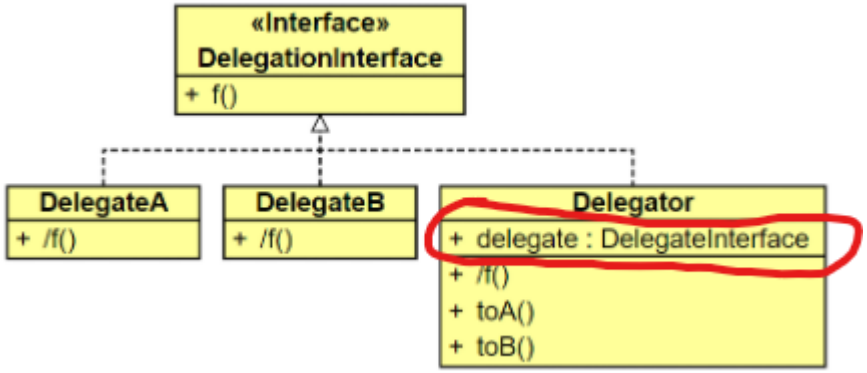
Ya indicamos su UML en un boletín anterior.



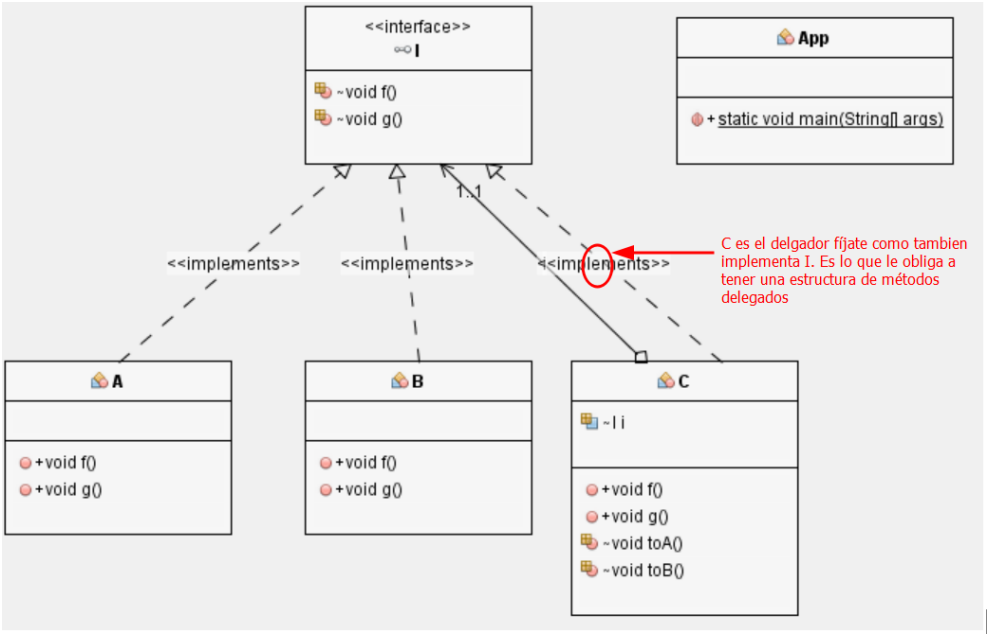
Observa las siguientes característica en el Delgador:

- Típicamente contiene métodos delegados como f()
- El delegador tiene una variable de tipo interface que puede señalar a cualquier tipo de instancia que implemente dicho interface
- En el delegador hay métodos que permiten cambiar de objeto delegado como toA() y toB()

Sobre este esquema en la práctica nos podemos encontrar muchas variantes y matices.

Veamos un ejemplo genérico pero un código para analizar el patrón

En el siguiente ejemplo C es la clase delegador y A y B delegados. Esta estructura permite que el objeto delegator pueda cambiar en tiempo de ejecución de objeto interior y por tanto de implementación de método f(), es decir, en un instante f() puede ejecutar el f() de DelegateA y en otro instante el de DelegateB.



```
interface I {
    void f();
    void g();
}

class A implements I {
    public void f() { System.out.println("A: haciendo f()"); }
    public void g() { System.out.println("A: haciendo g()"); }
}

class B implements I {
    public void f() { System.out.println("B: haciendo f()"); }
    public void g() { System.out.println("B: haciendo g()"); }
}

class C implements I {
    // objeto asociado de delegacion
    I i = new A();

    public void f() { i.f(); }
    public void g() { i.g(); }

    // atributos normales
    void toA() { i = new A(); }
    void toB() { i = new B(); }
}

public class Main {
    public static void main(String[] args) {
        C c = new C();
        c.f();
        c.g();
        c.toB();
        c.f();
        c.g();
    }
}
```

¿Pero este patrón no es el mismo que Strategy?

El patrón Delegate y el patrón Strategy se confunden fácilmente ya que ambos patrones permiten que se pueda cambiar de objeto en tiempo de ejecución pero "ese cambio" se utiliza en diferentes situaciones y con diferentes objetivos en mente. En muchos ejemplos de la web hay un verdadero lío al respecto y mezclan en sus explicaciones ambos patrones de forma muy confusa. Tienes que tener en cuenta que en casi todos los patrones hay interfaces, composición etc. así que el matiz de uso es a veces crucial para distinguir uno de otro. En el caso de delegate son características importantes:

- Es una alternativa a la reutilización que típicamente se consigue con herencia
- Es importante trabajar con métodos delegados
- Hay un objeto que llamamos en el patrón genérico Delegator que también implementa el interface. Aquí ya vemos una diferencia clara con Strategy donde lo que en Strategy llamamos contexto no implementa el interface de las estrategias.
- El interface definirá todo el conjunto de métodos que quiero delegar.

UN EJEMPLO MÁS CONCRETO

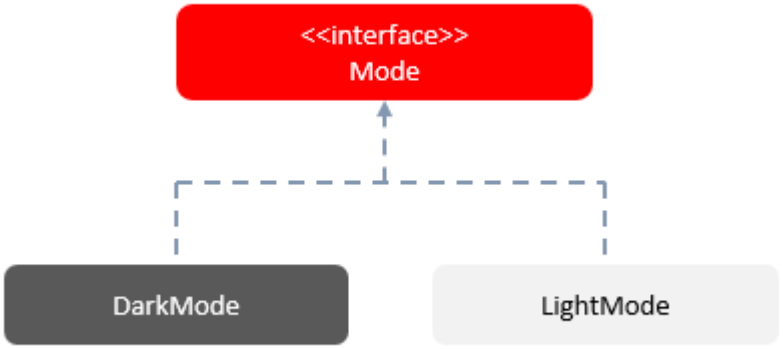
Como se indicó, muchos ejemplos de la web son realmente confusos. Concretamente suelen mezclar confusamente delegate y strategy. Vamos a considerar el siguiente ejemplo que demuestra solvencia y precisión

https://hmkcode.com/kotlin/delegation-design-pattern-in-kotlin/

Está escrito para Kotlin pero lo adaptaremos aquí a Java.

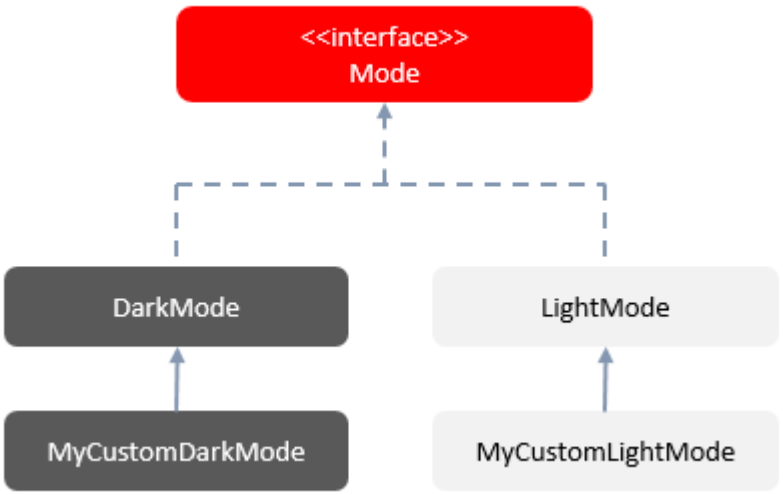
Observa que el objetivo es conseguir reusar código sin usar herencia que es lo definitivo para el patrón delegate, sin esto podemos hablar de composición a secas con algún método delegado por ahí suelto pero no de patrón delegate.

El ejemplo parte de la siguiente situación, en la página puedes ir viendo el código Kotlin paso a paso pero nosotros sólo incorporamos el resultado final en Java para hacer más simple la lectura del boletín.



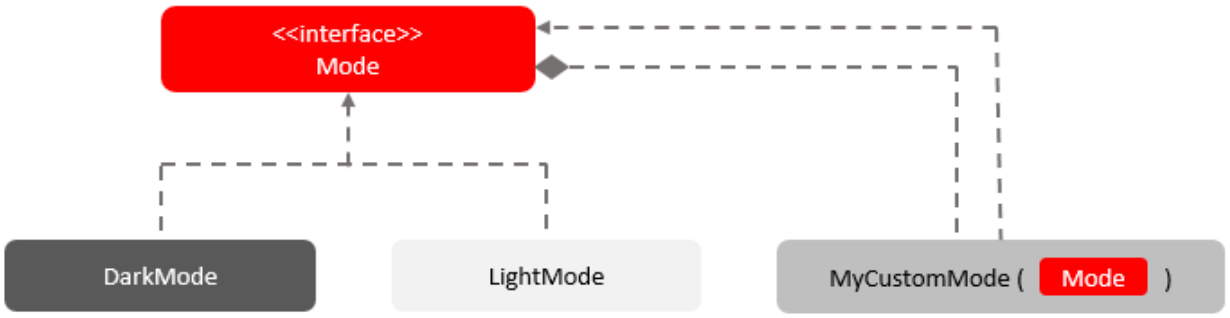
Hay dos modos de pantalla, pero ahora sobre estos modos se desean hacer variantes personalizadas.

Usar Herencia "clásica" para obtener variantes personalizadas.



Usar patrón Delegation para conseguir las personalizaciones

La reutiliación con herencia es intuitiva y se entiende bien, pero como ya se discutió tiene problemas de mantenimiento y curiosamente es más "ágil" conseguir reutilización usando composición con la estructura delegate Pattern que sería la siguiente



Fíjate bien en que MyCustomMode tiene relación de implements y de composición con el interface, esto como ya vimos es definitorio para Delegate extendido.

Recuerda que " El patrón de delegación es un patrón de diseño orientado a objetos que permite que la composición de objetos logre la misma reutilización de código que la herencia".

Ten siempre en mente que queremos reutilizar código y entonces:

- Sí a MyCustomMode le pasamos un objeto DarkMode hacemos una reutilización equivalente a cuando por herencia creamos en el diagrama anterior la clase derivada MyCustomDarkMode
- Si a MyCustomMode el pasamos un objeto LightMode hacemos una reutilización equivalente a cuando por herencia creamos en el diagrama anterior la clase derivada MyCustomLightMode

Fíjate entonces que el objeto Delegador ahora consigue el efecto de los objetos derivados de herencia pero de una forma mucho más flexible aunque ciertamente es más difícil de entender.

En nuestro caso, podemos crear un modos personalizado que reutilicen el *display()* de los modos iniciales.

```
interface Mode {
   String getColor();
   void display();
}

class DarkMode implements Mode {
   private String color;
   public DarkMode(String color) {
      this.color = color;
   }
   public String getColor() {
      return color;
4
```

```
  }
  public void display() {
    System.out.println("Dark Mode..." + color);
  }
}

class LightMode implements Mode {
  private String color;
  public LightMode(String color) {
    this.color = color;
  }
  public String getColor() {
    return color;
  }
  public void display() {
    System.out.println("Light Mode..." + color);
  }
}

class MyCustomMode implements Mode {
  private Mode mode;

  public MyCustomMode(Mode mode) {
    this.mode = mode;
  }

  public String getColor() {
    return mode.getColor();
  }

  public void display() {
    mode.display();
  }
}
class App{
  public static void main(String[] args) {


    MyCustomMode mimodo= new MyCustomMode(new DarkMode("OSCURO PERSONALIZADO"));
    mimodo.display();
    mimodo= new MyCustomMode(new LightMode("CLARO  PERSONALIZADO"));
    mimodo.display();

  }
}
```

Como siempre, tenemos que hacer un esfuerzo de imaginación para ver que esta solución es buena. En lo más importante que te tienes que fijar es que tienes un modo personalizado que reutiliza el código de display() de otro objeto gracias a composición y además que ya que seguimos delegate extendido el display() puede ser de cualquier objeto que implemente el interface Mode. En un ejemplo real además podría usar la delegación de varios métodos y la clase MyCustomMode además de reutilizar el display() de otros objetos tendría funcionalidades extra que justifican su existencia.

EJERCICIO:
Utilizando le patrón delegate extendido, añade al siguiente código un delegador ControlReproductor

```
interface Reproductor {
  void reproducir();

  void pausar();
```

```
    void detener();
}



class ReproductorMp3 implements Reproductor {
    private String archivo;

    public ReproductorMp3(String archivo) {
        this.archivo = archivo;
    }

    public void reproducir() {
        System.out.println("Reproduciendo archivo " + archivo);
    }

    public void pausar() {
        System.out.println("Pausando reproducción del archivo " + archivo);
    }

    public void detener() {
        System.out.println("Deteniendo reproducción del archivo " + archivo);
    }
}




public class App {

    public static void main(String[] args) {

        ReproductorMp3 reproductorMp3 = new ReproductorMp3("cancion.mp3");
        ControlReproductor controlReproductor = new ControlReproductor(reproductorMp3);
        controlReproductor.reproducir();
        controlReproductor.pausar();
        controlReproductor.detener();

    }
}
```

SALIDA:
Reproduciendo archivo cancion.mp3
Pausando reproducción del archivo cancion.mp3
Deteniendo reproducción del archivo cancion.mp3