Patrón Delegate y Herencia Múltiple

El patrón Delegate.
El patrón Delegation describe como un objeto, en lugar de realizar una tarea por sí mismo, delega la tarea a otro objeto que es capaz de realizarla de manera más adecuada. En este patrón, un objeto delegado es responsable de llevar a cabo la tarea delegada, mientras que el objeto delegador se encarga de controlar el proceso y coordinar el flujo de datos.

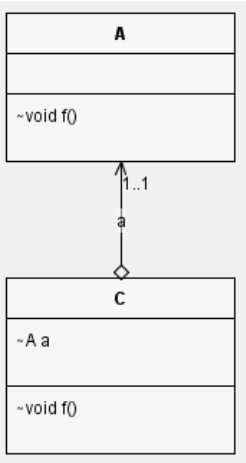
Cual es objetivo de Delegate
Este patrón es útil cuando se necesita una flexibilidad y modularidad adicionales en el diseño del software. Al utilizar la delegación, se puede cambiar la funcionalidad de un objeto delegador sin tener que cambiar su implementación. Además, permite a los objetos delegadores reutilizar la funcionalidad proporcionada por los objetos delegados, lo que puede ahorrar tiempo y reducir la complejidad del código.

Delegate y composición.
La delegación puede ser implementada de varias maneras, pero una de las más comunes es mediante la composición de objetos. En este enfoque, el objeto delegador contiene una referencia al objeto delegado y, cuando se llama a un método en el objeto delegador, se delega la ejecución del método al objeto delegado. El objeto delegado es el responsable de realizar la tarea y devolver el resultado al objeto delegador.

¿Se puede conseguir delegación sin composición?.
Sí, y hay una forma que estás harto de usar: una subclase al utilizar el código de su superclase, delega en esta superclase. En este caso el subobjeto padre es el objeto delegado. Y se puede considerar delegación algún otro mecanismo de programación pero cuando hablamos de delegación sin especificar nada más se sobreentiende que nos referimos a delegación con composición.

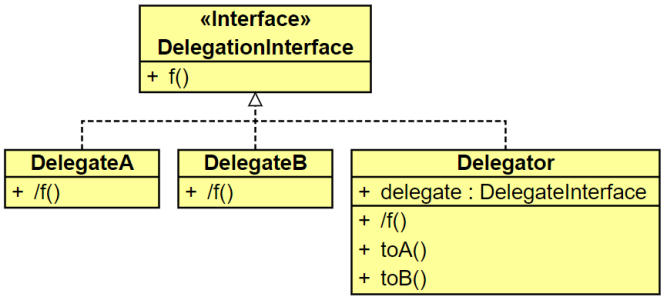
Relación entre el patrón Delegate y herencia múltiple
Una forma de conseguir el efecto de Herencia múltiple es utilizando el patrón Delegation basado en composición.

Versión sencilla del Patrón delegate.
Se establece una relación fija entre un objeto C y A
Es la que usamos cuando estudiamos "métodos delegados" en el boletín de composición. El Patrón lo podríamos describir con el siguiente UML.



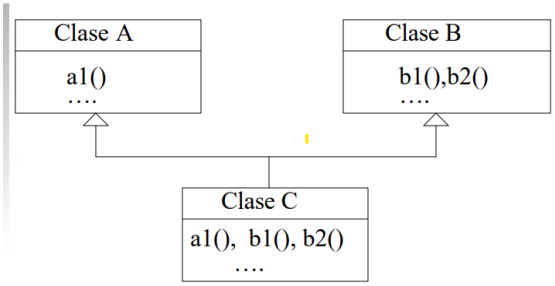
El patrón indica que una clase C :
1. Utiliza composición con una clase A.
2. Define métodos delegados

Versión extensible del patrón Delegate
El patrón Delegater con esta estructura puede elegir en qué clase quiere delegar utilizando los métodos toA() y toB().
https://es.wikipedia.org/wiki/Delegation_(patr%C3%B3n_de_dise%C3%B1o)



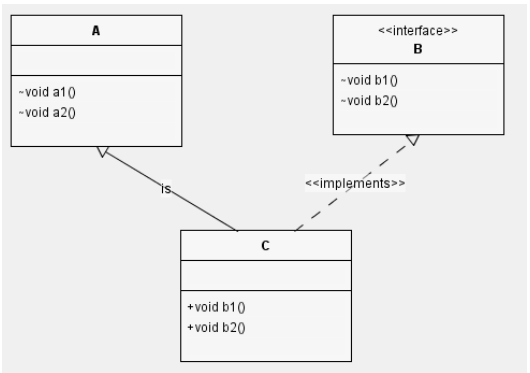
Hablaremos de este esquema en otro boletín.

Patrón Delegation para permitir simular herencia múltiple.
¡Ya lo utilizamos intuitivamente al ver herencia vs Composición!
Vamos a aplicar dos conceptos que ya usamos en anteriores ejemplos, sustituir extends por composición(recuerda los ejemplos de herencia vs composición) y generar métodos delegados(recuerda los ejemplos que vimos en composición y delegación).
Supongamos que deseo que una clase C herede de A y B como se describen en el siguiente diagrama.



Ya sabemos que no es posible en Java reutilizar más de una clase vía Herencia "extends".

La solución que vimos en boletín pasado: extends + implements
Sabemos también que si la clase B se puede definir en realidad como un conjunto de métodos abstractos, basta con crear un interface e implementarlo:

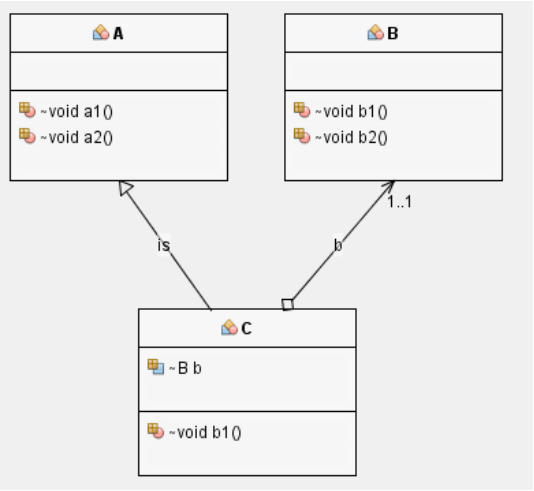


Observa que al implementar un interface hay que dar implementación a todos sus métodos, por ejemplo aunque C no tenga interés en b2() tiene que sobrescribirlo obligatoriamente.

```
class A {
     void a1() { System.out.println("soy a1()");}
     void a2() { System.out.println("soy a2()");}
 }
interface B{
    void b1();
    void b2();
}
class C extends A implements B{
    @Override
    public void b1() {System.out.println("soy b1()");}
    //el método b2() no me interesa pero tengo que sobrescribirlo
    @Override
    public void b2() {System.out.println("soy b2()");}

}
```

Otro enfoque: introducir la composición en la solución

Si la clase B tiene estado(atributos), reutilizamos la clase B por composición. Esto además tiene la ventaja de que aquellos métodos de B que no le interesan a C simplemente se ignoran y no se les hace método delegado. En cambio los métodos de los interfaces hay que sobreescribirlos obligatoriamente.
En el ejemplo b2() suponemos que no nos interesa




```
class A {
     void a1() { System.out.println("soy a1()");}
     void a2() { System.out.println("soy a2()");}
 }
class  B{
     void b1() {System.out.println("soy b1()");}
     void b2() {System.out.println("soy b2()");}
}
class C extends A {
   B b= new B();
   void b1(){b.b1();}

}
```

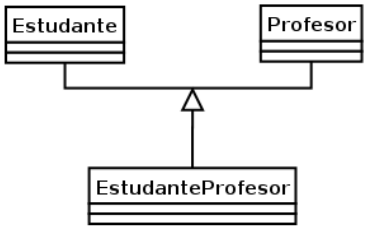
¿Dónde se debe crear el objeto delegado?

La forma en que se crea el objeto delegado puede variar según la situación. En el ejemplo de arriba lo crea el propio objeto delegador C, pero podría crearse fuera y llegarle al objeto C por ejemplo como parámetro del constructor como en el siguiente

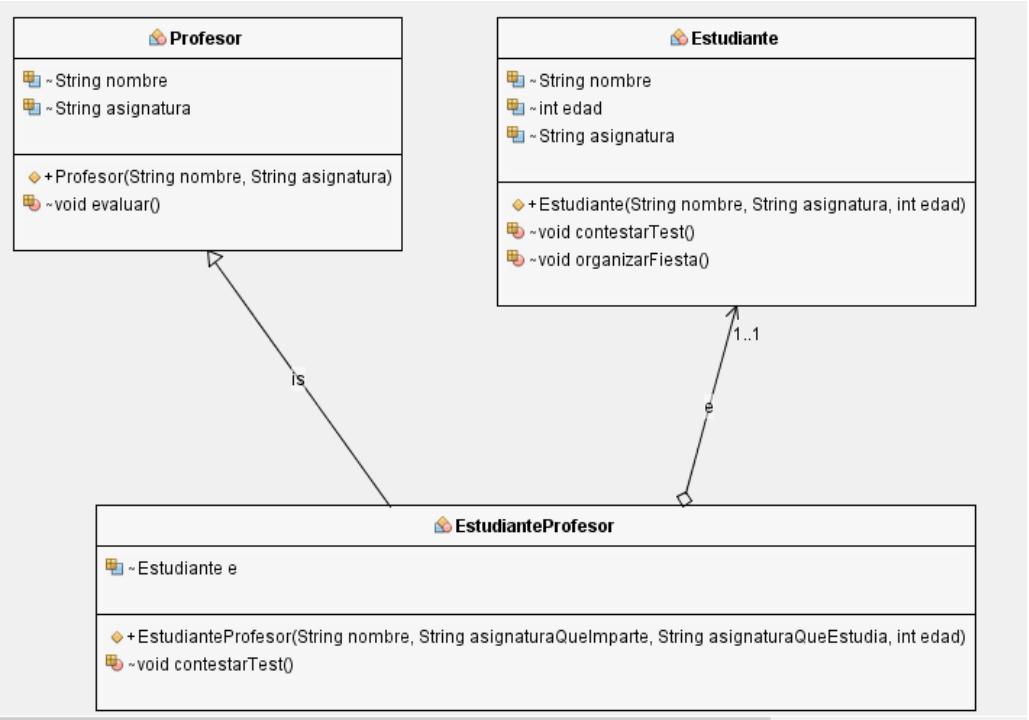
ejemplo en el que el objeto delegado es Rectangle y se le suministra al objeto Delegador como parámetro de su constructor.

```
class Rectangle {
    private int width;
    private int height;

    public Rectangle(int width, int height) {
        this.width = width;
        this.height = height;
    }

    public int area() {
        return width * height;
    }
}

class Window {
    private Rectangle bounds;

    public Window(Rectangle bounds) {
        this.bounds = bounds;
    }

    public int area() {
        return bounds.area();
    }
}
```

EJEMPLO: vamos a conseguir que un objeto EstudianteProfesor reutilice las clases Profesor(por herencia "extends") y Estudiante(por composición).



En el ejemplo, además vamos a tratar la colisión de nombre y asignatura. El nombre tendrá que ser el mismo para ambos y utilizamos por ejemplo el heredado de profesor. El atributo asignatura también colisiona pero son asignaturas diferentes ya que como estudiante está matriculado en una asignatura y como profesor imparte otra diferente. La colisión se resuelve ya que una es *super.asignatura* y otra *e.asignatura*. Por último, con el mecanismo de métodos delegados reutilizamos el código de contestarTest() del estudiante




```
class Estudiante{
    String nombre;
    int edad;
    String asignatura;//para simplificar 1 estudiante estudia sólo 1 asignatura

    public Estudiante(String nombre, String asignatura,int edad) {
        this.nombre = nombre;
        this.asignatura = asignatura;
        this.edad=edad;
    }

    void contestarTest(){
        System.out.println("contestando test de "+asignatura);
    }
    void organizarFiesta(){
        System.out.println("no todo es estudiar");
    }

}
class Profesor{
    String nombre;
    String asignatura;//para simplificar 1 profesor imparte sólo 1 asignatura

    public Profesor(String nombre, String asignatura) {
        this.nombre = nombre;
        this.asignatura = asignatura;
    }

    void evaluar(){
        System.out.println("evaluando ..."+asignatura);
    };
}
class EstudianteProfesor extends Profesor{
    Estudiante e;
    public EstudianteProfesor(String nombre, String asignaturaQueImparte,String asignaturaQueEstudia,int edad) {
        super(nombre, asignaturaQueImparte);
        e= new Estudiante(nombre,asignaturaQueEstudia,edad);
    }
    void contestarTest(){
        e.contestarTest();
    }

}


public class HerenciaMultiple {
```

```
    public static void main(String[] args) {
        EstudianteProfesor ep=new EstudianteProfesor("chuski","lógica","programación",35);
        ep.contestarTest();
        ep.evaluar();
        System.out.println(ep.nombre);
    }
}
```

EJERCICIO U5_B6B_E1:

```
class Mamifero{
    int tiempoLactancia;

    public Mamifero(int tiempoLactancia) {
        this.tiempoLactancia = tiempoLactancia;
    }
    void mamar(){System.out.println("chup chup");}

}
class AnimalAcuatico{
    int temperaturaAgua;
    public AnimalAcuatico(int temperaturaAgua) {
        this.temperaturaAgua = temperaturaAgua;
    }
    void nadar(){System.out.println("glu glu");}


}
```

Utilizando extends y el patrón delegate escribe una clase Ballena que reutiliza el código de Mamifero y AnimalAcuatico, funcionando el siguiente main()

```
public static void main(String[] args) {
        Ballena b= new Ballena(2,15);
        b.nadar();
        b.mamar();
    }
```