Polimorfismo, clases base abstractas e interfaces en C++ 11

Prfr: M. en I. Fco. Javier Rodríguez García
Depto. de Ingeniería en Computación, F. I., UNAM
México, 2017-1

Resumen métodos virtuales

La herencia, los métodos virtuales y la anulación o redefinición de tales métodos obliga al compilador a posponer la decisión sobre la versión del método a usar hasta que el programa ya se está ejecutando. Su contraparte, la herencia sin métodos virtuales, decide en el preciso momento de la compilación la versión de los métodos a utilizar. En inglés a esto se le llama *hard-wired* (alambrado).

Polimorfismo (programando para el futuro)

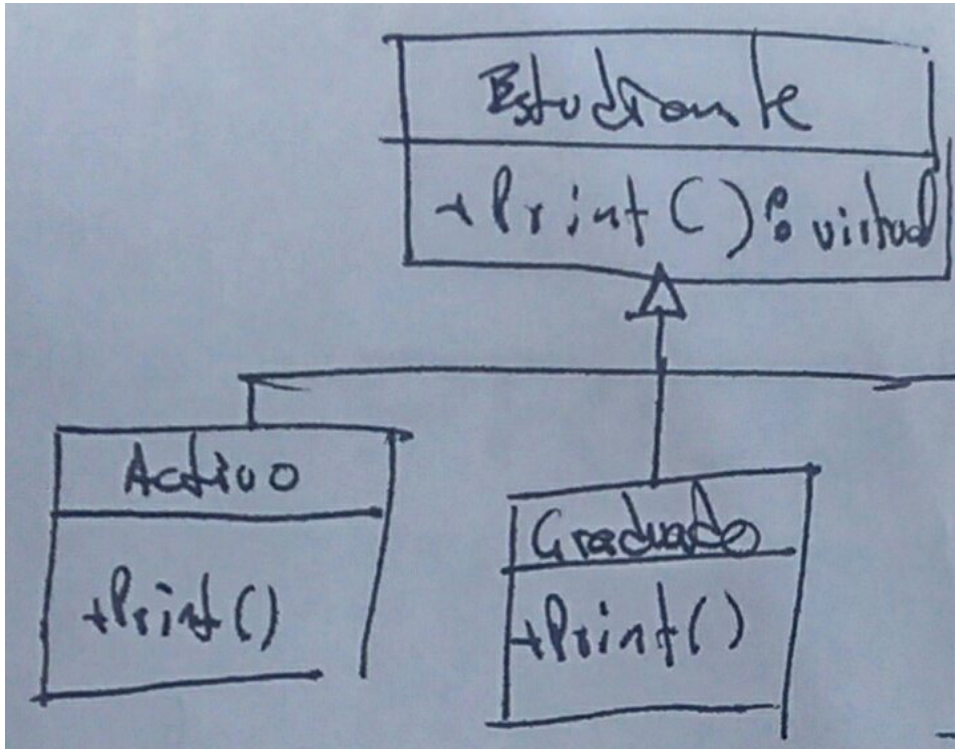
El polimorfismo es la habilidad de un objeto para tomar diferentes formas; esto es, un objeto polimórfico responderá al mismo mensaje pero de manera acorde al objeto que se supone recibirá dicho mensaje.

Para que el polimorfismo se pueda llevar a cabo se necesitan cuatro cosas:

1)Herencia
2)Métodos virtuales
3)Anulación o redefinición de métodos virtuales
4)Un apuntador o referencia a la clase base

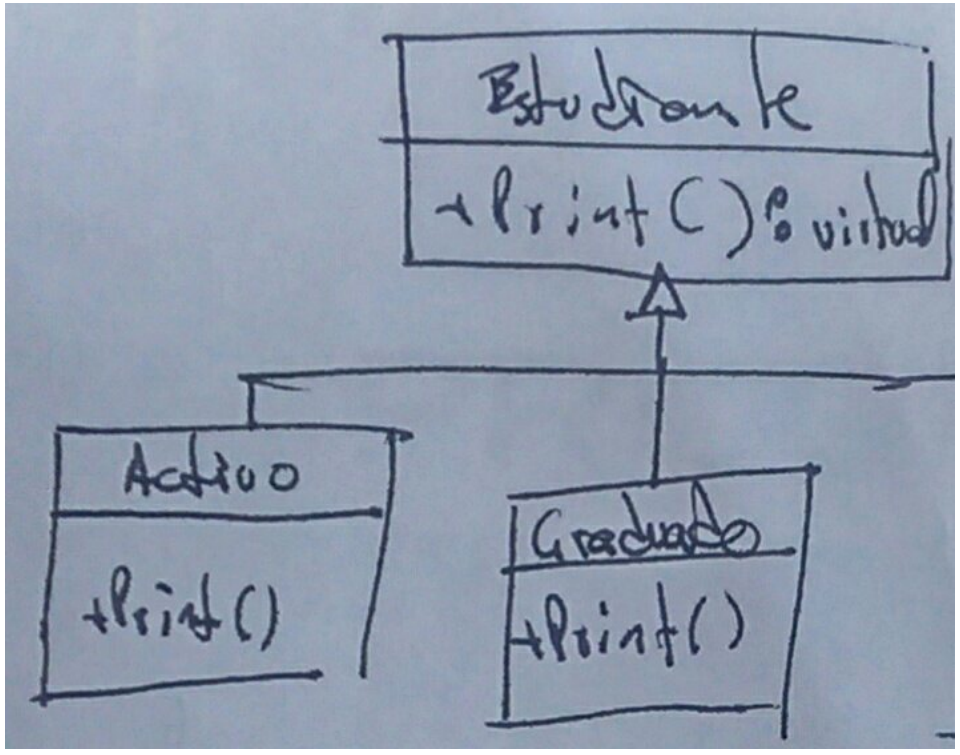
El polimorfismo es útil en muchos aspectos de la programación:

- Como substituto de sentencias tipo switch
- Para iterar sobre una colección de objetos relacionados por la herencia
- Como base en muchos de los patrones de diseño de software



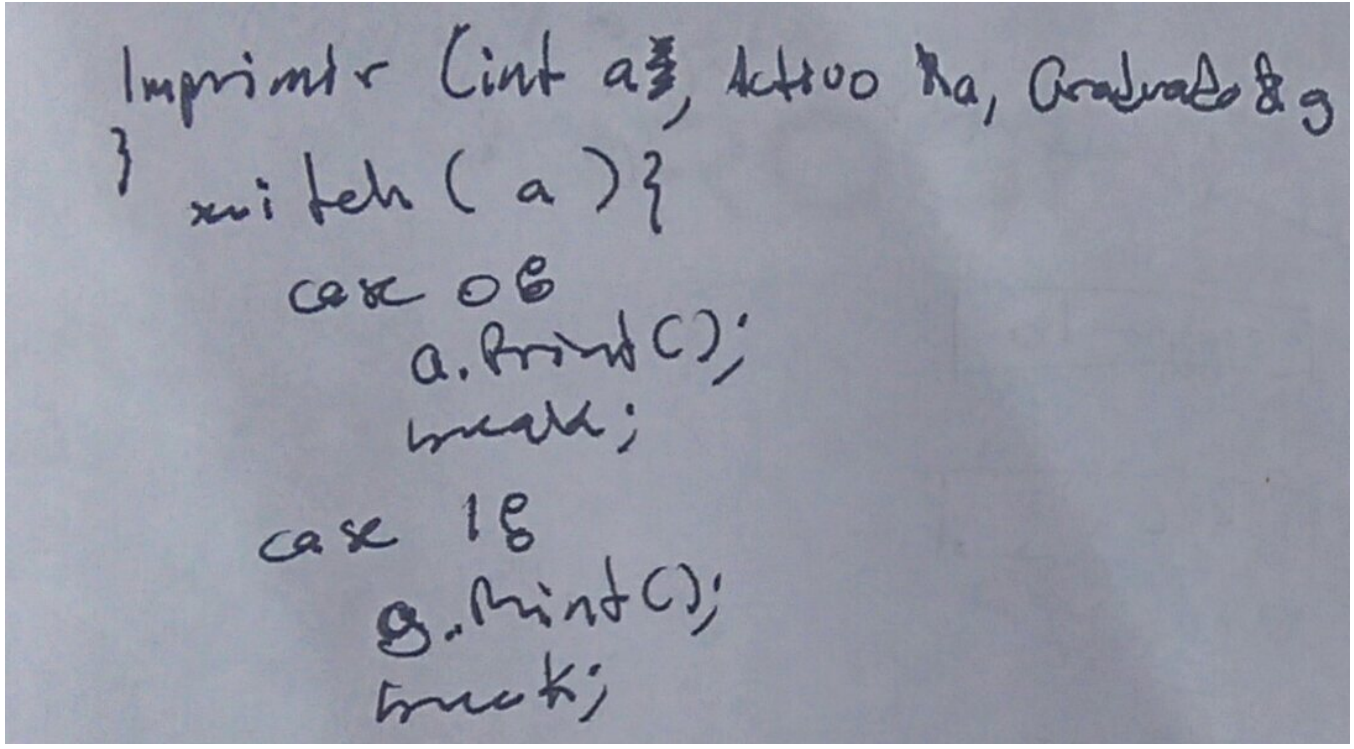
Un estudiante activo responde al mensaje Print() imprimiendo su nombre, semestre y promedio.

Un estudiante graduado responde al mensaje Print() imprimiendo su nombre, promedio final y número de cédula profesional.

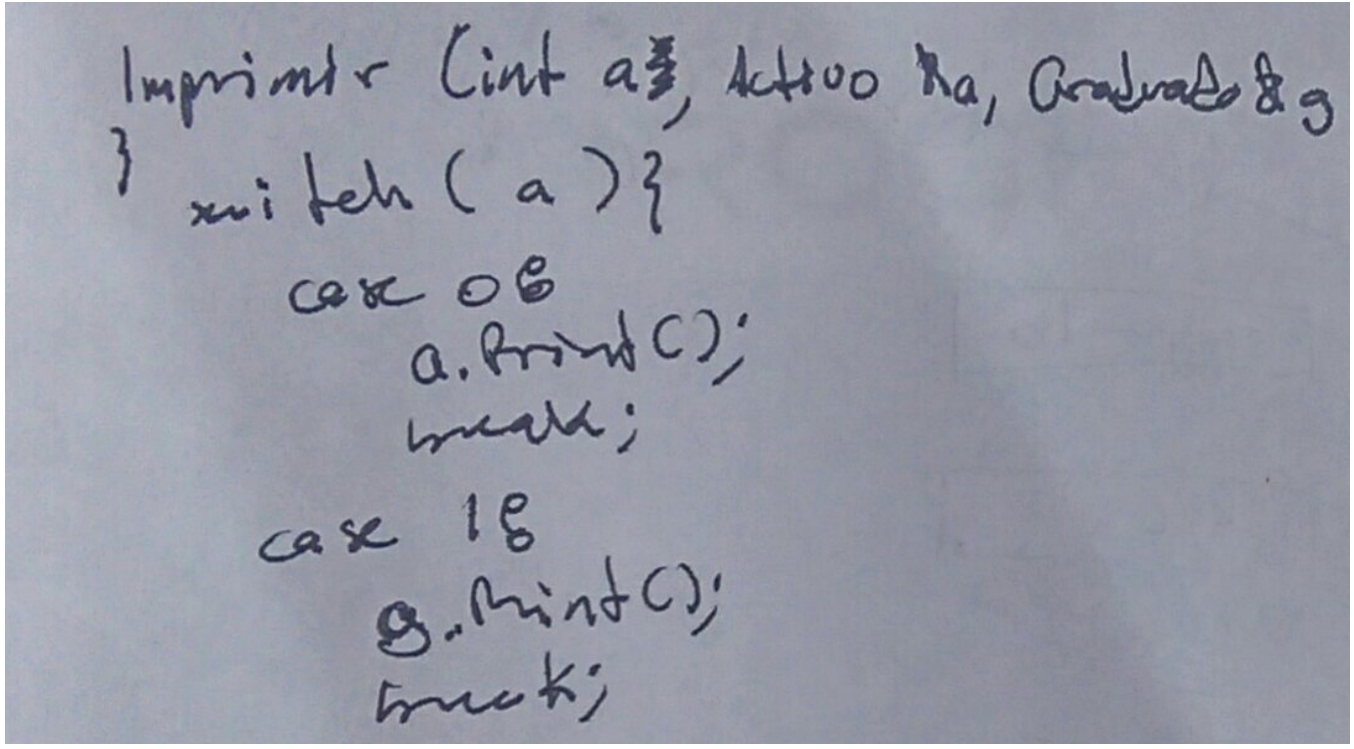


En diversos puntos del programa tendremos la necesidad de enviar el mensaje `Print()` a cualquiera de los dos tipos de estudiantes.

¿Cómo lo harían?

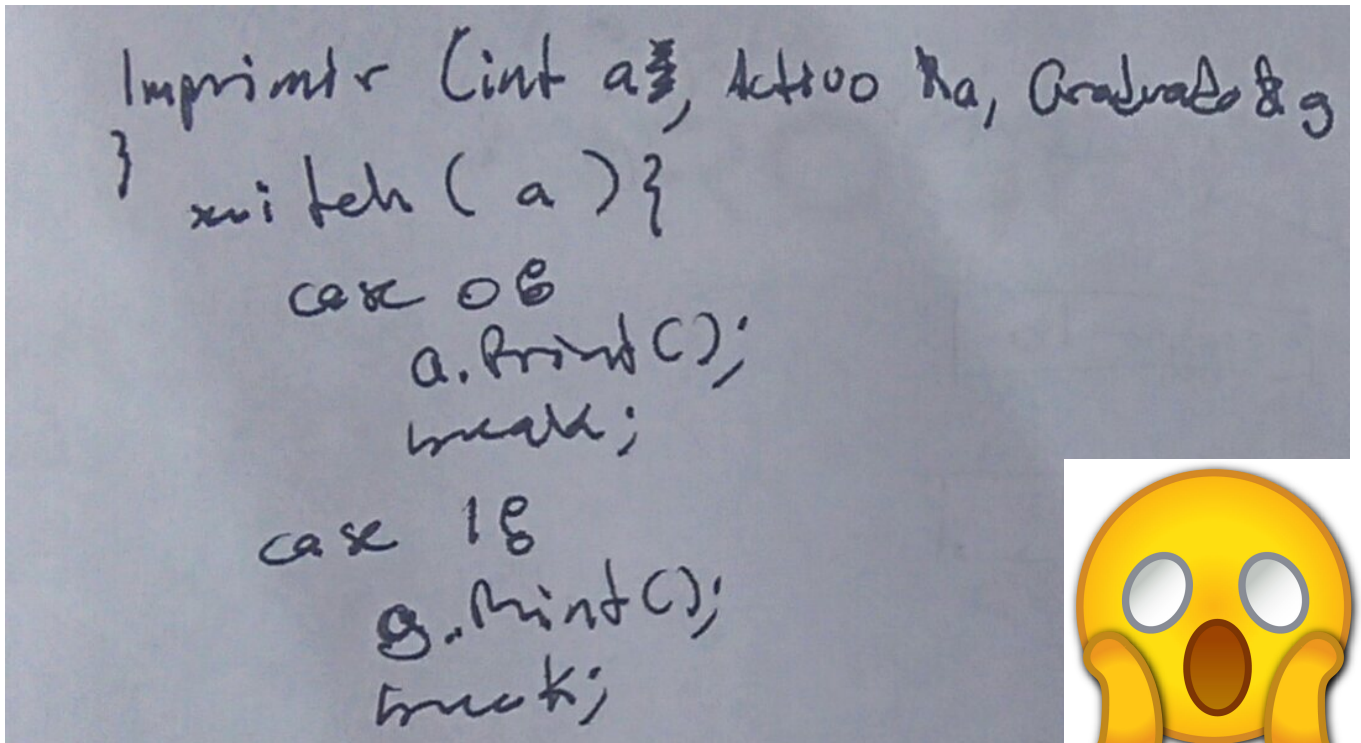


En este ejemplo la función `Imprimir()` requiere imprimir la información de cada objeto de los dos tipos de estudiantes que existen actualmente.



En este ejemplo la función `Imprimir()` requiere imprimir la información de cada objeto de los dos tipos de estudiantes que existen actualmente.

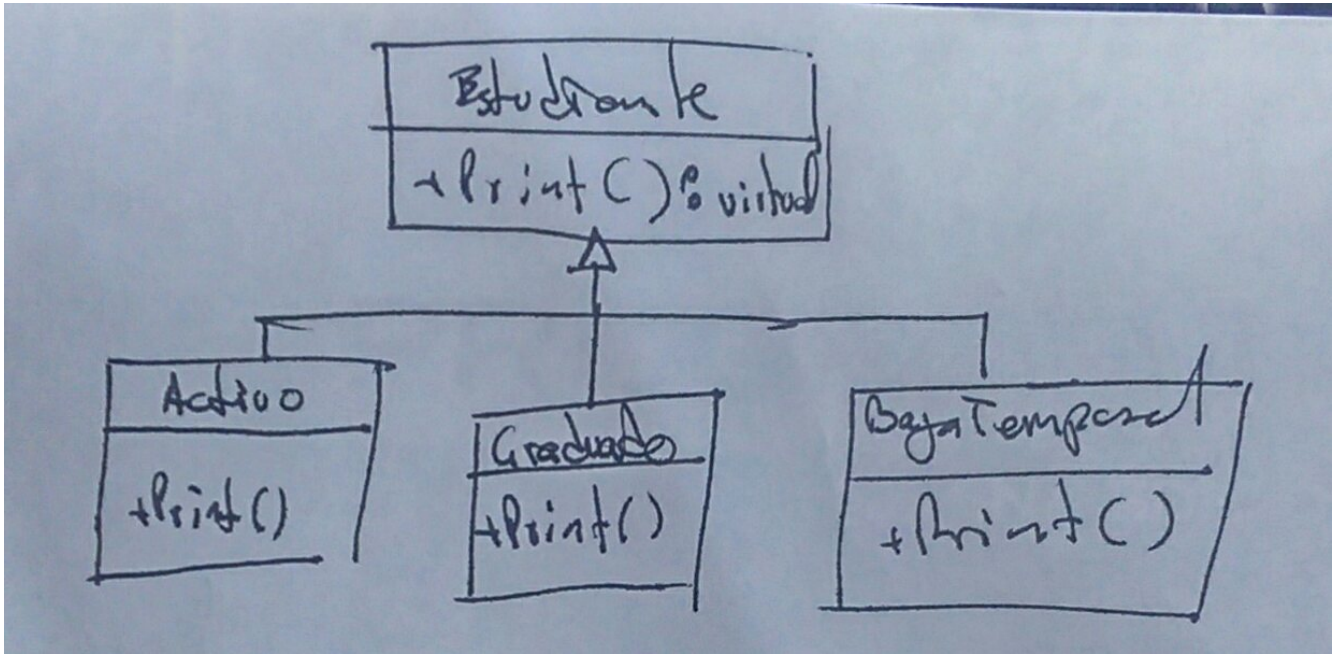
¿Y si fueran 20 funciones diferentes donde tuviéramos que hacer algo similar?



En este ejemplo la función `Imprimir()` requiere imprimir la información de cada objeto de los dos tipos de estudiantes que existen actualmente.

¿Y si fueran 20 funciones diferentes donde tuviéramos que hacer algo similar?

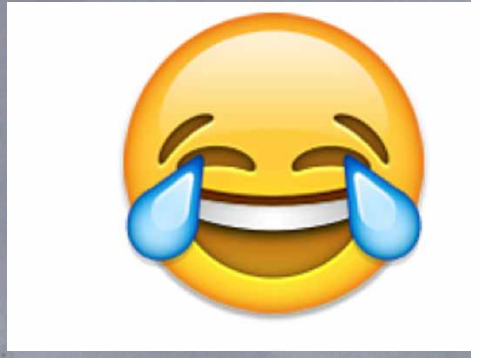
¿Y si agregáramos un tercer tipo de estudiante?



Aquí hemos agregado a un tercer tipo de estudiante, `BajaTemporal`

Y nos queda así el código con la solución tradicional

```
Imprimir (int a, Activo &a, Graduado &g, Egresado &e)
{
  switch (a){
    case 0:
      a.Print();
      break;
    case 1:
      g.Print();
      break;
    case 2:
      e.Print();
      break;
  }
}
```



El polimorfismo resuelve tal problema eliminando todos los `switch`s, excepto uno, y las funciones se simplifican:

```
int main()
{
    Activo a;
    Graduado g;

    int a; cin << a;
    switch (a) {
        case 0: Imprimir (a); break;
        case 1: Imprimir (g); break;
    }
}

void Imprimir (Estudiante & r)
{
    r.Print ();
}
```

El polimorfismo resuelve tal problema eliminando todos los `switch`s, excepto uno, y las funciones se simplifican:

```
void Imprimir (Estudiante & r)
{
    r.Print ();
}
```

Se puede pensar en el objeto como un interruptor multiposición, que una vez que se selecciona una posición en particular, ahí se mantiene hasta que la perilla se vuelve a mover.

Métodos virtuales puros y clases base abstractas

Los métodos virtuales puros son métodos virtuales declarados en la clase base y no tienen cuerpo, es decir, no tienen código.

Las clases derivadas son responsables de escribir el código correspondiente. En caso de no hacerlo el compilador reportará un error.

Un método virtual puro se declara de la siguiente manera:

```
virtual tipo nombre (args) const = 0;
```

Una clase que tenga al menos un método virtual puro se convierte en una clase base abstracta.

Una clase base abstracta puede tener métodos virtuales puros y no puros.

NO se pueden instanciar objetos de una clase base abstracta, porque éstas son clases incompletas!

Pero lo que sí podemos (y debemos) tener son apuntadores o referencias a dicha clase base abstracta.

El poder de la abstracción [KEOGH04]

Abstracción es la técnica utilizada por los programadores de una clase base para obligar a los programadores de las clases derivadas a definir la *funcionalidad* de un *comportamiento*.

– **Comportamiento**: son los servicios o responsabilidades que la clase provee (*¿qué hace?*)
– **Funcionalidad**: son las instrucciones que implementan a un comportamiento (*¿cómo lo hace?*)

Los métodos virtuales puros, al no implementar ninguna *funcionalidad*, evitan que los programadores “olviden” escribir su versión del *comportamiento*.

Vocabulario POO

A las clases base abstractas de C++ se les llama ***clases abstractas***.

A las clases derivadas de una clase base abstracta se le llama ***clases concretas***.

A los métodos virtuales puros de C++ se les llama ***métodos abstractos***.

A los métodos de las clases derivadas que *implementan* los métodos virtuales puros de la clase base se les llama ***métodos concretos***.

Usos de la abstracción

Algunos usos de la abstracción son:

– Frameworks: Un framework es una base de software (muy especializada) a partir de la cual se pueden crear aplicaciones. Algunos les llaman “semi-aplicaciones”.

– Bibliotecas: Son un conjunto de clases especializadas escritas expresamente para ser utilizadas por terceros.

Abstraction is sometimes a game of guessing the future. [KEOGH04]

Interfaces

Una interfaz describe el comportamiento de un componente, pero no especifica su implementación. Lo podemos considerar como un contrato.

La interfaz de un componente es el uso estándar que se le puede dar, o que esperamos de él. Por ejemplo, sin importar el fabricante de un control remoto para una televisión, lo menos que podemos esperar de este dispositivo es:

Un botón para encender/apagar el dispositivo
Botones para subir/bajar el volumen
Botones para cambiar los canales

Interfaces

El fabricante podrá usar la tecnología y diseño exterior que desee, pero la funcionalidad de un control remoto siempre es la misma.

Los pedales de un coche
Los mandos en una motocicleta
Los colores de un semáforo

Este concepto de *funcionalidad estándar* se lleva a los componentes de software a través de las interfaces.

Interfaces

Como clientes de una base de datos esperamos, al menos, poder abrirla, cerrarla y hacer queries (una query puede leer y escribir a la base de datos).

Como clientes de un administrador de dispositivos (*device driver)* esperamos, al menos, poder abrirlo, cerrarlo, leer de y escribir hacia.

De un control de una GUI esperamos poder crearlo, dibujarlo, actualizarlo, y recibir notificaciones cuando el mouse haga un click sobre él.

Interfaces en C++

Una interfaz se implementa en C++ como una clase base abstracta, con todos sus métodos públicos y virtuales puros, sin constructores, con el destructor virtual, y sin datos miembro. No existe ninguna palabra clave que indique que la clase será una interfaz (Java y C# sí la tienen).

Las clases que deriven de una interfaz deberán implementar su versión de los métodos virtuales puros, los constructores necesarios, y agregar los datos miembro requeridos, así como otros métodos públicos y privados.

Interfaces en C++ (II)

En ocasiones necesitamos que un componente herede la funcionalidad de dos clases. C++ permite la herencia múltiple; esto es una clase derivada de dos o más clases base.

Sin embargo, es más práctico y limpio que el componente resultante herede de dos interfaces:

```
class x : public IClase1, protected IClase2 {...}
```

Es una práctica comun usar una í mayúscula como prefijo de una clase que es interfaz.