Tema 4

Herencia ,clases abstractas y polimorfismo.

<<cambios pag 505 libro Meyer><

UPSAM 2010-2011

Índice de contenidos

1. Herencia, clases abstractas y polimorfismo.

1.1.Generalización

Según la DRAE el significado de generalizar es el siguiente:

> *Abstraer lo que es común y esencial a muchas cosas, para formar un concepto general que las comprenda todas.*

El proceso de generalizar es fundamental en ciencia, constituye la base de la creación de reglas Universales.

La generalización se constituye así como el proceso de abstraer y representar lo común. En este proceso siempre diferencia dos roles: *lo que es general , y lo que es específico y puede ser representado por lo general*. Veremos más adelante como la generalización y especialización crea una nueva relación entre clases .Además la generalización **es una relación mucho más fuerte que la asociación**, de hecho, la generalización implica el nivel más alto de dependencia entre dos clases (y por consiguiente de **acoplamiento**).

Generalización de clases

Conceptualmente la generalización es una idea muy simple. Pensando en cosas generales encontramos conceptos como por ejemplo el de asignatura , árbol ,etc.., y en algo más concreto encontramos la asignatura de programación, el abeto, el pino ,etc. Son conceptos relacionados entre sí , pero a distinto nivel de detalle. Si pensemos en los razonamientos y propiedades aplicados al concepto general serian aplicables a los conceptos más específicos.

En la **Figura 1**, tenemos una clase general denominada *Forma* , que como se puede comprobar es un concepto bastante general. A partir de este concepto derivamos otros más específicos, que son variantes de la idea general *Forma*.

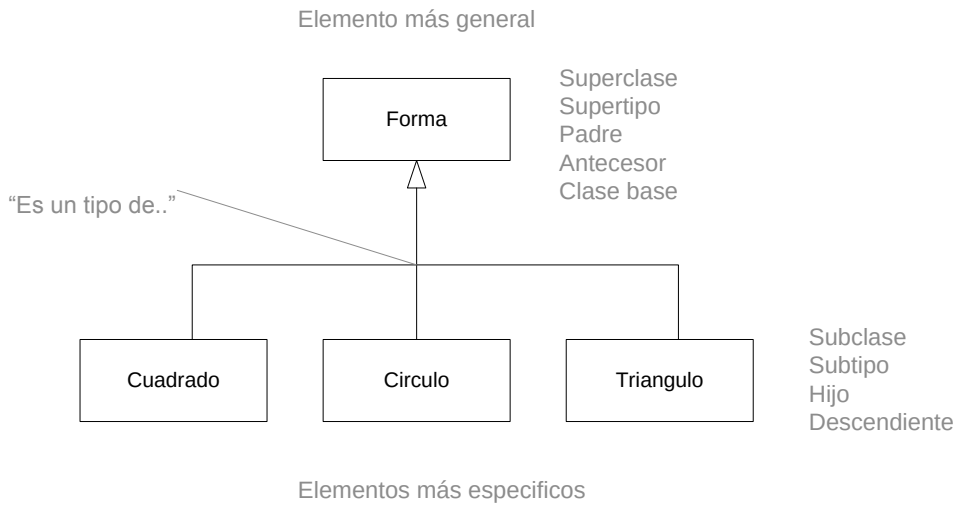


Figura 1 Generalización

Existen dos procesos por los que podríamos llegar a esta jerarquía: a partir de un proceso de especialización o por medio de la generalización. En la especialización, identificaríamos primero el concepto general *Forma* y posteriormente veríamos la necesidad de buscar o definir conceptos más específicos. En la generalización, podríamos identificar una serie de clases como *Cuadrado, Circulo y Triangulo* y encontrar que comparten atributos y comportamiento, lo que daría lugar a una clase más general.

De manera general, podemos definir la relación de generalización como un tipo de relación en la que una clase (clase general) generaliza el comportamiento y las propiedades de otras clases (subclases). De otra manera, podemos decir que las subclases especializan el comportamiento de la clase general.

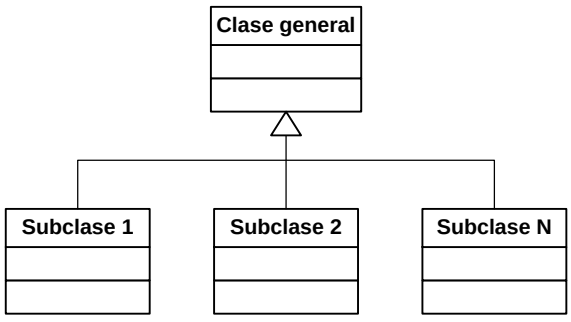
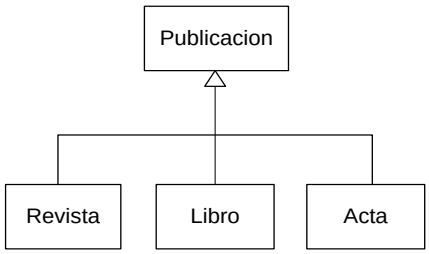
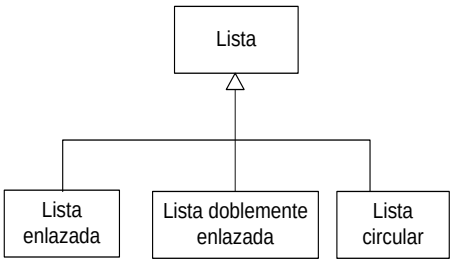


Figura 2. Generalización/Especialización en notación UML

Algunos ejemplo de relaciones Generalización/especialización.


Publicacion
Revista
Libro
Acta


Lista
Lista enlazada
Lista doblemente enlazada
Lista circular

2. Herencia

Cuando se crea una jerarquía como la de la **Figura 1**, implícitamente **existe una relación de herencia** entre los participantes, por que **las subclases heredan** todas las características de sus superclases. Para ser más específicos en relación con la programación orientada a objetos, las subclases heredan:

- Atributos
- Operaciones
- Relaciones
- Restricciones

Las **subclases también pueden añadir nuevas características o reescribir las operaciones** de las superclases.

2.1. Reescritura de métodos

En el ejemplo mostrado en **Figura 2** las subclases *Cuadrado y Círculo* **heredan** todos los atributos, operaciones y restricciones de la superclase *Forma*. Esto significa que aunque no podamos observar estas características en las subclases, estas están implícitas. A nivel conceptual podemos decir entonces que un *Círculo* y un *Cuadrado* son un tipo de *Forma*.

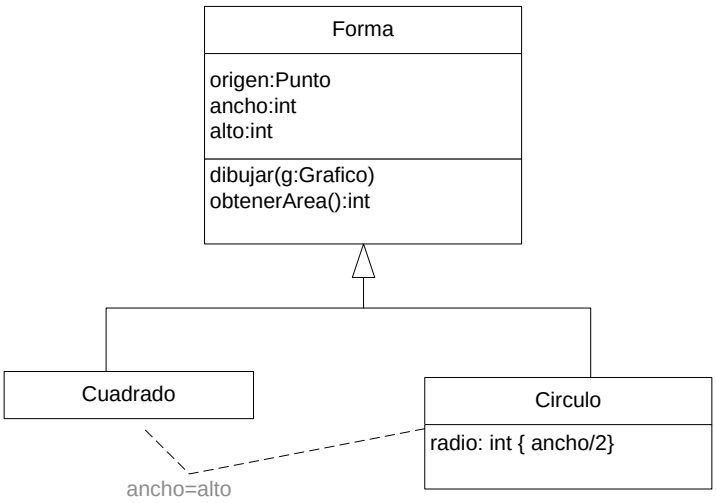


Figura 3. Jerarquía y herencia

En este ejemplo (Figura 3) la clase *Forma* define dos métodos denominados *dibujar(…)* y *obtenerArea()*. Estos métodos podrían no ser apropiados para la clase *Círculo* o *Cuadrado*. Es de esperar que se dibuje un cuadrado cuando se envié el mensaje

dibujar( ) a un objeto de la clase *Cuadrado* y un *Circulo* cuando se envíe este mismo mensaje a un objeto de la clase *Círculo*. Por tanto la operación *dibujar()* definida en la clase base y que ambas subclases han heredado no sirve, de hecho, esta operación podría no dibujar nada en absoluto ¿qué aspecto tiene una forma?. Los mismos argumentos son aplicables a la operación *obtenerArea()* . ¿Cómo podemos calcular el área de una forma no definida?

Estos problemas indican claramente la **necesidad de permitir que las subclases sean capaces de cambiar el comportamiento de superclase**. La subclase *Cuadrado* y la subclase *Circulo* necesitan implementar sus propio comportamiento para las operaciones *dibujar(...) y obtenerArea(...)* y sobrescribir el comportamiento por defecto heredado de la clase base para proporcionar un comportamiento más especifico y apropiado.

La **Figura 4** muestra esta acción: las subclases *Circulo* y *Cuadrado* proporcionan sus propias operaciones para *dibujar() y obtenerArea()* .

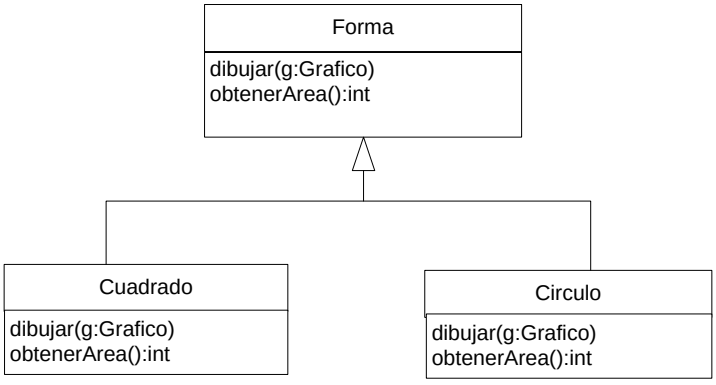


Figura 4 **Reescritura de métodos heredados**

Para **reescribir** una operación de la clase base, **una subclase debe proporcionar una operación con exactamente el mismo prototipo de la operación de la clase base.**

2.2. Clases abstractas

Las relaciones de generalización, introducen a nivel conceptual **clases, que más que clases de implementación son elementos o conceptos de organización.** En muchas generalizaciones, será necesario delegar la implementación de las operaciones a las subclases. En nuestro ejemplo la operación *dibujar (...)* de *Forma* es un buen ejemplo, porque no sabríamos como pintar una *Forma* sin determinar. El concepto de dibujar una forma es demasiado abstracto para tener una implementación.

Se dice entonces que una **clase tienen operaciones abstractas o un comportamiento abstracto**. Podemos indicar esto **haciendo que el método sea abstracto**.

Debido a que las **clases generales** con métodos abstractos **no tienen un comportamiento definido(**es decir **no tienen una implementación para sus métodos**), e**stas clases no pueden ser instanciadas. A este tipo de clases** se las denomina **clases abstractas.**

En la **Figura 5,** tenemos una **clase abstracta** llamada *Forma*, con dos métodos abstractos *dibujar(…) y obtenerArea().* La implementación para estas operaciones las proporcionan tanto la subclase *Circulo* como la subclase *Cuadrado*. Aunque la clase abstracta *Forma* es incompleta, y no puede ser instanciada, sus subclases si son completas cuando son instanciadas.

Existe una serie de **ventajas** en la utilización de **clases abstractas:**

- Se pueden definir un conjunto de **operaciones abstractas** en la clase base clase abstracta de forma que deben ser implementadas por las subclases. Se puede ver esto como **"un contrato"** que todas las subclases deben cumplir (o implementar).
- Se puede escribir **partes de código** que manipulen *Formas* y de acuerdo al principio de substitución esta parte de código se podrá **usar con cualquier subclase**.

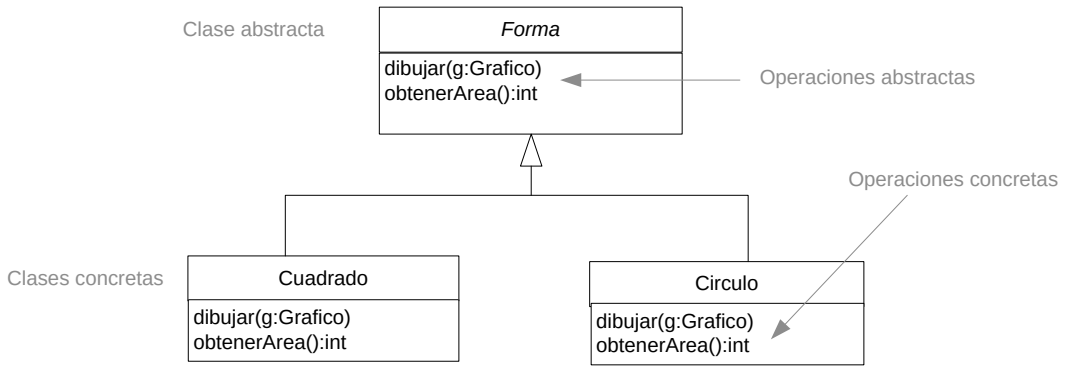


Figura 5. Clases abstracta y clases concretas

3. Polimorfismo

El termino polimorfismo deriva del griego (*poly=”muchos”; morphos=”forma”*).Polimorfismo significa “varias formas”. En la POO una **operación polimórfica es aquella que tiene varias implementaciones**.

Hemos visto en el ejemplo anterior dos operaciones polimórficas. Las operaciones abstractas *dibujar(…)* y *obtenerArea(…)* de la clase *Forma* tienen dos implementaciones diferentes, una implementación en la subclase *Cuadrado* y otra en la subclase *Círculo*.

La **Figura 6** ilustra perfectamente el polimorfismo

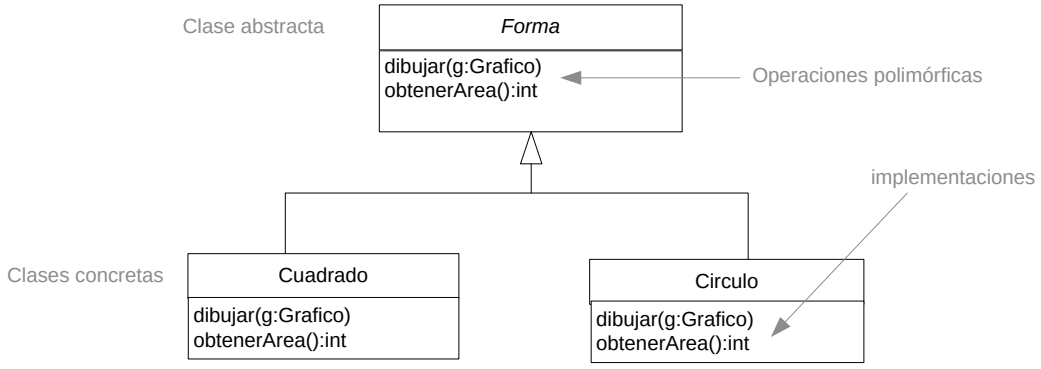


Figura 6. Operaciones polimórficas

La subclase *Círculo y Cuadrado* heredan de la superclase *Forma* y proporcionan una implementación para las operaciones polimórficas *dibujar(…) y obtenarArea()*. Todas **las subclases** de *Forma* **deben proporcionar una implementación concreta** para las operaciones *dibujar(…) y obtenarArea(),* esto significa que desde el punto de vista de **estas operaciones se puede tratar todas las subclases de *Forma* de la misma manera**. Un conjunto de operaciones abstractas es por consiguiente una forma de definir un conjunto de operaciones que todas las subclases deben implementar. Esto se conoce como contrato.

Claramente la implementación de *dibujar() y obtenerArea()* diferirán para la subclase *Cuadrado* y la subclase *Círculo*. Así por ejemplo el método *obtenerArea()* de Circulo devolverá el resultado de la operación π*r*r mientras que la subclase *Cuadrado* devolverá el resultado de la operación alto*ancho. Esta es la **esencia del polimorfismo**, objetos **de diferentes clases tienen operaciones con el mismo prototipo (***igual signatura***) pero diferentes implementaciones.**

Encapsulación, herencia y polimorfismo son los **"tres pilares" de la orientación a objetos.** El polimorfismo nos permite diseñar sistemas más simples, que se adaptan mejor a los cambios por que permite tratar objetos diferentes de la misma forma**.**

De hecho lo que hace al **polimorfismo un aspecto esencial** de la orientación a objetos es que **permite enviar el mismo mensaje a objetos de diferentes clases.**

3.1.Ejemplo de polimorfismo

Vamos a ver un ejemplo del uso del polimorfismo. Suponemos que tenemos una clase *VentanaGráfica* que mantiene y utiliza una colección de *Formas*. El modelo de clases es el mostrado en la **Figura 7.**

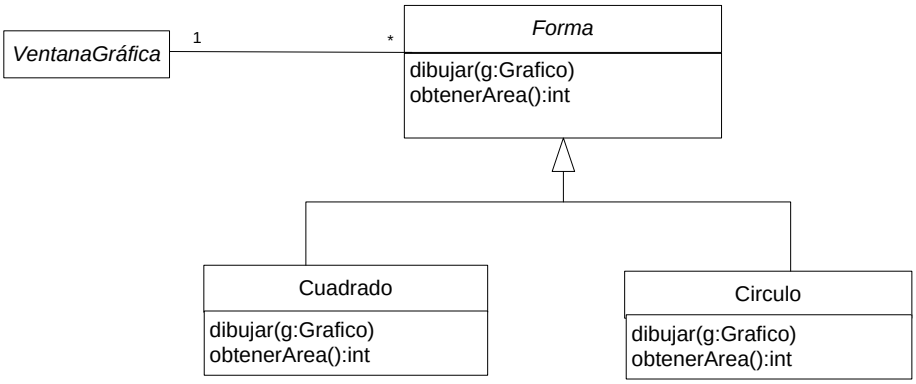


Figura 7 Uso del polimorfismo

Hemos comentado que una clase abstracta no puede ser instancia, luego **no podemos instanciar la clase *Forma***, pero de acuerdo con **el principio de substitución**, **se pueden crear instancias de las subclases concretas de *Forma* en cualquier lugar donde aparezca la clase base.**

Así, aunque se puede observar que un objeto de la clase *VentanaGráfica* puede contener una colección de varios objetos *Forma*, los únicos objetos que realmente se pueden usar son instancias de las subclases, por qué *Forma* es abstracta, y no puede ser instanciada. En este caso particular hay dos subclases concretas, la subclase *Circulo y Cuadrado*. Así, Ventana gráfica contendrá colecciones de objeto de tipo *Círculo y Cuadrado*.

En la **Figura 8** se muestra un modelo de objetos instancia del diagrama de la **Figura 7**. Este modelo muestra como un objeto *VentanaGráfica* soporta una colección de cuatro objetos s1,s2,s3 y s4 donde s1, s2 y s4 son objetos de la clase Circulo y s3 es un objeto de la clase *Cuadrado*. La cuestión es ¿Qué ocurre cuando *VentanaGráfica* interactúa con la colección de formas, y envía a cada objeto de la colección el mensaje *dibujar(…)*. La respuesta es que cada objeto se comportará de la forma esperada: los Cuadrados se

dibujarán como cuadrados y los Círculos como círculos. Es la clase del objeto la que determina lo que el objeto debe dibujar.

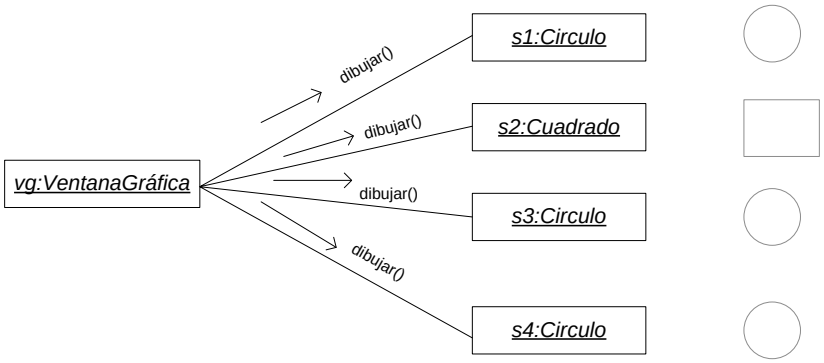


Figura 8 .Uso de operaciones polimórfica.

El punto clave en todo esto es que cada objeto responde a un mensaje por medio de la invocación de la operación correspondiente especificada por su clase.

La **construcción del lenguaje** que **hace posible el polimorfismo** es la **ligadura dinámica** (visto más adelante) conocida también como ligadura tardía o postergada entre llamadas a funciones y los cuerpos reales de dichas funciones.

4. Tipos de herencia

En las relaciones de herencia, se podría dar el caso de que una clase podría heredar las propiedades de más de una clase. Esto da lugar a que tengamos dos tipos de herencia:

- La **herencia simple** .Es aquella en la que cada clase hereda de una única clase.
- La **herencia múltiple.** Es la transmisión de métodos y datos de más de una clase base a la clase derivada. (**Figura 9**).

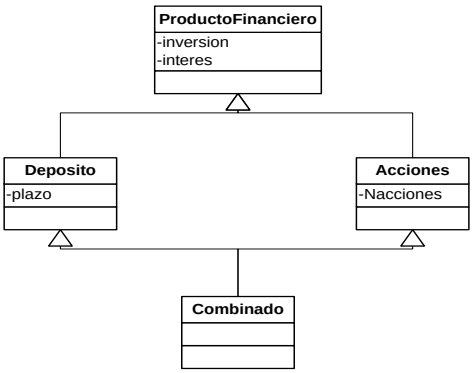


Figura 9.Herencia múltiple

Se pueden presentar dos problemas cuando se diseñan clases con herencia múltiple:

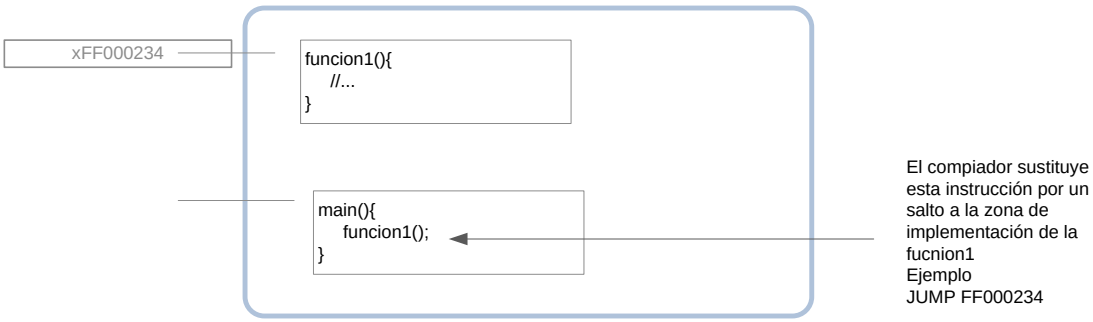
- Colisiones de nombres de diferentes clases base.
- Herencia repetida de una misma clase base.

Las jerarquías de herencia múltiple pueden ser complejas de gestionar. De hecho, no todos los lenguajes OO la implementan. Por ejemplo en Java, C# y SmallTalk no existe la herencia múltiple Aunque en Java se puede simular o implementar a través del concepto de Interfaz. Sin embargo en C++ si se admiten herencia múltiple.

En el ejemplo anterior existe el problema de la repetición de elementos heredados. La clase *Combinado* hereda desde dos clases diferentes los mismos atributos inversión e interés, produciéndose una colisión.

5. Ligadura

La ligadura representa, generalmente una **conexión entre una entidad y sus propiedades**. Si la propiedad se limita a funciones, la ligadura es la **conexión entre la llamada a una función y el código que se ejecuta tras la llamada**.



El **momento de tiempo en el que un atributo o función se asocia con sus valores** o funciones se denomina **tiempo de ligadura**. La ligadura se clasifica según sea el tiempo o momento de la ligadura, y esta puede ser: **estática o dinámica.**

La **ligadura estática** se produce antes de la ejecución (durante la compilación), mientras que la **ligadura dinámica** se produce durante la ejecución.

En un lenguaje de programación con **ligadura estática**, todas las referencias se determinan en tiempo de compilación. La mayoría de los lenguajes procedimentales son de ligadura estática; el compilador y el enlazador (*linker*) definen directamente la posición fija del código que se va a ejecutar en cada llamada a la función.

La **ligadura dinámica** supone que **el código a ejecutar en respuesta a un mensaje no se determinará hasta el momento de la ejecución**. Únicamente la ejecución del programa (normalmente un puntero a la clase base) determinará la ligadura efectiva entre las diversas que son posibles (una para cada clase derivada).

Vamos a ver y a estudiar estos conceptos a través de un ejemplo. Para ello vamos a considerar que la notación ***virtual*** sobre un método de una clase indica al compilador que esa función que será implementada en una subclase no en la clase donde se define.

Vamos a suponer, que sólo se puede usar esta opción cuando exista la posibilidad de que una subclase implemente los métodos denominados *virtuales.*

Vamos a imaginarnos que un instante de tiempo dado en la memoria hay tres objetos instancia de tres clases diferentes:

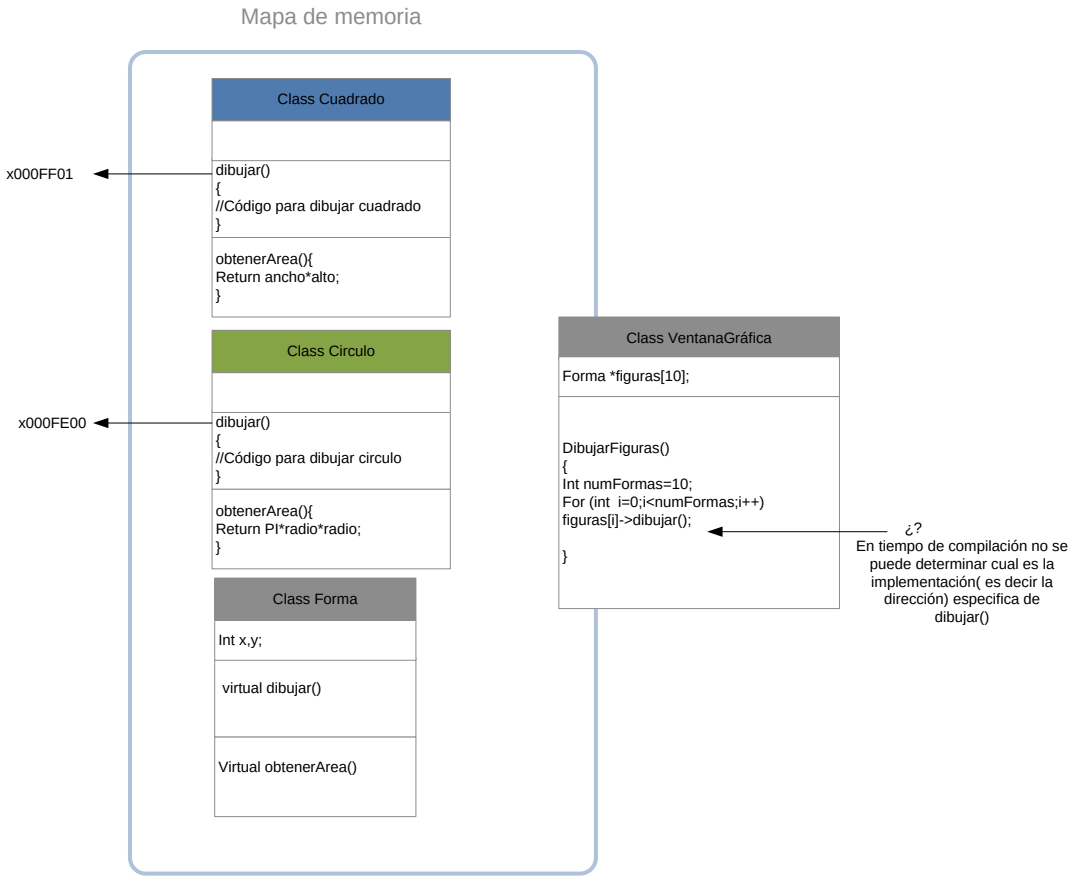


Figura 10 Mapa de memoria, funciones virtuales y polimorfismo

La cuestión es ¿Como resuelve el compilador la siguiente instrucción?

```
figuras[i]->dibujar();
```

Donde `figuras[i]` es una array de Figuras.

El compilador tiene que enlazar la llamada a la función *dibujar()* con alguna zona de memoria , pero en este caso, como vemos en la **Figura 10,** no tiene ninguna implementación. Pero como está indicado con la palabra reservada virtual. El compilador "sabe" que debe buscar esta implementación en alguna clase derivada. ¿Cuál de ellas? La que en ese momento este en la posición indexada

5.1. Ventajas de la ligadura dinámica

La principal ventaja de la ligadura dinámica frente a la ligadura estática es que la dinámica ofrece un alto grado de flexibilidad y diversas ventajas prácticas para manejar

jerarquías de clases de un modo muy simpe. Entre las desventajas de la ligadura dinámica se encuentra que en principio es menos eficiente que la ligadura estática.

Los lenguajes OO que si siguen estrictamente el paradigma ofrecen sólo ligadura dinámica. Los lenguajes híbridos (C++,Simula) ofrecen los dos tipos de ligadura.

5.2. Polimorfismo con ligadura dinámica

Con la ligadura dinámica, el tipo de objeto no es preciso decidirlo hasta el momento de la ejecución. El ejemplo anterior en la sección de la clase **VentanaGrafica :**

```
DibujarFiguras()
{
Int numFormas=10;
For (int  i=0;i<numFormas;i++)
figuras[i]->dibujar();

}
```

La instrucción *figuras[i]->dibujar();* envía al programa al bloque apropiado de código basado en el tipo de objeto . En concreto pasará el mensaje *dibujar()* a la figura apuntada por *figuras[i]*. La indicación virtual en el método *dibujar()* de la clase *Figura* ha indicado al compilador que esta función se puede llamar por medio de un puntero. Mediante la ligadura dinámica, el programa determina **el tipo de objeto en tiempo de ejecución.**