones que la sustituyen parcial o totalmente. De esta forma podemos encontrar los siguientes tipos de paradigmas:

**Paradigma Imperativo**

Describe la programación como una secuencia de instrucciones o comandos que cambian el estado de un programa. El código máquina en general está basado en el paradigma imperativo. Su contrario es el paradigma declarativo. En este paradigma se incluye el paradigma procedural.

**Paradigma Declarativo**

No se basa en el cómo se hace algo (cómo se logra un objetivo paso a paso), sino que describe (declara) cómo es algo. En otras palabras, se enfoca en describir las propiedades de la solución buscada, dejando indeterminado el algoritmo (conjunto de instrucciones) usado para encontrar esa solución. Es más complicado de implementar que el paradigma imperativo, tiene desventajas en la eficiencia, pero ventajas en la solución de determinados problemas.

**Paradigma Estructurado**

La programación se divide en bloques (procedimientos y funciones) que pueden o no comunicarse entre sí. Además, la programación se controla con secuencia, selección e iteración. Permite reutilizar código programado y otorga una mejor compresión de la programación. Es contrario al paradigma no estructurado, de poco uso, que no tiene ninguna estructura, es simplemente un “bloque”, como, por ejemplo, los archivos en lote o batch (.bat).

Paradigma Funcional

Este paradigma concibe a la computación como la evaluación de funciones matemáticas y evita declarar y cambiar datos. En otras palabras, hace hincapié en la aplicación de las funciones y composición entre ellas, más que en los cambios de estados y la ejecución secuencial de comandos (como lo hace el paradigma procedimental). Permite resolver ciertos problemas de forma elegante y los lenguajes puramente funcionales evitan los efectos secundarios comunes en otro tipo de paradigmas.

Paradigma lógico

Este paradigma se basa en los conceptos de lógica matemática; trabaja con predicados que caracterizan o relacionan a los individuos involucrados y la deducción de las posibles respuestas a una determinada consulta. Es un tipo de paradigma de programación declarativo. La programación lógica gira en torno al concepto de predicado, o relación entre elementos. Define reglas lógicas para luego, a través de un motor de inferencias lógicas, responder preguntas planteadas al sistema y así resolver los problemas. Ej.: Prolog.

**Paradigma Orientado a Objetos**

La programación orientada a objetos intenta simular el mundo real a través del significado de objetos que contiene características y funciones. Está basado en la idea de encapsular estado y operaciones en objetos. En general, la programación se resuelve comunicando dichos objetos a través de mensajes. Su principal ventaja es la reutilización de código y su facilidad para pensar soluciones a determinados problemas. El lenguaje de programación elegido para trabajar es Java, que pertenece al paradigma orientado a objetos

**¿Qué es una Clase?**

Los conceptos de clase y objeto están estrechamente relacionados; sin embargo, existen diferencias entre ambos: mientras que un objeto es una entidad concreta que existe en el espacio y en el tiempo, una clase representa sólo una abstracción5 , la “esencia” de un objeto. Se puede definir una clase como un grupo, conjunto o tipo marcado por atributos comunes, una división, distinción o clasificación de grupos basada en la calidad, grado de competencia o condición. En el contexto del paradigma Orientado a Objetos se define una clase de la siguiente forma: “Una clase es un conjunto de objetos que comparten una estructura común y un comportamiento común” Un objeto es una instancia de una clase. Los objetos que comparten estructura y comportamiento similares pueden agruparse en una clase. En el paradigma orientado a objetos, los objetos pertenecen siempre a una clase, de la que toman su estructura y comportamiento. Por mucho tiempo se ha referenciado a las clases llamándolas objetos, no obstante, son conceptos diferentes.

Abstracción: es la acción de separar propiedades, en informática es muy utilizado como mecanismo de simplificación de la realidad. Se separan características esenciales de las que no son tanto. Se separa el “que” debe hacerse, del “cómo” debe hacerse.

¿Qué es un Objeto?

Desde el punto de vista de los humanos un objeto es cualquiera de estas cosas:

* Una cosa tangible y/o visible.
* Algo que puede comprenderse intelectualmente.
* Algo hacia lo que se dirige un pensamiento o una acción.

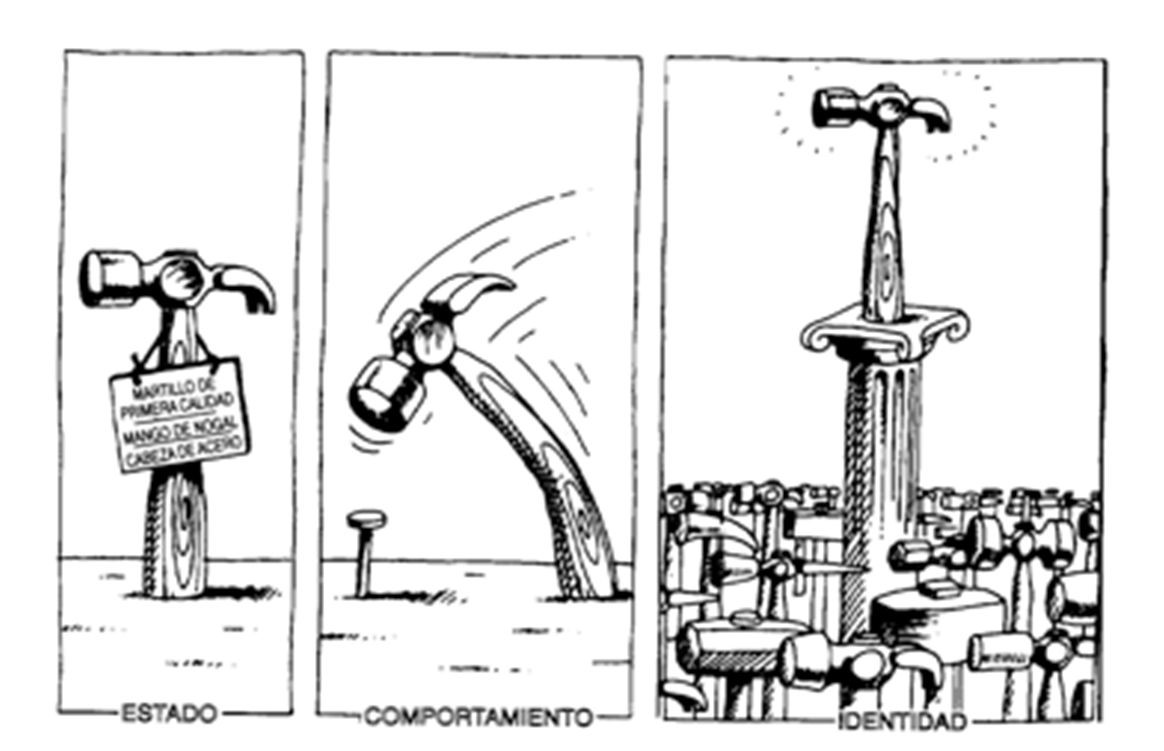
En software, el término objeto se aplicó formalmente por primera vez en el lenguaje Simula, por primera vez, donde los objetos existían para simular algún aspecto de la realidad. Los objetos del mundo real no son el único tipo de objeto de interés en el desarrollo de software. Otros tipos importantes de objetos son invenciones del proceso de diseño cuyas colaboraciones con objetos semejantes llevan a desempeñar algún comportamiento de nivel superior.

|  |
| --- |
| Podemos decir, entonces, que un objeto: “representa un elemento, unidad o entidad individual e identificable, ya sea real o abstracta, con un *rol bien definido* en el dominio del problema”. |

En términos más generales, se define a un objeto como: “cualquier cosa que tenga una frontera definida con nitidez”, pero esto no es suficiente para servir de guía al distinguir un objeto de otro ni permite juzgar la calidad de las abstracciones.

De esta forma en el dominio de un problema relacionado con vehículos podríamos identificar como ejemplos de objetos a cada una de las ruedas, las puertas, las luces y demás elementos que lo conforman. Los casos anteriores son representaciones de cosas tangibles y reales, aunque también

pueden serlo entidades abstractas como la marca, el modelo, el tipo de combustible que usa, entre otras. Además, como veremos luego, un objeto puede estar formado por otros objetos y en el ejemplo anterior el Automóvil en sí podría ser un objeto formado por los otros objetos que ya mencionamos.



|  |
| --- |
| ” Un objeto tiene un estado, comportamiento e identidad; la estructura y comportamiento de objetos similares están definidos en su clase común; los términos *instancia y objeto* son intercambiables”. |

Estado

|  |
| --- |
| El estado de un objeto abarca todas las propiedades (normalmente estáticas) del mismo más los valores actuales (normalmente dinámicos) de cada una de esas propiedades. |

Una propiedad es una característica inherente o distintiva, un rasgo o cualidad que contribuye a hacer que ese objeto sea ese objeto y no otro. Las propiedades suelen ser estáticas, porque atributos como este suelen ser inmutables y fundamentales para la naturaleza del objeto. Se dice “suelen” porque en algunas circunstancias las propiedades de un objeto pueden cambiar.

Todas las propiedades tienen un valor. Este valor puede ser una mera cantidad o puede denotar a otro objeto. Así distinguimos a:

* Objetos: existen en el tiempo, son modificables, tienen estado, son instanciados y pueden crearse, destruirse y compartirse.
* Valores simples: son atemporales, inmutables y no instanciados.

El hecho que todo objeto tiene un estado implica que todo objeto toma cierta cantidad de espacio, ya sea del mundo físico o de la memoria de la computadora.

Cada parte de la estructura (registro de personal) denota una propiedad particular de la abstracción de un registro de personal. Esta declaración denota una clase, no un objeto, porque no representa una instancia específica.

Puede decirse que todos los objetos de un sistema encapsulan (protegen) algún estado, y que todo estado de un sistema está encapsulado en un objeto. Sin embargo, encapsular el estado de un objeto es un punto de partida, pero no es suficiente para permitir que se capturen todos los designios de las abstracciones que se descubren e inventan durante el desarrollo. Por esta razón hay que considerar también cómo se comportan los objetos.

Comportamiento

Ningún objeto existe en forma aislada, en vez de eso, los objetos reciben acciones, y ellos mismos actúan sobre otros objetos. Así puede decirse que:

|  |
| --- |
| “El comportamiento es cómo actúa y reacciona un objeto, en términos de sus cambios de estado y paso de mensajes”. |

Dicho de otra forma: el comportamiento de un objeto representa su actividad visible y comprobable exteriormente.

Una operación en una acción que un objeto efectúa sobre otro con el fin de provocar una reacción. Las operaciones que los objetos cliente pueden realizar sobre un objeto, suelen declararse como métodos y forman parte de la declaración de la clase a la que pertenece ese objeto.

El paso de mensajes es una de las partes de la ecuación que define el comportamiento de un objeto; la definición de comportamiento también recoge que el estado de un objeto afecta asimismo a su comportamiento. Así, se puede decir, que el comportamiento de un objeto es función de su estado, así como de la operación que se realiza sobre él, teniendo algunas operaciones, el efecto colateral de modificar el estado del objeto. Refinando la definición de estado podemos decir:

|  |
| --- |
| *“El estado de un objeto representa los resultados acumulados de su comportamiento.”* |

**Operaciones**: una operación denota un servicio que un objeto que pertenece a una clase ofrece a sus clientes. Los tres tipos más comunes de operaciones son los siguientes:

|  |  |
| --- | --- |
| **Modificador:** | Una operación que altera el estado de un objeto |
| **Selector:** | Una operación que accede al estado del objeto, pero no lo altera. |
| **Iterador:** | Una operación que permite acceder a todos los objetos de una clase en algún orden, perfectamente establecido. |
| Hay otros dos tipos de operaciones habituales: representan la infraestructura necesaria para crear y destruir instancias de una clase, son: | |
| **Constructor:** | Una operación crea un objeto y/o inicializa su estado. |
| **Destructor:** | Una operación que libera el estado de un objeto y/o destruye el propio objeto. |

Unificando las definiciones de estado y comportamiento se definen las responsabilidades de un objeto de forma tal que incluyan dos elementos clave: el conocimiento que un objeto mantiene y las acciones que puede llevar a cabo. Las responsabilidades están encaminadas a transmitir un sentido del propósito de un objeto y de su lugar en el sistema.

La responsabilidad de un objeto son todos los servicios que proporciona para todos los contratos que soporta.

En otras palabras, se puede decir que el estado y el comportamiento de un objeto definen un conjunto de roles que puede representar un objeto en el mundo, los cuales, a su vez, cumplen las responsabilidades de la abstracción.

## **Identidad**

“La identidad es aquella propiedad de un objeto que lo distingue de todos los demás objetos”

La mayoría de los lenguajes de programación y de bases de datos utilizan nombres de variables para distinguir objetos, mezclando la posibilidad de acceder a ellos con su identidad. También utilizan claves de identificación para distinguir objetos, mezclando el valor de un dato con su identidad.

Un objeto dado puede nombrarse de más de una manera; a esto se lo denomina compartición estructural. En otras palabras, existen alias para el objeto que nos permiten llamarlo de diferentes formas en base a las necesidades de acceso que tengamos.

El tiempo de vida de un objeto se extiende desde el momento en que se crea por primera vez (y consume espacio por primera vez) hasta que se destruye. Para crear un objeto hay que declararlo o bien asignarle memoria dinámicamente; generalmente esto se realiza con la ayuda de los métodos **constructores** y **destructores** mencionados anteriormente.

La declaración e inicialización de objetos consume memoria RAM[[1]](#footnote-1), por lo que su administración es de suma importancia para garantizar el correcto funcionamiento del programa. Dependiendo del lenguaje de programación que utilicemos es posible que, para liberar memoria, una vez que un objeto deja de ser necesario, debamos destruirlo de forma explícita. En cambio, otros lenguajes de programación están provistos de herramientas que se encargan automáticamente de esta tarea, liberándonos de la responsabilidad de destruir los objetos.

## **Relaciones entre objetos**

Los objetos contribuyen al comportamiento de un sistema colaborando con otros. La relación entre dos objetos cualesquiera abarca las suposiciones que cada uno realiza acerca del otro, incluyendo qué operaciones pueden realizarse y qué comportamiento se obtiene. Hay dos tipos de jerarquías de especial interés:

* Enlaces
* Agregación

**Enlaces**

Definido como “conexión física o conceptual entre objetos”. Un objeto colabora con otros a través de sus enlaces con éstos. Dicho de otra forma: un enlace denota la asociación específica por la cual un objeto (cliente) utiliza los servicios de otro objeto (servidor), o a través de la cual un objeto puede comunicarse con otro.

Como participante de un enlace, un objeto puede desempeñar uno de tres roles:

* **Actor**: Un objeto puede operar sobre otros objetos, pidiéndole colaboración por medio de sus mensajes, pero nunca otros objetos operan sobre él, a veces, los términos *objeto activo y actor* son equivalentes.
* **Servidor**: Un objeto que nunca opera sobre otros, sólo otros objetos, operan sobre él.
* **Agente**: Un objeto puede operar sobre otros objetos, y además otros objetos operan sobre él. Un agente se crea normalmente para realizar algún trabajo en nombre de un actor u otro agente.

*Visibilidad*

|  |
| --- |
| La visibilidad es una propiedad que permite a un objeto operar sobre otro. Si un objeto no ve a otro, no puede enviarle un mensaje, para pedirle su colaboración. |

Puede postergarse la definición de visibilidad de un objeto durante los momentos iniciales del análisis de un problema; no obstante, ni bien comienzan las actividades de diseño e implementación, hay que considerar la visibilidad a través de los enlaces porque las decisiones en este punto dictan el ámbito y acceso de los objetos a cada lado del enlace. Agregación Mientras que los enlaces denotan relaciones de igual a igual o de cliente/servidor, la agregación denota una jerarquía todo/parte, con la capacidad de ir desde el todo (agregado) hasta las partes. En este sentido la agregación es un tipo especial de relación. La agregación puede o no denotar contención física. La relación todo/parte es más conceptual y por ende menos directa que la agregación física. Existen claros pros y contras entre la agregación y los enlaces. La agregación es a veces mejor porque encapsula partes y secretos del todo. A veces, son mejores los enlaces porque permiten acoplamientos más débiles entre los objetos. Es necesario sopesar cuidadosamente ambos factores a la hora de elegir un tipo de relación entre los objetos.

**Agregación**

Mientras que los enlaces denotan relaciones de igual a igual o de cliente/servidor, la agregación denota una jerarquía todo/parte, con la capacidad de ir desde el todo (agregado) hasta las partes. En este sentido la agregación es un tipo especial de relación. La agregación puede o no denotar contención física. La relación todo/parte es más conceptual y por ende menos directa que la agregación física. Existen claros pros y contras entre la agregación y los enlaces. La agregación es a veces mejor porque encapsula partes y secretos del todo. A veces, son mejores los enlaces porque permiten acoplamientos más débiles entre los objetos. Es necesario sopesar cuidadosamente ambos factores a la hora de elegir un tipo de relación entre los objetos.

**Relaciones entre clases**

Las clases, al igual que los objetos, no existen aisladamente. Para un dominio de problema específico, las abstracciones suelen estar relacionadas por vías muy diversas y formando una estructura de clases del diseño. Se establecen relaciones entre dos clases por una de estas dos razones: ● una relación entre clases podría indicar algún tipo de compartición. Programa Nacional 111 Mil – Analista

● una relación entre clases podría indicar algún tipo de conexión semántica.

En total existen tres tipos básicos de relaciones entre clases:

● Generalización/Especificación: denota un tipo de relación “es un” o mejor aún, “se comporta como”.

● Todo/Parte: denota una relación “parte de”.

● Asociación: que denota alguna dependencia semántica entre clases de otro modo independientes, muchas veces referenciada como hermano-hermano. La mayoría de los lenguajes de programación orientados a objetos ofrecen soporte directo para alguna combinación de las siguientes relaciones:

• Asociación

• Herencia

• Agregación

De estos seis tipos de relaciones, las asociaciones son en general las más utilizadas, pero son las más débiles semánticamente. La identificación de asociaciones se realiza en al análisis y diseño inicial, momento en el que se comienzan a descubrir las dependencias generales entre las abstracciones. Al avanzar en el diseño y la implementación se refinarán a menudo estas asociaciones orientándolas hacia una de las otras relaciones de clases más específicas y refinadas.

Asociación Dependencias semánticas:

una asociación denota una dependencia semántica y puede establecer o no, la dirección de esta dependencia, conocida como navegabilidad. Si se establece la dirección de la dependencia o navegabilidad, debe nombrarse el rol que desempeña la clase en relación con la otra. Esto es suficiente durante el análisis de un problema, donde sólo es necesario identificar esas dependencias. Multiplicidad: existen tres tipos habituales de multiplicidad en una asociación:

● Uno a uno

● Uno a muchos

● Muchos a muchos

**Herencia**

Las personas tendemos a asimilar nuevos conceptos basándonos en lo que ya conocemos. “La herencia es una herramienta que permite definir nuevas clases en base a otras clases”. La herencia es una relación entre clases, en la que una clase comparte la estructura y/o el comportamiento definidos en una (herencia simple) o más clases (herencia múltiple). La clase de la que otras heredan se denomina “superclase”, o “clase padre”. La clase que hereda de una o más clases se denomina “subclase” o “clase hija”. La herencia define una jerarquía de “tipos” entre clases, en la que una subclase hereda de una o más superclases.

Una subclase habitualmente aumenta o restringe la estructura y el comportamiento existentes en sus superclases. Una subclase que aumenta una superclase se dice que utiliza herencia por extensión. Una subclase que restringe el comportamiento de sus superclases se dice que utiliza herencia por restricción. En la práctica no es muy claro cuando se usa cada caso; en general, es habitual que hagan las dos cosas.

Algunas de las clases tendrán instancias, es decir se crearán objetos a partir de ellas, y otras no. Las clases que tienen instancias se llaman concretas y las clases sin instancias son abstractas.

Una clase abstracta se crea con la idea de que sus subclases añadan elementos a su estructura y comportamiento, usualmente completando la implementación de sus métodos (habitualmente) incompletos, con la intención de favorecer la reutilización . En una estructura de clases, las clases más generalizadas de abstracciones del dominio se denominan clases base. Una clase cualquiera tiene típicamente dos tipos de “clientes”:

● Instancias

● Subclases

Resulta útil definir interfaces distintas para estos dos tipos de clientes. En particular, se desea exponer a los clientes instancia sólo los comportamientos visibles exteriormente, pero se necesita exponer las funciones de asistencia y las representaciones únicamente a los clientes subclase. El uso de la herencia expone algunos de los secretos de la clase heredada. En la práctica, esto implica que, para comprender el significado de una clase particular, muchas veces hay que estudiar todas sus superclases, a veces incluyendo sus vistas internas.

**Polimorfismo**

El polimorfismo es un concepto aplicado en el contexto de la teoría de tipos, como en las relaciones de herencia o generalización, donde el nombre de un método puede implicar comportamientos diferentes, conforme esté especificado en clases diferentes, en tanto y en cuanto estas clases estén relacionadas por una superclase común.

Un objeto puede responder de diversas formas a un mismo comportamiento, denotado por este nombre, dependiendo de la clase a la que pertenezca. El polimorfismo es la propiedad que potencia el uso de la herencia y para poder utilizarla, las clases deben especificar los comportamientos con los mismos protocolos.

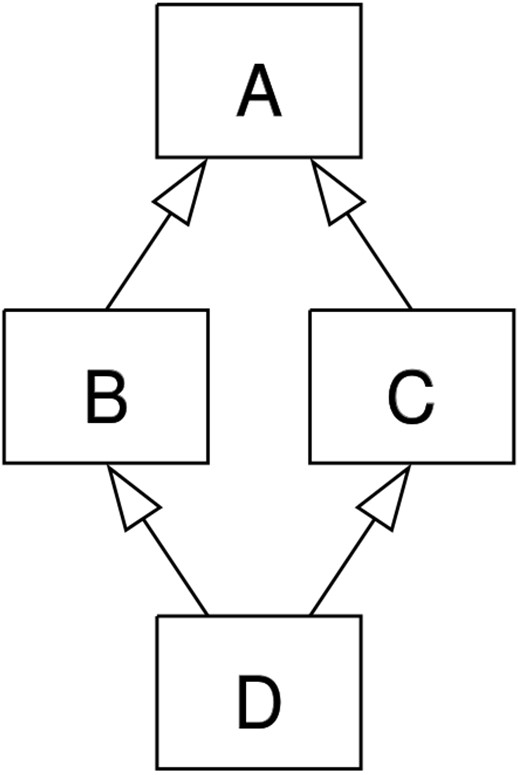
La herencia sin polimorfismo es posible, pero no es muy útil. La herencia puede utilizarse como:

● Una decisión privada del diseñador para “reusar” código porque es útil hacerlo; debería ser posible cambiar esta decisión con facilidad.

● La realización de una declaración pública de que los objetos de la clase hija obedecen a la semántica de la clase padre, de modo que la clase hija implementa, especializa o refina la clase padre. Con herencia simple, cada subclase tiene exactamente una superclase; aunque la herencia simple es muy útil, frecuentemente fuerza al programador a derivar de una entre dos clases igualmente atractivas.

Esto limita la aplicabilidad de las clases predefinidas, haciendo muchas veces necesario duplicar el código. La herencia múltiple es bastante discutida aún, y permite que una clase hija puede heredar de más de una clase padre. Podría decirse que es como un paracaídas: no siempre se necesita, pero cuando así ocurre, uno está verdaderamente feliz de tenerlo a mano. Distintos lenguajes de programación posibilitan o no este tipo de herencia.

En el contexto de la herencia múltiple, se conoce como problema del diamante a una ambigüedad que surge cuando dos clases B y C heredan de A, y la clase D hereda de B y C. Si un método en D llama a un método definido en A, ¿por qué clase lo hereda, B o C? Se llama el problema del 'diamante' por la forma del diagrama de herencia de clase en esta situación. La clase A está arriba, B y C están separadas debajo de ella, y D se une a las dos en la parte inferior consiguiendo la forma de un diamante. El ornitorrinco, que deriva de mamífero y ovíparo. Las dos clases derivan de animal.



Agregación

Las relaciones de agregación entre clases tienen un paralelismo directo con las relaciones de agregación entre los objetos correspondientes a esas clases. La agregación establece una dirección en la relación todo/parte. Pueden identificarse dos tipos de contención.

* Contención física también conocida como composición: es una contención por valor, lo que significa que el objeto no existe independientemente de la instancia que lo encierra. El tiempo de vida de ambos objetos está en íntima conexión.
* Contención por referencia: es un tipo menos directo de agregación, en este caso se pueden crear y destruir instancias de cada objeto en forma independiente.

La contención por valor no puede ser cíclica, es decir que ambos objetos no pueden ser físicamente partes de otro. En el caso de la contención por referencia puede serlo (cada objeto puede tener un puntero apuntando al otro).

No hay que olvidar que la agregación no precisa contención física. Podemos afirmar que: sí y solo sí existe una relación todo/parte entre dos objetos, podremos tener una relación de agregación entre sus partes correspondientes.

**Interfaces**

Una interfaz es un conjunto de métodos abstractos y de constantes cuya funcionalidad es la de determinar el funcionamiento de una clase, es decir, funciona como un molde o como una plantilla. Al ser sus métodos abstractos estos no tiene funcionalidad alguna, sólo se definen su tipo, argumento y tipo de retorno.

Veamos algunas características de las Interfaces:

* Las Interfaces solo pueden tener visibilidad de **package**o **public**.
* Todos los métodos declarados en una Interfaz son **public**y **abstract**, si no se le indica, Java lo pondrá implícitamente.
* Los métodos de una Interfaz no pueden ser estáticos, ya que estos deben ser redefinidos, y al ser estáticos les sería imposible.
* Todos los atributos declarados en una Interfaz son **"public static final"** y deberán tener asignado un valor constante. Todos los nombres de constantes van en **"MAYÚSCULAS"**
* Una clase puede implementar una o más interfaces.
* Una Interfaz puede heredar de una o varias interfaces, pero no pueden implementar **NADA**. Como sabemos, las clases solo pueden heredar de otra clase, pero las Interfaces pueden hacer herencia múltiples.
* Todas las clases que implementen una interfaz deben de redefinir todos los métodos de esa interfaz.
* Las interfaces no tienen constructor, por lo que no es posible crear objetos con el operador **"new"** de ésta.
* Si se implementa en una clase una Interfaz, y esta Interfaz hereda de otra, la clase deberá implementar todos los métodos de la interfaz que implementa y de los métodos que esta hereda.
* Para la herencia de las interfaces se pone la palabra clave **extends**y se ponen todas las interfaces que se deseen separadas por comas.
* Una Interfaz puede ocultar constantes y métodos heredados de otras Interfaces si los vuelve a declarar en la clase que hereda.

¿Por qué son útiles la interfaces?

En cambio sí podemos decir que reúne las otras dos ventajas de la herencia: favorecer el mantenimiento y la extensión de las aplicaciones. ¿Por qué? Porque  **al definir interfaces permitimos la existencia de variables polimórficas y la invocación polimórfica de métodos.** En el diagrama que vimos anteriormente tanto árboles como arbustos, vehículos y personas son de tipo Actor, de modo que podemos generar código que haga un tratamiento en común de todo lo que son actores. Por ejemplo, podemos necesitar una lista de Actores. Podemos declarar una variable como de tipo Actor (aunque no puedan existir instancias de Actor) que permita referenciar alternativamente a objetos de las distintas subclases de la interfaz.

Un aspecto fundamental de las interfaces en Java es hacer lo que ya hemos dicho que hace una interfaz de forma genérica: **separar la especificación de una clase (qué hace) de la implementación (cómo lo hace).** Esto se ha comprobado que da lugar a programas más robustos y con menos errores.

**CLASES Y MÉTODOS ABSTRACTOS EN JAVA.**

Supongamos un esquema de herencia que consta de la clase Profesor de la que heredan ProfesorInterino y ProfesorTitular. Es posible que todo profesor haya de ser o bien ProfesorInterino o bien ProfesorTitular, es decir, que no vayan a existir instancias de la clase Profesor. Entonces, ¿qué sentido tendría tener una clase Profesor?

El sentido está en que una superclase permite unificar campos y métodos de las subclases, evitando la repetición de código y unificando procesos. Ahora bien, una clase de la que no se tiene intención de crear objetos, sino que únicamente sirve para unificar datos u operaciones de subclases, puede declararse de forma especial en Java: como clase abstracta. La declaración de que una clase es abstracta se hace con la sintaxis **public abstract class NombreDeLaClase { … }**. Por ejemplo public abstract class Profesor. Cuando utilizamos esta sintaxis, no resulta posible instanciar la clase, es decir, no resulta posible crear objetos de ese tipo. Sin embargo, sigue funcionando como superclase de forma similar a como lo haría una superclase “normal”. La diferencia principal radica en que no se pueden crear objetos de esta clase.

Declarar una clase abstracta es distinto a tener una clase de la que no se crean objetos. En una clase abstracta, no existe la posibilidad. En una clase normal, existe la posibilidad de crearlos aunque no lo hagamos. El hecho de que no creemos instancias de una clase no es suficiente para que Java considere que una clase es abstracta. Para lograr esto hemos de declarar explícitamente la clase como abstracta mediante la sintaxis que hemos indicado. Si una clase no se declara usando abstract se cataloga como “clase concreta”. En inglés abstract significa “resumen”, por eso en algunos textos en castellano a las clases abstractas se les llama resúmenes. Una clase abstracta para Java es una clase de la que nunca se van a crear instancias: simplemente va a servir como superclase a otras clases. No se puede usar la palabra clave new aplicada a clases abstractas. En el menú contextual de la clase en BlueJ simplemente no aparece, y si intentamos crear objetos en el código nos saltará un error.

A su vez, las clases abstractas suelen contener métodos abstractos: la situación es la misma. Para que un método se considere abstracto ha de incluir en su signatura la palabra clave abstract. Además un método abstracto tiene estas peculiaridades:

a) **No tiene cuerpo** (llaves): sólo consta de signatura con paréntesis.

b) Su signatura **termina con un punto y coma.**

c) **Sólo puede existir dentro de una clase abstracta.** De esta forma se evita que haya métodos que no se puedan ejecutar dentro de clases concretas. Visto de otra manera, si una clase incluye un método abstracto, forzosamente la clase será una clase abstracta.

d) Los métodos abstractos **forzosamente habrán de estar sobreescritos en las subclases.** Si una subclase no implementa un método abstracto de la superclase tiene un método no ejecutable, lo que la fuerza a ser una subclase abstracta. Para que la subclase sea concreta habrá de implementar métodos sobreescritos para todos los métodos abstractos de sus superclases.

1. [↑](#footnote-ref-1)