Unidad 4

DIPLOMATURA EN PROGRAMACION JAVA

Capítulo 5

Herencia, Polimorfismo e Interfaces

Herencia, Polimorfismo e Interfaces

Herencia

Desde el punto de vista de una clase y considerándola como un módulo, la herencia es la técnica de reutilización por excelencia. Para resumir el concepto en pocas palabras, se puede definir a la herencia como:

> *"La capacidad de una clase de heredar los miembros de otra, pudiendo, inclusive redefinir los componentes heredados".*

Por eso se puede enunciar fácilmente la frase que caracteriza a la herencia:

> *Una clase hereda de otra cuando se define por medio de la frase **es un** o **es una**. También se dice que una clase si hereda de otra **es del tipo** de la clase que hereda.*

La razón por la cual una clase es del mismo tipo de que la que hereda es porque se deriva de esta y las clases forman un tipo de datos por si mismas

Si un módulo es un conjunto de servicios ofrecidos a módulos externos y **no** se tuviera la capacidad de heredar miembros de otras clases, cada proveedor (se debe entender como proveedor a la clase que brinda el servicio) debería definir por sí mismo todos los servicios posibles.

Más aún, si un módulo en su diseño necesita servicios que provee otro ya diseñado, se debería volver a escribir ese servicio dentro del mismo nuevamente. Este es uno de los principales motivos de la relación entre clientes (clases que invocan servicios de otras clases) y proveedores, no volver a escribir servicios existentes.

Este nuevo estilo para programar es completamente diferente a las aproximaciones tradicionales de la programación estructurada, teniendo el concepto sus raíces en técnicas utilizadas por la inteligencia artificial.

Si una clase puede tener acceso a ciertos miembros de otra e incorporarlos como propios, no tiene que volver a escribirse código previamente confeccionado. Pero en los lenguajes de programación, el tener acceso es siempre un problema de **visibilidad**. Por lo tanto, la cuestión se reduce a tener un cierto tipo de visibilidad que permita que los módulos sean abiertos y cerrados al mismo tiempo, es decir, que existan servicios que se puedan incorporar en otras clases como si fueran propios mientras los servicios que eran previamente internos lo sigan siendo y no sean accesibles. Se debe tener presente que los elementos inaccesibles de una clase **siempre** son sus atributos y servicios privados.

Los módulos deben ser pensados para ser reutilizados cada vez que se pueda en una etapa de diseño, por lo tanto se deben cuidar mucho las relaciones entre clases

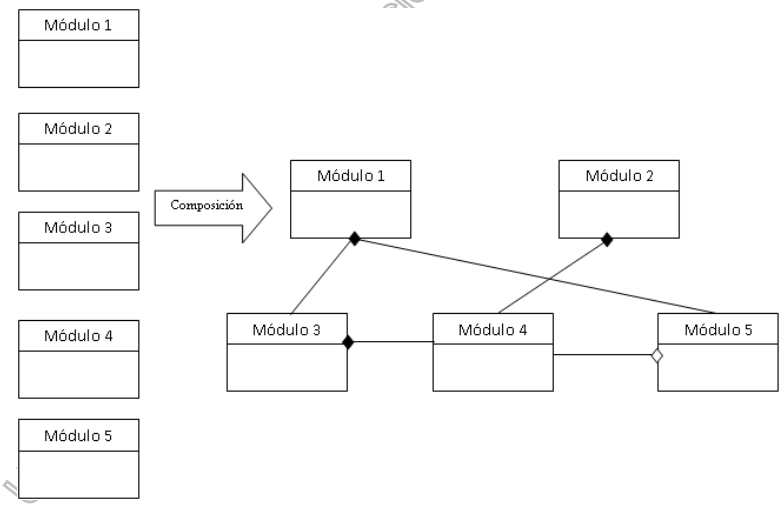
Si se tiene en cuenta las relaciones entre módulos, se deben pensar a éstos como lo suficientemente flexibles para poder ser implementados en cualquier nueva composición que se intente, aún si esta no tiene nada que ver con aquella para la que fue diseñada por primera vez.

Es claro que el criterio a tener apunta directamente a uno de los elementos más buscados en el desarrollo de software: la **reutilización**.

Un ejemplo del criterio son las librerías definidas por el usuario, ya sean éstas de subprogramas o funciones. Cuando se crean las mismas lo que se tiene en mente es la creación de elementos fáciles de componer en futuras aplicaciones.

Un hecho que viola abiertamente el criterio es el desarrollo de módulos específicos, ya que estos no podrán reutilizarse y, por ejemplo, el diseño top-down tiene esta tendencia, por lo que deberá utilizárselo con cierto criterio para no caer en módulos sumamente específicos, ya que estos tienen además un efecto negativo muy fuerte respecto de la continuidad en el desarrollo (al último estado de división realizado en un diseño top-down se suele llamar formato final de la función, lo que implica que no admite cambio).

No se debe confundir a las relaciones entre módulos con un criterio dependiente de la descomposición, ya que si bien éstos deben interactuar, ninguno debe condicionar al otro para lograr módulos genéricos.



Notar que el criterio señala indirectamente el cuidado de cómo los módulos se deben componer entre sí. Esto apunta claramente al diseño de las interfaces de cada módulo, las cuales serán el vínculo de unión con los otros.

Se puede enunciar un criterio y un principio para la ayuda de toma de decisiones de este tipo de la siguiente manera

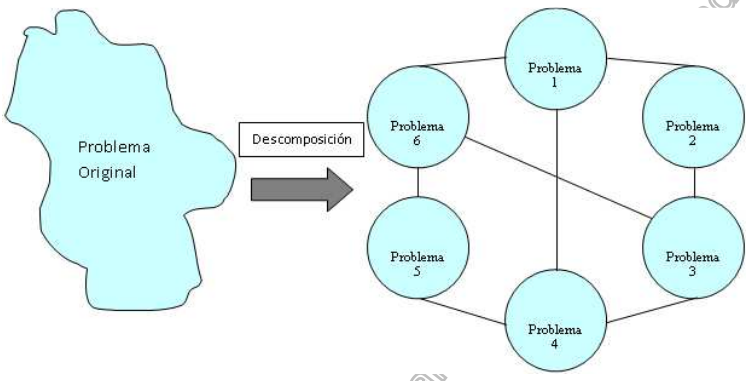
Criterio de descomposición en módulos

El criterio de descomposición se basa en el siguiente argumento:

> *Cuanto más chico es un problema, más fácil será de resolver*.

La idea surge del hecho que cuando un problema se subdivide en pequeños sub problemas, éstos son más fáciles de resolver por separado, quedando como paso final la interconexión entre los sub problemas que generan la respuesta del problema mayor.

El siguiente gráfico expone lo afirmado



Como se puede ver, de la complejidad inicial se construyen una serie de sub problemas resueltos interconectados entre sí. Este proceso se puede continuar, ya que algunos de los sub problemas se pueden volver a subdividir, reiniciando el ciclo. Los sub problemas considerarán módulos que se irán integrando entre sí.

Un ejemplo contrario a éste criterio son aquellos módulos que se enlazan con otros a través de una cohesión temporal. Esto quiere decir que son módulos, por ejemplo, de inicialización y preparación previa o reiniciación automática que contienen una variedad de funciones cuyo único elemento común es el de ejecutarse al mismo tiempo. El hecho de ejecutarse al mismo tiempo no es indicativo de compartir servicios, la funcionalidad brindada lo es. El tiempo es un factor que marca una secuencia de acciones, no a las acciones en si mismas.

La cuestión es ¿Cómo realizar correctamente este tipo de descomposición? La respuesta es bastante simple y se la viene utilizando a lo largo de las diferentes explicaciones. Se deben mantener firmes los conceptos que permiten tomar la decisión acerca de lo que es un objeto y lo que no, agregándole la técnica de revisar cuales objetos comparten funcionalidad o servicios en común.

Por ejemplo, se puede afirmar sin temor a equivocarse que una clase Empleado tiene servicios que son iguales a los de una clase Persona. Por ejemplo, ambos retornan un nombre o un apellido si se

les pregunta, o sea, ese servicio es común a los objetos de ambos tipos. No olvidar el uso de la frase que caracteriza a la herencia. De esta manera, un Empleado **es una** Persona

Principio del desarrollo de software abierto y cerrado

Este es quizás el principio que realiza la más tajante división entre la programación y diseño estructurado y las técnicas de orientación a objetos.

Si se utiliza bien el criterio de descomposición, los módulos a diseñar deben ser a la vez abiertos y cerrados.

> *Un módulo es abierto si el mismo se puede extender adicionándole elementos que se necesiten y que no se tuvieron en cuenta al momento de diseñar el módulo, esto es, haciéndolo más específico.*
>
> *Un módulo es cerrado cuando está disponible para ser usado por otros módulos sin necesidad de tocar los servicios internos de este, en otras palabras, tiene comportamiento encapsulado.*

Un ejemplo de módulo cerrado en los lenguajes tradicionales es cuando se crean librerías. En el caso particular de las diseñadas por el usuario, este puede crear funciones dentro de un archivo separado para luego ser utilizadas por un programa. Dicho archivo se compila y enlaza (linking) con un enlazador de librerías (a veces llamado programa bibliotecario por algunos autores o en inglés, lib manager). Luego cuando se enlacen los distintos programas se podrá reutilizar las funciones dentro de esta librería con sólo decirle al enlazador (linker) donde encontrarla, este buscará en el índice creado por el programa bibliotecario e incorporará el código de la función que se halle dentro del archivo de librería. Pero las funciones en sí mismas de la librería **siempre** son inalterables. La analogía es que se pueden invocar los servicios encapsulados de un módulo a través de sus interfaces públicas, pero nunca modificarlos.

Cuando se mencionó el principio de desarrollo de software abierto y cerrado, se aclaró que la única forma de implementar esto era a través de las técnicas de programación orientada a objetos las cuales son utilizadas al **heredar** miembros entre clases.

Recordando el principio, éste establece que una buena estructura modular debe ser a la vez abierta y cerrada por los siguientes motivos:

- **Cerrada**: porque los clientes que necesitan los servicios de los módulos para proseguir con sus propios desarrollos, no deberían ser afectados por cambios en los servicios internos del proveedor porque estos tienen acceso a ellos sólo a través de las interfaces públicas.
- **Abierta**: para poder incorporar como propio aquel servicio que no se haya incluido hasta el momento y que un módulo pueda necesitar, teniendo la facultad, inclusive, de rescribir dicho servicio para que funcione distinto dentro de él, ya sea total o parcialmente.

Este doble requerimiento que parece un dilema en la programación estructurada es satisfecho a través de la herencia.

Para entender bien el funcionamiento de la herencia podemos decir que mediante una definición sintáctica del lenguaje utilizado, los miembros de una clase pasan a formar parte de otra con ciertas restricciones. Las restricciones nacen para proteger criterios y principios ya expuestos como el de ocultamiento de la información. Analizando un poco más profundamente este punto, si algo es declarado, por ejemplo, privado de una clase y se mantiene lo enunciado, nada debería acceder a ello, ni siquiera la clase que pueda considerar sus elementos como propios.

Cuando un módulo hereda de otro y este a su vez sirve para que otro herede de él y así sucesivamente, se lo denomina **cadena de herencia**. Se enuncia entonces

> *Una **cadena de herencia** esta conformado por los sucesivos módulos que heredan unos de otros. Una **sub cadena de herencia** es una parte de una cadena de herencia.*

También nace otra necesidad a incorporar. Por ejemplo, ¿qué pasa si se presenta el caso que se quiera que un servicio interno o atributo se pueda acceder en un módulo que hereda los servicios de otro pero para aquellos que no heredan de él se vea como un servicio interno que no se pueda acceder?

Nuevamente se toma ventaja de una vieja conocida, la visibilidad. Se puede definir una nueva visibilidad, a la que por ahora llamaremos visibilidad protegida de un módulo, para que aquellos servicios o atributos definidos con dicha visibilidad sólo sean visibles para los que heredan del módulo y no puedan ser accedidos directamente desde fuera de la clase o desde fuera de la cadena de herencia.

Sin embargo, no se debe perder de vista que esto genera una nueva interfaz para una clase a nivel de su cadena de herencia, dejando en manos del diseñador el problema de analizar las interfaces para que sean claras y no se generen interfaces “ocultas” a simple vista que propaguen errores.

La herencia es la relación más fuerte de todos y genera un acoplamiento fuerte (definición de la necesidad que tienen de permanecer juntos de dos o más módulos relacionados), por lo que se debe tener sumo cuidado en el desarrollo de las interfaces.

Se puede extender el concepto de desarrollo de la estructura modular con software abierto y cerrado más allá de la modificación de las interfaces propias de un módulo a través de crear clases más específicas.

Esto involucra una forma de pensar diferente. Si un módulo necesita más elementos públicos para poder brindar los servicios necesarios, se deberá revaluar si no son dos módulos diferentes que comparten características y servicios similares, describiendo uno de ellos funcionalidades más genéricas y el otro más específicas.

Pensar en los servicios que presenta un dado módulo en términos que estos sean más **genéricos** o más **específicos** deja en manos del analista y el diseñador el trabajo de determinar si un módulo debe tener servicios de otra clase como propios, o tan sólo relacionarse con la clase y delegarle el trabajo de brindar dicho servicio. En otras palabras, se debe decidir si una clase hereda, asocia, agrega o compone los servicios de otra. Este es uno de los problemas fundamentales del análisis y diseño orientado a objetos.

Nuevamente es necesario enunciar un principio para el desarrollo de software como el siguiente

Claridad en el desarrollo de interfaces

En el diseño de un módulo deben ser claras las interfaces diseñadas, de manera tal que no se utilice el llamado "acople indirecto". Este tipo de unión se consigue cuando, por ejemplo, dos módulos comparten una variable o servicio y su manejo descuidado pueda ocasionar propagación de errores.

Si bien la interfaz no es visible para quien utilice el módulo y no hereda de él, es real y se propaga en una cadena de herencia, y el desconocimiento de la misma puede causar serios efectos colaterales.

En este principio se hace referencia a esto, de manera tal que si dos módulos se comunican entre sí debe ser claro el "acceso" o interfaz que utiliza uno con el otro, ya sea en su diseño o en su código.

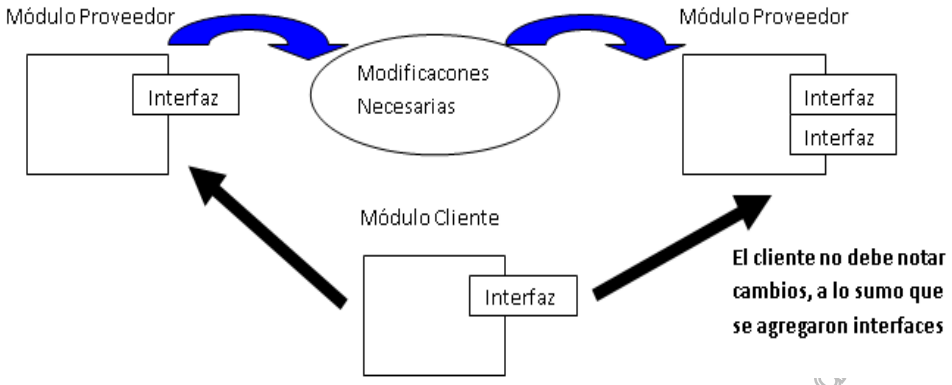
Para enunciarlo más formalmente, se dice que:

> *Cuando dos módulos cualesquiera se comunican, la acción debería ser obvia, tanto si se observa el texto de uno, del otro o de ambos.*

Como herramienta, muchos lenguajes modernos de programación implementan este principio mediante una limitación que no permite a las clases que heredan de otra bajar el nivel de visibilidad de un atributo o servicio. De esta manera, si algo es público y una clase hereda el servicio o atributo, no lo puede redefinir como privado o protegido.

La claridad en el desarrollo de interfaces permite crear buenos módulos abiertos y cerrados a la vez. La manera en que algo puede ser abierto y cerrado a la vez es a través de un diseño consiente de los módulos y su posible expansión. Con esto se implica el desarrollo de alguna nueva interfaz o la implementación de nuevos componentes privados que puedan expandir su funcionalidad, pero de manera tal que, las clases que usaban sus servicios anteriores a la modificación no deban ser restructuradas por los cambios que sufrió el módulo, ya que estas clases utilizaban un módulo y lo deben seguir viendo como que no hubiesen ocurrido cambios.

El gráfico muestra como un módulo cliente accede a un proveedor antes y después de ser modificado en el proveedor sin enterarse de dichas modificaciones.



Adaptabilidad de los módulos

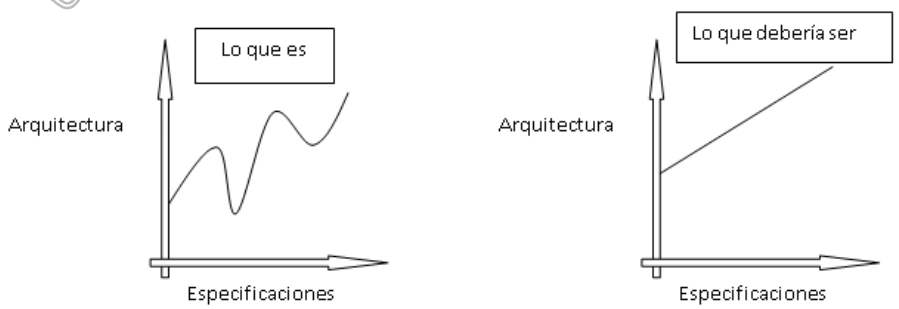
El desarrollo de un sistema, según se vio anteriormente, se basa en las especificaciones iniciales. Sería utópico pensar que las mismas se mantendrán constantes a lo largo del desarrollo del mismo.

Es muy común que los usuarios cambien las especificaciones a lo largo del desarrollo o se descubran problemas o condiciones no tenidas en cuenta, pero los métodos utilizados tradicionalmente fracasaban porque tendían a forzar a al usuario a no cambiarlas. Este es un ámbito incontrolable para el desarrollador, porque se basa en las muy aleatorias especificaciones de los usuarios y no en su muy controlado ámbito de diseño.

Sería ideal, por lo tanto, llevar el caso a un área totalmente dominada por el diseñador, lográndose esto a través de cambios que no produzcan alteraciones graves de diseño, lo cual se consigue previniendo en el desarrollo la creación de módulos que tengan la capacidad que, si se cambian las especificaciones, sólo afecten a un módulo o, a lo sumo, a unos pocos.

Un ejemplo análogo simple es definir las constantes con nombres simbólicos en los programas, así, si cambian las constantes, sólo habrá que redefinir dicho nombre y no recorrer todo el sistema cambiándolas.

La adaptabilidad está directamente ligada con la continuidad en el desarrollo, como se ve en los siguientes gráficos que muestran los cambios que puede sufrir la arquitectura de un sistema (módulos compuestos para formar el sistema) a causa del cambio de especificaciones.

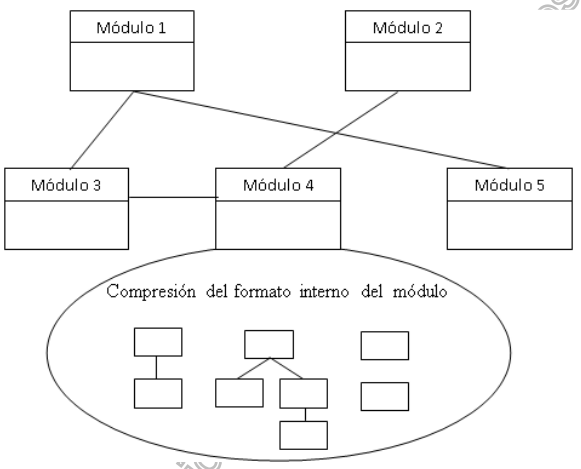


Criterio de mantener múltiples niveles

Cuando se aplica el criterio de descomposición se puede caer en un exceso perjudicial para la comprensión posterior de los módulos elaborados. Cuando se observa un módulo, éste se debe poder comprender sin tener la necesidad de observar otros.

Cuando este criterio es contemplado, el mantenimiento del software es mucho más sencillo y la comprensión del módulo es un factor fundamental cuando se introduce en él algún tipo de modificación.

Un ejemplo claro en contra de este criterio, es cuando los módulos dependen unos de otros por algún tipo de secuencia o acoplamiento. En este caso se debe recorrer la totalidad de ésta para comprender cada módulo dentro de ella.



Definiciones

En base a todo lo enunciado hasta este punto, debemos definir los nombres con que se tratarán a las diferentes clases según su desempeño en el diseño de un sistema:

- **Superclase o clase base**: es la clase que se heredará
- **Subclase o clase derivada**: es la clase que hereda de otra
- **Cadena de herencia**: es el conjunto de clases que se obtiene a partir de la creación de distintas subclases
- **Sub cadena de herencia**: cadena de herencia parcial a partir de una determinada clase
- **Base de la cadena o sub cadena de herencia**: es la clase a partir de la cual se crea la o las primeras subclases

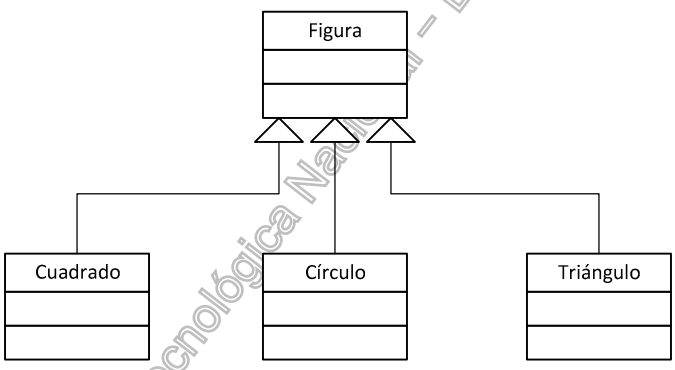
Se debe tener en cuenta que una subclase tiene una y solo una superclase. Si la superclase a la vez es subclase de otra, siempre seguirá siendo superclase de su derivada. Es común que se cometa el error de enunciar “la clase más base de otra” o algo por estilo, para definir aquellas clases que anteceden en una cadena de herencia. Esto es un error, puesto que si bien existen múltiples

niveles se acceden a ellos de a uno por vez tanto sea en el sentido de la superclase como en el de la subclase.

Suponiendo que se tiene una clase denominada Figura y se descubren dos abstracciones más, llamadas Círculo y Cuadrado. Es claro que un Círculo **es una** Figura y un Cuadrado **es una** Figura. Por lo tanto se pueden definir DOS cadenas de herencia, una entre Figura y Cuadrado y otra entre Círculo y Figura.

Retomando el ejemplo de las figuras, podemos querer añadir una nueva figura, que podría ser, un triángulo. La clase, según fue diseñada, puede haber sido utilizada por otros módulos y si efectuamos cambios, sobre todo en su interfaz, NO se deberían alterar aquellos clientes que la utilizaron. Por otro lado, si heredamos la clase, obtenemos una nueva que accede a los miembros de la anterior tanto como a los propios siempre y cuando los mismos sean **visibles**.

Si se quiere representar a la herencia en UML esto se realiza mediante una flecha con la punta en blanco que apunta a la clase base o superclase y sale desde la sub clase o clase derivada, como muestra el gráfico:



Los constructores en la herencia

Al igual que cualquier elemento privado de una clase o módulo, un constructor por más que sea público pertenece a la clase en donde fue declarado (en realidad no tiene sentido pensar en un constructor fuera de la clase donde se declaró porque ¿qué construiría?), por lo tanto cuando se crea una instancia de una subclase, esta debe llamar al constructor de la clase base antes de realizar cualquier otra operación (primero se construye la base y a partir de ella se construye la subclase). Si la clase base, a su vez, es subclase de otra, antes de realizar cualquier operación debe llamar al constructor de "su clase base".

Esta operatoria se propaga a lo largo de la cadena de herencia hasta llegar a la clase base de la cadena de herencia. Una vez terminada la construcción de la clase base de la cadena de herencia,

se invierte el proceso para terminar de construir las subclases a partir de ella hasta llegar al constructor de la clase de la cual se desea crear una instancia.

También se puede presentar la siguiente inquietud antes mencionada: ¿cómo hacer que un elemento de una clase permanezca privado para el mundo exterior (no se pueda acceder por un módulo cliente) y sea accesible para las subclases que se creen a partir de esta? Con los elementos definidos hasta el momento eso no es posible. Sin embargo muchos lenguajes de programación incluyen una declaración más de visibilidad que se agregan a las de público y privado, PROTEGIDO. Cuando un elemento es declarado como protegido será accesible a cualquier elemento de la cadena de herencia a partir de su declaración, pero no fuera de esta.

Generalización y especialización

Cuando se diseñan clases se pueden descubrir distintos objetos candidatos en un sistema los cuales pueden agruparse según sus servicios. Otra forma de agruparlos es a través de evaluar los que tienen similares atributos y operaciones (servicios) para decidir si son del mismo tipo o no.

En el caso de ser del mismo tipo se puede diseñar con ellos una cadena de herencia en la que participen clases que definan los tipos de los objetos descubiertos.

Cuando se realiza esto, se pueden dar dos casos bien definidos:

- Dada una clase que define un objeto candidato, se descubre una clase que puede ser superclase de esta. A esto se lo llama **generalización** porque define una clase más genérica que la actual.
- Dada una clase que define un objeto candidato, se descubre una clase que puede ser subclase de esta. A esto se lo llama **especialización** porque define una clase más específica que la actual.

Tanto la generalización como la especialización pueden descubrirse a partir de los objetos candidatos o ser parte de ellos, ya que a través de estos se descubren las clases que definen sus tipos. El “arte” del diseño consiste en hallar estas relaciones, definirlas y caracterizarlas por sus tipos y, en caso de no ser parte de los objetos candidatos, definir los tipos que sean necesarios por medio de generalizar o especializar para ir armando las diferentes cadenas de herencia.

Ejercicios 1 y 2

Los temas de los que se tratan en este ejercicio son los siguientes:

- Herencia
- Generalización
- Especialización

En este ejercicio se busca afirmar los conceptos enunciados creando cadenas de herencia

Sobrecarga y rescritura

Sin entrar en mayores detalles, anteriormente se efectuaba una sobrecarga cuando se ponía más de un constructor para una clase. Cuando se declaraba en ejemplos anteriores más de un constructor, se usaba sobrecarga.

Una sobrecarga en sí es, cuando dos o más funciones tienen el mismo nombre y el compilador diferencia entre una y otras, de la siguiente manera:

- Tienen distinta cantidad de argumentos.
- Por los menos un argumento difiere en el tipo.

Cuando al menos una de las reglas anteriores se cumple, se está en presencia de una sobrecarga.

Cuando la sobrecarga se da dentro de una clase, es fácil seguir como se escribieron los métodos, pero cuando se esta en presencia de la herencia, se debe tener cuidado, porque si los métodos sobrecargados se pueden acceder dentro de la cadena de herencia y se cumplen las condiciones anteriores, dicha sobrecarga se mantiene a lo largo de la cadena.

Cuando un método en una subclase es **exactamente igual, incluyendo el valor que el mismo retorna**, a otro en la superclase, se dice que el mismo esta **rescrito** o **sobrescrito**. Si la clase Figura definiera un método para calcular el área de la figura, la clase Circulo se vería obligada a redefinir ese cálculo que es particular de los círculos y diferente de otros cálculos de área. Cuando se cree un objeto del tipo círculo y se invoque el cálculo del área, la operación a realizar será aquella que renombró dicho cálculo y no el de la clase base. Por lo tanto, rescribir un método en una subclase tiene el efecto de "ocultar" dicho método de la superclase cuando se crea una instancia de ella, ya que tendrá prioridad de acceso el método que rescribe sobre el rescrito.

El siguiente ejemplo en seudocódigo trata de clarificar lo enunciado

Clase Figura

```
CLASE Figura {

        //por el momento no se pone nada en esta clase

        PROTEGIDO: REAL resultado
        PUBLICO: FUNC Figura(){
                        resultado = 0
                        }
        PUBLICO: FUNC calculaArea(){
                        }
        PUBLICO: FUNC muestraResultado() {
                        IMPRIMIR "Esta es la superclase"
                        }
```

```
} // Fin de la declaración de la clase Figura
```

Clase Rectangulo

```
CLASE Rectangulo HEREDA Figura {

        PRIVADO: REAL X1, X2, Y1, Y2

        PUBLICO: FUNC Rectangulo(REAL a, REAL b, REAL c, REAL d){
                        X1 = a
                        X2 = b
                        Y1 = c
                        Y2 = d
                        }
        PUBLICO: FUNC calculaArea(){
                        resultado = ( x2 - x1 ) * ( y2 - y1)
                        }
        PUBLICO: FUNC muestraResultado() {
                        IMPRIMIR "El rectángulo con vértices (",X1,",",Y1,"),
                        (",X2,    ", ", Y2, ") Tiene área", resultado
                        }
} // Fin de la declaración de la clase Rectángulo
```

Clase Circulo

```
CLASE Circulo HEREDA Figura {

        PRIVADO: REAL radio, x, y

        PUBLICO: FUNC Circulo(REAL e, REAL f, REAL g) {
                        radio = e
                        x = f
                        y = g
                        }
        PUBLICO: FUNC calculaArea(){
                        resultado  = 3.1416 * radio * radio
                        }
                PUBLICO: FUNC muestraResultado(){
                        IMPRIMIR "El círculo de centro (",x,",",y,") y radio  ",radio, " ",
                                "Tiene área", resultado
```

```
                    }
} // Fin de la declaración de la clase
```

Programa principal

```
PROGRAMA

    Rectangulo rec( 3.18, 9, 44.5, 6)
    Circulo circ( 3, 4, 4)
    rec.calculaArea()
    rec.muestraResultado()
    circ.calculaArea()
    circ.muestraResultado()

FIN PROGRAMA
```

Clases abstractas

En el ejemplo de herencia se utilizó como superclase a Figura sin poner dentro de ella a ningún elemento en el método calculaArea. La razón para esto fue exclusivamente de diseño ya que una figura es un concepto "más genérico" que un círculo o un rectángulo. Los métodos pueden ser pensados para ser rescritos en las subclases, como en el ejemplo anterior calculaArea, ya sea por ser específicos de las subclases o por no conocer suficiente de esta operación al momento de diseñar la clase. Cuando un método no tiene cuerpo (bloque de sentencias) se lo puede nombrar como abstracto y a toda la clase que lo posee también, con la siguiente limitación principal:

> ***No se puede crear una instancia de una clase abstracta, pero si se puede crear una referencia a un objeto de su tipo porque la referencia no es el objeto en si mismo***

Esto es muy importante porque en una clase de este tipo tiene que no estar definido algún método.

Cuando una clase es diseñada como abstracta, el módulo que la quiera utilizar esta obligado a heredarla y sobre escribir sus métodos abstractos.

Se puede definir en este punto también, como son las clases que no son abstractas, para ello se enuncia:

> *Una clase concreta es aquella en la que todos sus métodos están bien definidos y se pueden crear objetos de su tipo. Los métodos pueden no tener instrucciones dentro de ellos* ***pero siempre*** *tienen un bloque de sentencias bien definido.*

La última oración define que los métodos o servicios tienen que declarar donde empiezan y donde terminan las declaraciones de sus componentes (variables, métodos, instrucciones, etc…). Cuando

un método o servicio no indica su bloque o cuerpo de sentencia, debe ser declarado como **abstracto** según las reglas de la gramática del lenguaje en particular.

Cuando se diseñan clases, un problema común es descubrir métodos que no se pueden implementar porque la operación que representan es muy genérica y se necesitan más detalles para describir su funcionalidad. A este tipo de funcionalidad "incompleta" (se sabe que se necesita pero no se puede especificar en este momento como trabaja la misma) se la denomina abstracta porque no se puede especificar en ese punto. Por lo tanto, las clases que poseen estos métodos (servicios u operaciones) son clases que contienen funcionalidad incompleta y de las que no puede crear una instancia en el sistema.

Muchos autores denominan a estas clases como "clases para ser heredadas", puesto que se necesita más información para convertirlas en concretas (clases que tienen todos sus métodos bien definidos). Una clase que hereda de otra que es abstracta hereda todos los métodos, incluyendo los abstractos. Por lo tanto se enuncia

> *Cualquier clase que hereda un método abstracto es considerada abstracta a menos que los métodos de este tipo se rescriban dentro de la subclase.*

Se utilizan cuando la forma de implementar un método estará disponible luego de especializar la clase.

Por ejemplo, si se tiene en cuenta el diseño de la clase Figura, cuando se creó dicha clase se puede definir claramente que una figura puede ser dibujada, lo cual indica que la clase necesita implementar un método "dibujar". Sin embargo, figura es un concepto muy *genérico* como para poder determinar "en qué forma se debe dibujar". Cuando otra clase herede de ella y el concepto sea más específico, como por ejemplo, una subclase de Figura podría ser Círculo, se puede especificar claramente cómo llevar a cabo la operación "dibujar".

Otro caso claro es el cálculo del área. En el ejemplo por simplicidad no se planteó que no hay información suficiente en Figura para saber cómo se calcula el área. Es necesario que una clase más específica desarrolle la forma de hacerlo.

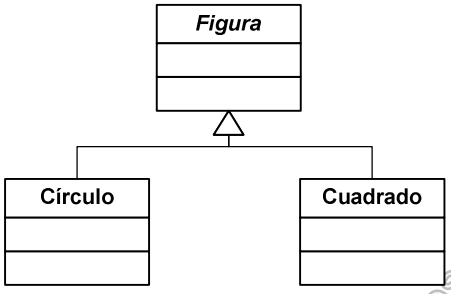
Los diagramas UML para representar clases abstractas son iguales que los diagramas de clases concretas, con la salvedad que los nombres de estas clases se diferencian por el formato del tipo de letra que tiene, el cual va en itálica.

Además, los métodos abstractos definidos en ellas también se escriben con letra itálica.

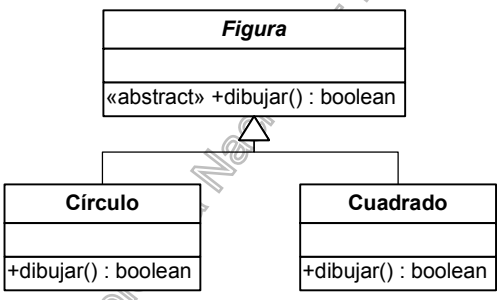
Existen alternativas a este formato, dado que muchas veces por imposibilidad de las herramientas de diseño utilizadas o por claridad visual, no es aconsejable usar el tipo de letra itálica. Dicha alternativa es poner la palabra abstract entre llaves luego de la declaración del método o un

estereotipo (palabra encerrada entre << >> que define el elemento declarado) antecediendo a la declaración. Estas reglas son válidas también para el nombre de la clase.

Por lo enunciado anteriormente, si se quiere diagramar en UML una clase abstracta Figura de la cual heredan dos clases concretas, Círculo y Cuadrado, las cuales forman dos cadenas de herencia, el diagrama sería como muestra la siguiente figura:



Si en el diagrama de clases se quiere representar el método abstracto dibujar y como este es rescrito en las subclases Circulo y Cuadrado, el diagrama que se obtendría sería como el siguiente:



Ejercicio 3

Los temas de los que se tratan en este ejercicio son los siguientes:

- Clases abstractas

En este ejercicio se busca afirmar los conceptos enunciados creando cadenas de herencia que posean clases abstractas

Polimorfismo y enlace o ligadura dinámica

Hay acciones u operaciones que no tienen sentido sin el objeto al cual se aplican. Por ejemplo, cuando en una conversación se dice "comer", se entiende perfectamente la acción a realizar pero no tiene un sentido concreto sin el objeto al cual se aplica la misma. De esta manera, lo primero que se responde a tal afirmación es "¿comer qué?", entonces la opción podría ser, por ejemplo, "comer una manzana". Sin el objeto manzana, comer es una acción pero no esta definida sobre qué actúa. En base a esto se puede definir:

> *Cuando para realizar una acción se debe especificar el objeto sobre la que se realiza para producir un hecho concreto, dicha acción debe estar definida dentro del objeto como un servicio ya que esta operación esta determinada por su comportamiento.*

De esta manera, la acción de comer del ejemplo esta determinada por el objeto manzana "*porque es el objeto el que se puede comer dado que brinda este servicio"*, por lo tanto debe existir una operación *dejarseComer* definida dentro del objeto manzana para poder indicar "manzana punto dejarse comer - Ej: manzana. dejarseComer()" y produzca un hecho concreto, el de comer la manzana.

De la misma manera, un objeto del tipo Vidrio puede no incluir un servicio de este tipo si no tiene sentido en el dominio de la solución. Por ejemplo, el contexto indica que se modela la forma en la cual se alimentan los seres humanos. Está claro que modelar un servicio para "dejarse comer" del objeto del tipo Vidrio sale del contexto (el menos en la mayoría de las situaciones normales).

Supongamos ahora que estamos modelando un objeto del tipo Fakir, que efectivamente "come vidrio". En este caso hay que modelar un servicio como el de manzana, pero el resultado de invocarlo es claramente diferente al de manzana, por lo tanto, en el único lugar que existen los servicios que determinan el estado y comportamiento de una operación de este tipo es en el mismo objeto que define el servicio "dejarse comer" para que opere según el caso.

A lo largo de la vida de un sistema se presentan muchas acciones que estarán bien definidas sólo cuando se apliquen a objetos concretos, esas acciones se las denomina polimórficas porque se pueden aplicar a distintos objetos y producir resultados diferentes dependiendo siempre de como brinda el servicio dicho objeto.

Es claro que esto sólo se puede definir en el momento en que se realiza la operación, por lo tanto se debe relacionar la acción con el objeto en el momento que se efectúa. En términos de un lenguaje de programación da la idea de "realizarlo en tiempo de ejecución del programa". Esto en realidad sucede así, ya que por técnicas de programación de lenguajes se permite que éste asocie las referencias en tiempo de ejecución. A este tipo de asociación en tiempo de ejecución se lo denomina **enlace o ligadura dinámica**.

Para sacar provecho del polimorfismo se debe dominar el diseño de métodos que se sobre escriban a lo largo de una cadena de herencia y que los mismos sean los que definan las operaciones polimórficas. Por ejemplo, supongamos que en el ejemplo de las figuras geométricas queremos heredar de círculo para crear la clase Esfera. Además, queremos crear un método dibujar() que la dibuje en pantalla. Si las figuras geométricas son para una pantalla, este método debería ser abstracto, estar definido en Figura y ser sobre escrito en Círculo y Esfera, ya que cada uno de estos objetos se dibujan diferente en una pantalla.

Luego debe existir un método más que invoque el servicio apropiadamente según el objeto sobre el que opera. Si por ejemplo existe un servicio llamado crearFigura(Objeto o) que recibe como parámetro el objeto al que se le va a solicitar el servicio, en el bloque de sentencias que posee se puede realizar la invocación o.dibujar() y la acción se llevará a cabo **según el objeto o que se haya recibido como parámetro**. Si el parámetro es un círculo, se invoca al servicio dibujar() definido en la clase círculo. Si el parámetro es una esfera, se invoca al método (servicio) dibujar() definido en la clase Esfera.

Como se puede ver, a través de la sobre escritura se puede sacar ventaja de la capacidad de realizar polimorfismo y, aunque no es excluyente, debería ser un objetivo cada vez que se diseñe un sistema.

Conversiones de tipo

¿Cómo sabe el método crearFigura() cuando llamar al servicio de Circulo o al del Esfera? La respuesta es simple pero su justificación no es clara a primera vista. La razón es que el método sabe “el tipo del objeto” y llama al servicio adecuado. Pero entonces, ¿cómo hace para saber el tipo del objeto?

Esta respuesta ya es un poco más complicada porque de fondo lo que ocurre es que se hace una **conversión de tipo sin que el objeto pierda su identidad**. Esto es como decir que si se le asigna a un entero largo (long en muchos lenguajes) un valor entero, se realiza la asignación por medio de una conversión de tipo pero de alguna manera lo que esta guardado adentro **sabe** que es un entero. Por lo tanto, si se llama a un servicio del valor que esta guardado, el servicio que responde es el del “entero” y no el del “entero largo”. Pero como ambos son del tipo “entero” y el entero largo es más grande, la conversión de tipo está bien definida.

Dejando la analogía de lado se puede enunciar lo siguiente

> ***Las conversiones de tipo entre referencias a objetos están bien definidas si estos pertenecen a una misma cadena de herencia. Sin embargo cada objeto después de la conversión conserva su tipo original que es el de la clase que lo define***

¿Cuál es entonces la conversión de tipo definida? Es la que hace que una referencia a un objeto se asigne a otro **porque son del mismo tipo**. Por ejemplo, la clase Circulo es del tipo Figura, por lo

tanto se puede asignar un objeto del tipo Circulo a uno del tipo Figura, pero no a la inversa. Volviendo a la analogía anterior se lo puede comparar diciendo que en lugar de justificar la conversión mediante “es más grande” se lo hace mediante “es más genérico”. Figura es un tipo más genérico que Círculo o Esfera, por lo tanto **abarca** el concepto que define a estas dos clases.

Cuando se realiza el camino inverso en la conversión, la misma es forzado (no natural de un lenguaje de programación) y se debe indicar explícitamente. Recurriendo nuevamente a la analogía, es como indicar que cuando se convierte de entero largo a entero debo realizar un truncamiento porque el espacio de almacenamiento es menor.

Se puede definir entonces

La conversión de tipo y la rescritura son los fundamentos del polimorfismo

Interfaces

Existen situaciones en las cuales cadenas de herencia diferentes prestan el mismo servicio para los objetos que se declaren de su tipo y se quiere implementar una lógica que los pueda invocar a cualquiera de ellos.

Por lo visto hasta el momento, existen situaciones en las cuales se puede implementar un llamado especial que permite invocar al método apropiado según el objeto sobre el cual se pide el servicio, a lo que se denominó como polimorfismo.

Sin embargo, el polimorfismo estaría limitado a la cadena de herencia donde se rescriben los métodos invocados y, salvo que las cadenas de herencia tengan una superclase en común, no se podría implementar un llamado polimórfico.

La única solución posible sería entonces que todas las clases que intervienen en este tipo de llamados tengan una superclase en común. Sin embargo, en muchísimas situaciones esto es conceptualmente incorrecto porque la conversión de tipo se hace a uno tan genérico que no indica la naturaleza del servicio polimórfico, sino sólo que se puede realizar.

La otra solución es permitir que las clases puedan tener más de una superclase, para así poder “conectar” aquellas que tengan servicios similares a través de una superclase en común, más allá de la que ya tengan definida.

Esta última solución es inviable en lenguajes que tiene herencia simple. Sin embargo, existe como alternativa un tipo de clase especial llamada **interfaz** que puede definirse como si fuera una herencia independientemente del mecanismo de herencia simple que pueda implementar una clase. Para diferenciar este tipo particular de herencia, las interfaces tienen una limitación importante:

Todos los servicios definidos en una interfaz son abstractos y no se pueden declarar en ella atributos

Una interfaz es el único medio de acceso de una clase a otra cuando estas no están directamente participando de una relación mediante una cadena de herencia, porque actúa como si fuera una superclase en común para ambas clases, lo cual permite realizar referencias polimórficas.

Al utilizar interfaces para que funcionen como superclases en común entre clases, o, inclusive, en cadenas o sub cadenas de herencias completas, se generan "puentes" entre las cadenas de herencia a través de los métodos de la interfaces que deben ser rescritos por ser abstractos.

Por lo tanto, una interfaz se puede utilizar conceptualmente, para agrupar clases que brindan servicios similares porque, si bien no se pueden declarar objetos del tipo de la interfaz, se pueden crear referencias con estas y aprovechar las conversiones de tipo.

Las clases abstractas, como se explicó anteriormente, pueden tener métodos abstractos, los cuales deben ser rescritos en las subclases, y métodos concretos.

Las interfaces existen en los lenguajes orientados a objetos modernos, pero pueden ser pensadas como clases abstractas puras también, porque todos sus métodos son abstractos y no pueden tener métodos ni variables implementados en ellas, sólo constantes. Esta es la principal diferencia entre las clases abstractas y las interfaces, pero su funcionamiento, salvo esta excepción, es el mismo.

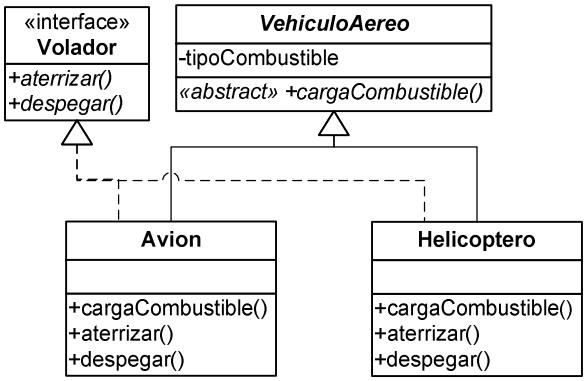
Como las interfaces sólo declaran los métodos en común para las clases que las implementan, evitan los problemas derivados de la herencia múltiple.

Por otra parte, se debe tener en cuenta que todos los métodos que pertenecen a una interfaz, funcionan como si fueran abstractos, por lo tanto, deben ser obligatoriamente sobrescritos en las clases que la implementen

El lenguaje de modelado UML usa un diagrama particular para las interfaces, pero por lo general se diagraman con el estereotipo <<interface>> en el nombre de la clase y la clase que la implementa debe tener una flecha de relación similar a la de la herencia salvo que la línea es punteada.

Por ejemplo, se puede suponer una clase llamada VehículoAereo, la cual es abstracta y dos subclases de esta, Avión y Helicóptero. Estas subclases forman cada una cadena de herencia independiente con la misma superclase, por lo tanto se podrían realizar las referencias polimórficas como se explicaron anteriormente. Pero se puede suponer, que por razones de diseño, se desea agrupar los métodos en común para ambas cadenas de herencia en una interfaz que implementen las clases.

Un posible diseño se muestra a continuación



Se puede observar que además de rescribir el método abstracto de la superclase, las clases concretas están obligadas a rescribir los métodos de la interfaz

Ejercicios 4 y 5

Los temas de los que se tratan en este ejercicio son los siguientes:

- Polimorfismo
- Interfaces

En este ejercicio se busca afirmar los conceptos enunciados creando servicios polimórficos e interfaces entre cadenas de herencia.