PRINCIPIO *MIXIN*

1. Problema

Implementar herencia múltiple en un lenguaje sin herencia múltiple de clases pero con herencia múltiple de interfaces.

2. Solución

Dado que el problema de la herencia múltiple es la combinación de comportamientos definidos dentro de las clases, se evita que las clases hereden directamente de otras, heredando sólo de interfaces. La herencia sólo de interfaces permite la reutilización del tipo. La reutilización del comportamiento se realiza por composición.

3. Ejemplo

En ciertos casos, la herencia múltiple es útil o requerida para indicar que un cierto elemento posee dos o más tipos, ya que posee dos o más naturalezas distintas. Por ejemplo, si tenemos las clases *PartículaCargada* y *PartículaPesada*, un *Electrón* tendría ambos tipos, pues un electrón tiene masa, y por tanto es una partícula pesada, y tiene carga, por lo que es una partícula cargada. Por tanto, de acuerdo al diseño que se muestra en la Figura 1, un electrón debería heredar de las clases *PartículaPesada* y *PartículaCargada*.

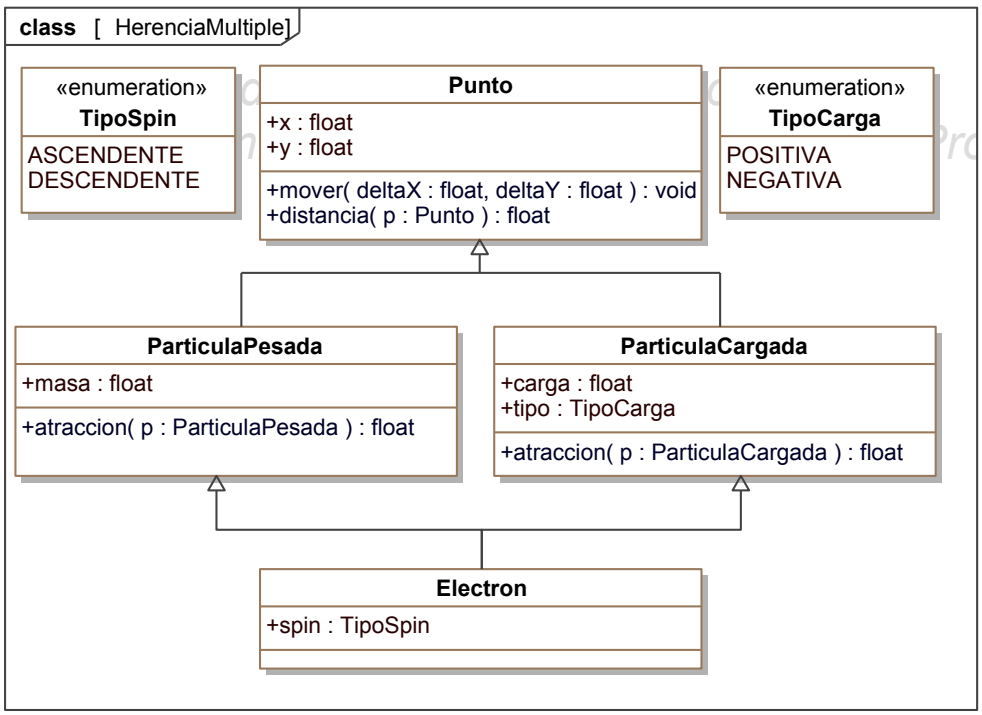

Figura 1. Herencia múltiple entre clases

Como sabemos, la herencia múltiple presenta dos problemas.

El primero de ellos depende de cómo el lenguaje implemente la herencia múltiple y se denomina *herencia en rombo*. Aparece cuando una subclase hereda una misma característica a través de dos caminos distintos del "*árbol*" de herencia. Este sería el caso de las propiedades *x* e *y* de la clase *Punto*, al igual que los métodos *mover* y *distancia*. La clase *Electron* heredaría estas propiedades tanto a través del camino *Electron->ParticulaPesada->Punto*, como del camino *Electron->ParticulaCargada->Punto*. Por tanto, en función de la estrategia elegida por el lenguaje para implementar la herencia, una llamada al método *mover* sobre un objeto de la clase *Electron* podría resultar en una ejecución doble del método *mover* de la clase *Punto*.

Este problema se puede corregir de diversas formas. Por ejemplo, utilizado un *algoritmo de linearización* (solución utilizada en Perl o Scala), que resuelve automáticamente el camino a seguir, o proporcionando mecanismos a nivel de lenguaje de programación que permitan al programador indicar qué camino seguir (solución utilizada en Eiffel).

El segundo problema que aparece con la herencia múltiple es el de la aparición de conflictos por colisión de elementos heredados. En este caso se heredan métodos con perfil compatible de dos superclases diferentes. Por ejemplo, la clase *Electron* hereda el método *atraccion* de las clases *ParticulaPesada* y *ParticulaCargada*. Por tanto, si tenemos dos objeto *e1* y *e2* de la clase *Electron*, si tratásemos de ejecutar la instrucción *e1.atraccion(e2)*, no sabríamos que método invocar, pues la llamada es compatible con los métodos de las dos superclases. Es decir, hay un conflicto por colisión de elementos heredados similares a través de dos caminos diferentes. Este problema se soluciona normalmente proporcionando mecanismos a nivel de lenguaje de programación que permitan al programador indicar qué camino seguir (solución utilizada en Eiffel o Perl), o prohibiendo que se den este tipo de situaciones, que se reportarían como errores del compilador.

Estos problemas se resuelven en la mayoría de los lenguajes de programación orientados a objetos actuales, como Java o C#, de las siguientes formas:

(1) En primer lugar, se evita que se puedan crear grafos de herencia entre clases, pudiendo crearse sólo *árboles de herencia.* En un árbol, por definición, sólo hay un camino desde un nodo del árbol a la raíz, por lo que no es posible que una misma superclase sea alcanzable desde dos caminos distintos desde una subclase. Para poder construir un árbol, cada nodo, salvo la raíz, debe tener un único padre, por lo que una clase sólo puede heredar de una superclase.
(2) Como consecuencia de lo anterior, al haber ahora una sola superclase, se evita la aparición de conflictos por colisión de elementos similares heredados por caminos distintos.
(3) Se permite la herencia de tantas interfaces como se desee, lo que permite que una clase pueda tener tantos tipos como se desee, indicando en cada interfaz, a qué métodos se espera que cada tipo sea capaz de responder, pero sin proporcionar una implementación para esos métodos, la cual deberá ser proporcionada por la clase hija. Al no existir

implementación para estos métodos, no hay problema de colisión, ya que aunque existan cabeceras repetidas, sólo habrá una implementación del método.

Por tanto, vemos que los lenguajes de programación orientados a objetos modernos, como Java o C#, no soportan herencia múltiple. Sin embargo, en la vida real, un objeto presenta muchos tipos, como es el caso del *Electron* en la Figura 1.

El patrón *mixin* trata de proporcionar una solución a este problema de forma que se pueda reutilizar tanto tipo como comportamiento sin necesidad de disponer de herencia múltiple en un lenguaje. La solución planteada es reutilizar el tipo mediante la herencia de interfaces y el comportamiento por *composición más delegación*. El esquema de esta solución se muestra en la Figura 2.

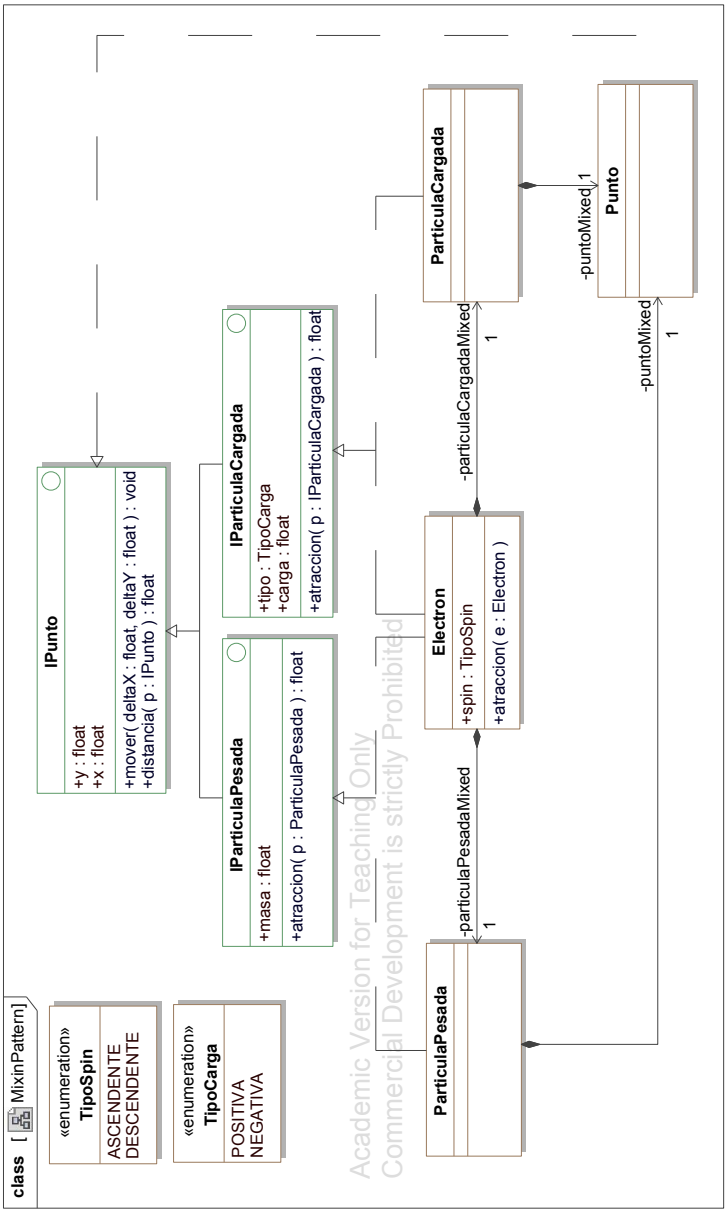

Figura 2. Patrón *mixin* aplicado sobre cada clase de la Figura 1

El esquema general de funcionamiento es como sigue:

(1) Para cada clase *ClaseEjemplo*, se crea una interfaz *IClaseEjemplo* con los métodos públicos de dicha clase (refactorización *ExtractInterface*). Obviamente, se obliga a que cada clase implemente o herede de su propia interfaz, es decir, *ClaseEjemplo* heredaría de *IClaseEjemplo*.

(2) Cuando una clase *Hija* hereda de otra clase *Padre* se realizan dos acciones diferenciadas, en lugar de hacer que la clase *Hija* herede directamente de la *Padre*:

a. En primer lugar, se hace que la clase de la interfaz asociada a la clase hija, es decir *IHija*, herede de la interfaz asociada a la clase padre, es decir, *IPadre*. Los objetos de la clase *Hija*, por tanto, tendrían como tipo *Hija*, *IHija* e *IPadre*. Con esto conseguimos poder reutilizar el tipo de la clase padre, pero no la implementación de sus métodos.

b. La reutilización de los métodos, se realiza por *composición más delegación*. Para ello, dentro de la clase *Hija* creamos un atributo privado con tipo la clase *Padre.* Este atributo se denomina *mixin*, por lo que se le suele nombrar como *mixin<NombreClaseRefenciada>*, que en nuestro caso sería *mixinPadre*.

c. A continuación, se hace que la implementación de cada método de la interfaz *IPadre* en la clase *Hija* delegue en el método con el mismo nombre del objeto *mixinPadre*. Por ejemplo, si la interfaz *IPadre* contuviese un método *void foo()*, la implementación que se realizaría sería la que se muestra en la Figura 3. Con esto conseguimos reutilizar el comportamiento de la case *Padre* por composición. Cabe destacar que muchos entornos de desarrollo modernos como Eclipse permiten la generación automática de estos delegados.

```
public void foo() {
    mixinPadre.foo();
}
```

Figura 3. Reutilización por composición más delegación

4. Variaciones

En una aplicación del patrón *mixin* requeriría declarar una interfaz por cada clase y nunca una clase heredaría de otra, sino que sólo se heredaría de interfaces. Esto generar un alto número de interfaces y delegados, la mayor parte innecesarios, sólo con el objeto de permitir que en un futuro dichas clases puedan participar en herencias múltiples, tanto como clases hijas o como clases padres. En la práctica real diaria, para reducir esta complejidad extra, que sería complejidad accidental, el patrón *mixin* se aplica solamente a aquellas clases de un diseño que se encuentran afectadas por la herencia múltiple.

Pablo Sánchez Barreiro