INTRODUCCIÓN

La herencia es una propiedad específica del paradigma orientado a objetos. Este pilar constituye también una relación entre clases. Una relación permite establecer como interactúan las clases. En el caso de la herencia esta relación establece que una clase declara a otra clase como su principal, también se dice que una clase (denominada subclase) declara a otra como su superclase. Esta declaración se denomina herencia porque la subclase adquiere en forma automática los atributos y operaciones de la superclase. De esta forma la subclase además de los atributos y operaciones que posee adquiere los atributos y operaciones de la superclase. El objetivo de este apunte será describir las implicancias de utilizar esta propiedad y su forma de modelado en UML.

REPRESENTACIÓN EN EL DIAGRAMA DE CLASES

En el diagrama de clases de UML la herencia posee un elemento de relación propio basado en una flecha que une la superclase con sus subclases. Sea el siguiente ejemplo

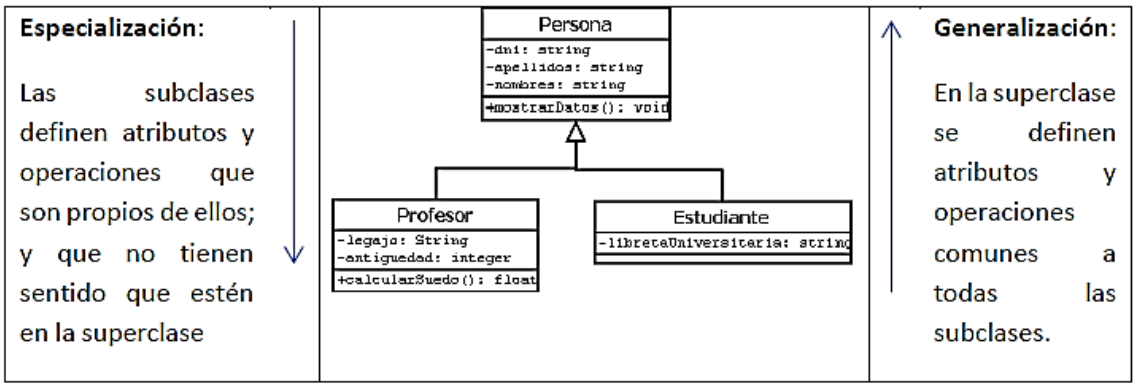

Figura 1. Ejemplo de herencia y tipos de vista de una herencia

Observe que se ha definido la clase Persona, la cual posee 3 atributos y una operación. Luego se han definido las clases Profesor y Estudiante. En principio la clase Profesor posee 2 atributos y 1 operación, mientras que la clase Estudiante posee 1 atributo y ninguna operación.

Sin embargo, la flecha que sale de cada una de estas clases hacia la clase Persona, indica que se está estableciendo una relación de herencia: La clase Persona se convierte en una superclase, mientras que Profesor y Estudiante se convierten en subclases. Como consecuencia, ahora Profesor posee 5 atributos y 3 operaciones, mientras que Estudiante posee 4 atributos y una operación.

La forma de la punta de la flecha de la relación de herencia es muy clara: un triángulo sin relleno que siempre apunta a la superclase.

VISTAS DE UNA RELACIÓN DE HERENCIA

Observe nuevamente la Figura 1. En ella se describe que existen dos formas de interpretar la relación de herencia dependiendo del punto de vista desde el cual se analice la relación. Si se parte desde la superclase hacia las subclases, la relación de herencia describe un proceso de especialización: esto significa que cada una de las subclases pueden definir atributos y operaciones que no son comunes entre ellas. Esto permite que cada una de estas subclases

pueda brindar con mayor detalle sus propias características (atributos) y responsabilidades (operaciones).

Si, por el contrario, analizamos la relación de herencia desde las subclases hacia la superclase, estaremos en presencia de un proceso de generalización. Esto es, en la superclase se definirán los atributos y operaciones que son comunes o "genéricos" a todas las subclases. El proceso de generalización-especialización de la relación de herencia debería poder brindarle un punto de referencia para determinar si existe esta relación entre dos clases.

VALIDACIÓN DEL MODELADO DE UNA RELACIÓN DE HERENCIA

En la figura 2, se intenta establecer una relación de herencia entre la clase Casa y la clase Mueble, donde la primera se convierte en la superclase y la segunda en la subclase. Se ha relacionado dos clases indicando que Casa es la superclase de Mueble. Si realizamos un análisis del proceso de generalización – especialización, propuesto para estas clases, podemos detectar que aparentemente la aplicación de la relación no tiene sentido: un mueble no hereda las cantidades de habitaciones de una casa, esto es: un mueble no posee como atributo genérico la cantidad de habitaciones.

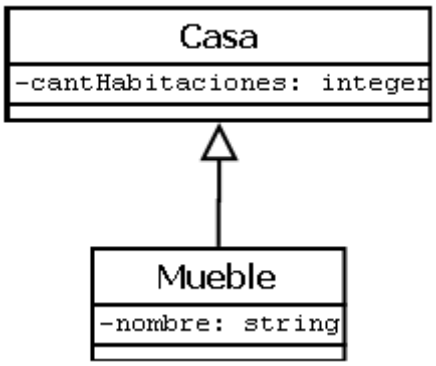

Figura 2. Una relación de herencia inválida

Esto significa que al momento de decidir establecer una relación de herencia entre dos clases se debe verificar que la misma sea correcta.

Para ello existe una frase que permite verificar si la relación de herencia que se desea establecer es correcta. Esta frase se denomina **Frase Semántica** y en el caso de la herencia es: ¿**<<La subclase>> es un tipo de/es un/es una <<La superclase>>**?

Si la respuesta es afirmativa entonces es correcto establecer la relación de herencia, caso contrario no se debe establecer la relación de herencia.

En el ejemplo de la figura 2 la frase semántica se aplicaría de la siguiente manera: ¿El mueble es un tipo de Casa? o ¿El mueble es una casa? Evidentemente la respuesta es NO. Las casas poseen muebles, pero los muebles no son casas.

De la misma manera para el ejemplo de la figura 1 se puede verificar:

- ¿Un profesor es un tipo de Persona? La respuesta es SI, por lo tanto, fue correcto establecer la herencia entre ambos
- ¿Un estudiante es un tipo de Persona? En este caso también es correcta la afirmación.

CONSECUENCIAS DE LA APLICACIÓN DE LA HERENCIA

La herencia conlleva un conjunto de efectos secundarios que se deben tener en cuenta cuando se decide aplicarla en un modelado de clases.

Redefinir la visibilidad de los atributos de la superclase

Observe nuevamente la figura 1. Como se había mencionado anteriormente la clase Profesor hereda 3 atributos de la clase Persona. Sería razonable pensar que la subclase debería poder manipular directamente esos atributos heredados, como si se hubieran definido dentro de la misma clase Profesor. Pero esto no sucede. Si bien se creará un único objeto con todos los atributos y operaciones definidos en Profesor y Persona, las clases diseñadas son diferentes, con una relación definida pero diferentes al final de cuentas.

Recuerde que la visibilidad privada de los atributos y operaciones (-) indica que ninguna clase externa puede acceder a esos atributos y operaciones. Por este motivo tanto en UML como en los lenguajes de programación se ha definido el modificador de acceso protected (protegido o #) que mantiene la imposibilidad de manipulación de atributos y operaciones de una clase a toda clase ajena, salvo que sea una subclase. Por lo tanto, en realidad la Figura 1 debería tener la siguiente forma

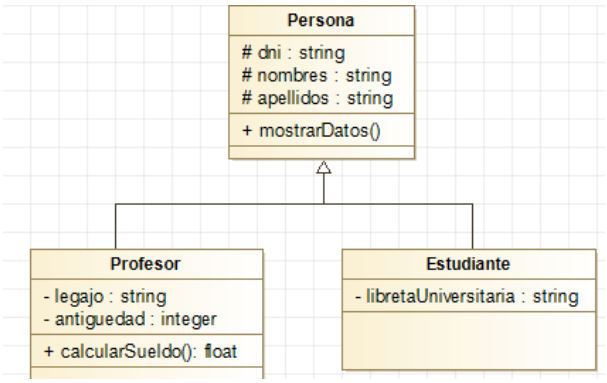

Figura 3. Herencia con atributos de la superclase con visibilidad protegida

La sobreescritura

Es aquella situación donde una misma operación se define tanto en la superclase como en la subclase. Por ejemplo:

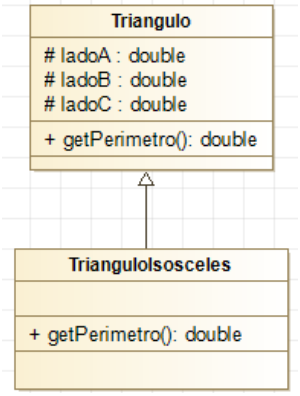

Figura 4. Ejemplo de sobreescritura

Observe que **TrianguloIsosceles** posee 2 operaciones con el mismo nombre y la misma lista de parámetros, por lo cual no es una sobrecarga de operaciones.

Formalmente se dice que ambas operaciones tienen ***la misma semántica*** (nombre de la operación) y la ***misma firma*** (lista de parámetros de la operación). Este es el único caso en el cual los lenguajes de programación permiten que un objeto posea dos operaciones iguales.

¿Porque se permite esta situación? En primer lugar, porque no se puede romper la característica principal de una herencia: la subclase hereda los atributos y operaciones de la superclase.

En segundo lugar, si bien tenemos dos clases diferentes, por la relación de herencia, la instanciación genera un único objeto. Es decir, cuando un lenguaje de programación crea un objeto de **TrianguloIsosceles,** el mismo poseerá 3 atributos y 2 operaciones.

Ante esta situación cabe la pregunta: Si se invoca la operación *getPerimetro()* ¿Cuál de las dos operaciones que posee se ejecutará?

Por defecto los lenguajes de programación asumirán que la operación que se ha definido en la subclase es una "**especificación**" de la definida en la superclase, por lo tanto, al poseer mayores detalles la lógica indica que se debe ejecutar, obviando de esta manera, la operación definida en la superclase. Este proceso se conoce como **sobreescritura**. Esto significa que existen ambas operaciones en la subclase pero que al momento de invocar el nombre de la operación se ejecutará la de la subclase, porque **sobre escribe o redefine** la de la superclase; dando el efecto o impresión de que la subclase solo posee el método definido en la subclase.

Formalmente los lenguajes de programación seguirán los siguientes criterios coherentes con la definición de herencia y sus procesos de especialización y generalización:

- Si se crean objetos de la subclase, la operación que se ejecutará será la definida en la subclase, ya que el proceso de especialización asume que esta operación redefine o sobreescribe la de la superclase.
- Si se crean objetos de la superclase, la operación que se ejecutará será la definida en esta, ya que el concepto de generalización supone que la superclase no conoce los atributos y operaciones de las subclases. En esto caso no se aplica la sobreescritura.

La sobre escritura supone un conjunto de beneficios:

- Es posible crear una nueva forma de ejecutar una operación sin necesidad de modificarla o eliminarla. Se redefine su funcionamiento en la subclase sin afectar la existente en la superclase. Esto facilita el mantenimiento del código. Lo único que se debe hacer es cambiar la instanciación de la superclase por la de la subclase.
- Dado que la subclase hereda todos los atributos y operaciones de la superclase, no debería ocurrir ningún error de integración de la subclase en el sistema, logrando de esta manera mantener la integridad del sistema a la vez que se optimiza la funcionalidad (y obviamente se podría agregar nueva funcionalidad en la subclase).
- Las operaciones redefinidas en una subclase pueden invocar la funcionalidad de la operación sobrescrita definida en la superclase. Es común que la propia operación de la subclase reutilice la funcionalidad de la operación sobrescrita en superclase para luego agregar una funcionalidad específica, con lo cual se estima que se disminuye el tiempo de desarrollo y se promueve la reutilización de funcionalidad.

Puede observar todos estos beneficios en acción en el video “programación de los beneficios de la sobrescritura”.

Clases abstractas

Debido a la herencia se establecen relaciones jerárquicas entre clases. Puede suceder que una superclase sea tan genérica que no tenga sentido instanciarla. En estos casos, se asume que la superclase posee un alto nivel de abstracción, lo cual provocará que no sea posible definir algunos o todos sus atributos ya que corresponden a alguna de sus subclases. Misma situación puede llegar a presentarse en las operaciones de esa clase. En este tipo de situaciones es conveniente definir clases abstractas. Una clase abstracta es aquella que por definición no se puede instanciar. Desde el punto de vista conceptual una clase abstracta es aquella que no permite una clase que esté más alta en la jerarquía de relaciones, por lo tanto, no podrá heredar de otra clase definida por el desarrollador. Observe como se modela una clase abstracta en el diagrama de clases:

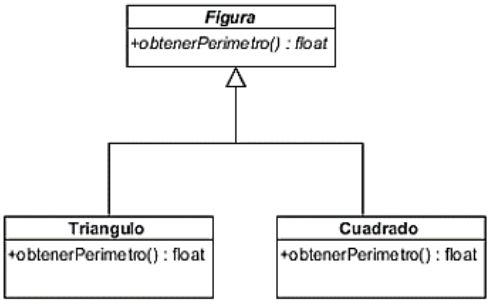

Figura 5. Modelado de una clase abstracta

Suponga que se crea un objeto de la clase **Figura** que se observa en la figura 5 y luego se invoca la operación *obtenerPerimetro*().

¿Cómo se determinaría cual de todas las posibles ecuaciones para obtener el perímetro de una figura se utilizaría?

Estamos ante una situación en la cual el desarrollador no puede determinar la ecuación que debe programar para obtener su perímetro. Esto sucede porque **Figura** representa a todas las figuras geométricas, por lo cual crear un objeto de este tipo significaría cometer un error en el diseño del diagrama de clases.

En estos casos lo que se espera es que se generen objetos de las subclases únicamente. En el ejemplo de la figura 5, esto representaría crear objetos de la clase Triangulo o Cuadrado, ya que para estos objetos está establecida la forma en que obtienen sus respectivos perímetros.

Así, **cuando no es posible crear un objeto de una clase, se dice que esa clase es abstracta**. En un diagrama de clases una clase abstracta se identifica con el nombre de la clase en cursiva (tal como está representado en la figura 5, Figura es una clase abstracta).

Observe además que en esta clase abstracta se ha definido una operación que también está en cursiva. ¿Será una operación abstracta? Exactamente esta operación es abstracta.

¿Qué es una operación abstracta? Es una operación que únicamente define su interfaz, pero no su implementación. La interfaz de una operación incluye el nombre, la lista de parámetros y el tipo de retorno de una operación. La interfaz es lo que utiliza el protocolo de mensajes para poder invocar operaciones. La implementación de una operación es el algoritmo que ejecuta la acción de la operación. Entonces, esto significa que una operación abstracta no definirá su algoritmo.

¿Entonces quien implementará ese algoritmo? Evidentemente lo implementarán sus subclases a través de la sobreescritura de las operaciones.

Esto proporciona una manera elegante de diseñar clases y facilitar su programación ya que:

1) Es posible indicar que una operación en la superclase no posee implementación (Por ejemplo, en `Figura` la operación *obtenerPerimetro*()).
2) Cuando se define una operación abstracta SE OBLIGA a que las subclases sobreescriban esa operación y la implementen. Esto genera lo que se denomina un contrato entre la superclase y la subclase por la cual la superclase encomienda a la subclase a implementar la operación definida como abstracta.

Nota importante: una clase abstracta no se puede instanciar, pero puede poseer operaciones comunes (no abstractas) y atributos.

TIPOS DE HERENCIA

La herencia es simple cuando una subclase hereda únicamente de una superclase. Por el contrario, una herencia es múltiple si la subclase hereda de varias 2 o más superclases.

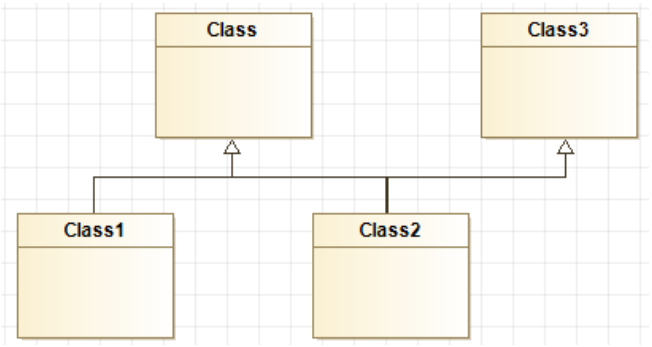

Figura 6. Tipos de Herencia

En la figura 6 se puede observar que Class1 hereda únicamente de Class, por lo cual estamos ante una herencia simple. En cambio, Class2 hereda a la misma vez de Class y Class3, lo cual significa que estamos ante una herencia múltiple.

LA HERENCIA EN LOS VIDEOJUEGOS

En el paradigma orientado a objetos, si bien es posible que el desarrollador cree clases donde se requiera la herencia, estas siempre convenientemente heredarán de GameObject. Esto se debe a que al menos dispondrán de un atributo que represente la posición del objeto y una operación para renderizarlo (dibujar, redibujar pintar o refrescar).

Entonces sería conveniente crear una estructura **similar** a la siguiente:

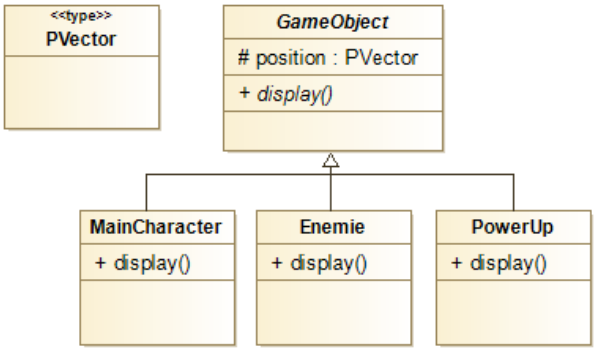

Figura 7. Esquema general de modelado de GameObjects

Como se puede observar, la figura 7 representa un esquema genérico donde todos los personajes del juego (MainCharacter y Enemie) así como sus recursos (PowerUp) son subclases de una clase abstracta (GameObject) que posee un atributo position y una operación abstracta (display()). Esta operación es sobrescrita por la implementación correspondiente a cada subclase, lo cual permitirá renderizar el objeto en virtud de su posición.

Podría pensar en una jerarquía donde los gameobjects dinámicos puedan ser subclases GameObject e incorporen una operación move(). De esta clase a su vez heredarían MainCharacter y Enemie; mientras que PowerUp directamente heredaría de GameObject debido a que no se mueve.

La complejidad de la jerarquía la definirá Ud. como diseñador.