

9
2

Introducción al diseño OO

9
2

Jerarquías

Interfaces y clases abstractas

Introducción

Si en la primera parte del tema nos centrábamos en los elementos básicos del modelo de objetos, ahora comenzaremos esta segunda parte hablando de algo tan fundamental en el diseño OO como son las **relaciones** (las colaboraciones entre objetos).

Más concretamente, empezaremos analizando los distintos **tipos de herencia** que existen (de interfaces y de implementación) y, relacionado con esto, cuándo y por qué debemos usar interfaces y para qué sirven entonces las clases abstractas (si es que sirven para algo).



```
class Dibujo {  
  
    private Rectángulo[] rectángulos = new Rectángulo[30];  
    private int contador = 0;  
  
    public void añadir(Rectángulo rectángulo) {  
        rectángulos[contador++] = rectángulo;  
    }  
  
    public void dibujar() {  
        for (int i = 0; i < contador; i++) {  
            System.out.println(rectángulos[i].x1 + ", " + rectángulos[i].y1);  
            System.out.println(rectángulos[i].x2 + ", " + rectángulos[i].y2);  
        }  
    }  
}
```



```
class Rectángulo {  
    int x1, y1;  
    int x2, y2;  
}
```





```
Dibujo dibujo = new Dibujo();

dibujo.añadir(new Rectángulo(10, 10, 20, 20));
dibujo.añadir(new Rectángulo(30, 30, 40, 40));

dibujo.dibujar();
```

¿Diseño correcto?



```
class Rectángulo {  
    int x1, y1;  
    int x2, y2;  
}
```

```
Dibujo dibujo = new Dibujo();
```

```
} ]  
    dibujo.añadir(new Rectángulo(10, 10, 20, 20));  
    dibujo.añadir(new Rectángulo(30, 30, 40, 40));  
dibujo.dibujar();
```

```
tángulos = new Rectángulo[30];  
;  
ígulo rectángulo) {  
: rectángulo;
```



```
class Rectángulo {  
    int x1, y1;  
    int x2, y2;  
}
```



```
class Dibujo {  
  
    private Rectángulo[] rectángulos = new Rectángulo[30];  
    private int contador = 0;  
  
    public void añadir(Rectángulo rectángulo) {  
        rectángulos[contador++] = rectángulo;  
    }  
  
    public void dibujar() {  
        for (int i = 0; i < contador; i++) {  
            System.out.println(rectángulos[i].x1 + ", " + rectángulos[i].y1);  
            System.out.println(rectángulos[i].x2 + ", " + rectángulos[i].y2);  
        }  
    }  
}
```

```
Dibujo dibujo = new Dibujo();
```

```
dibujo.añadir(new Rectángulo(10, 10, 20, 20))  
dibujo.añadir(new Rectángulo(30, 30, 40, 40))  
dibujo.dibujar();
```



Qué ocurre si . . .

El rectángulo pasa a estar
definido por un único
punto más su ancho y alto.

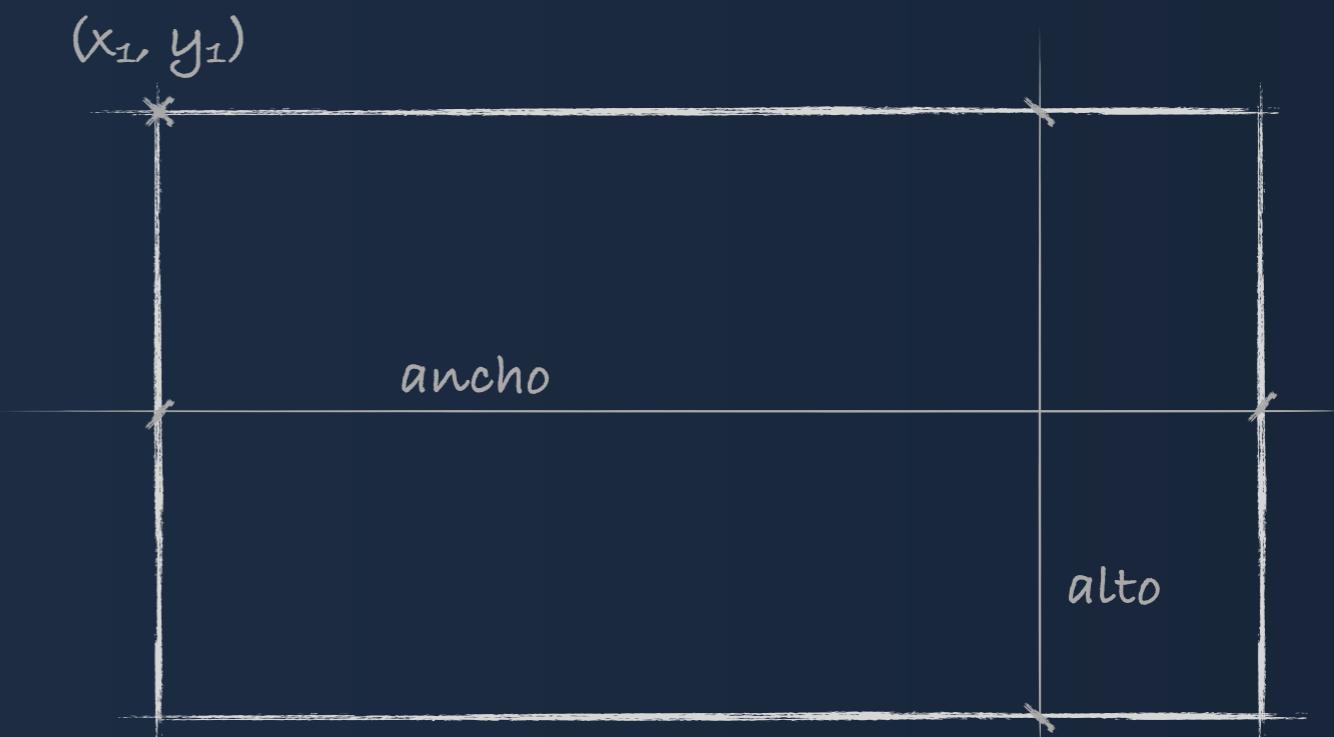
¿Afecta a la clase Dibujo?

¿Debería?



Rectángulo.java

```
class Rectángulo {  
    int x1, y1;  
    int ancho, alto;  
}
```





```
class Dibujo {  
  
    private Rectángulo[] rectángulos = new Rectángulo[30];  
    private int contador = 0;  
  
    public void añadir(Rectángulo rectángulo) {  
        rectángulos[contador++] = rectángulo;  
    }  
  
    public void dibujar() {  
        for (int i = 0; i < contador; i++) {  
            System.out.println(rectángulos[i].x1 + ", " + rectángulos[i].y1);  
            System.out.println(rectángulos[i].x2 + ", " + rectángulos[i].y2);  
        }  
    }  
}
```



```
class Dibujo {  
  
    private Rectángulo[] rectángulos = new Rectángulo[30];  
    private int contador = 0;  
  
    public void añadir(Rectángulo rectángulo) {  
        rectángulos[contador++] = rectángulo;  
    }  
  
    public void dibujar() {  
        for (int i = 0; i < contador; i++) {  
            System.out.println(rectángulos[i].x1 + ", " + rectángulos[i].y1);  
            System.out.println(rectángulos[i].x1 + rectángulos[i].ancho + ", "  
                + rectángulos[i].y2 + rectángulos[i].alto);  
        }  
    }  
}
```

Ahora queremos añadir un círculo.

¿Qué habría que cambiar de la clase Dibujo?

¿Cuál es el
problema?

Por un lado, que hemos basado el diseño de la clase enteramente en los atributos.

Recordemos que lo importante de una clase es su comportamiento y que por tanto hemos de empezar siempre por los métodos (los atributos no son más que un mero apoyo de estos).



Dibujo.java

```
class Dibujo {  
  
    private Rectángulo[] rectángulos = new Rectángulo[30];  
    private int contador = 0;  
  
    public void añadir(Rectángulo rectángulo) {  
        rectángulos[contador++] = rectángulo;  
    }  
  
    public void dibujar() {  
        for (int i = 0; i < contador; i++) {  
            System.out.println(rectángulos[i].x1 + ", " + rectángulos[i].y1);  
            System.out.println(rectángulos[i].x2 + ", " + rectángulos[i].y2);  
        }  
    }  
}
```



Dibujo.java

```
class Dibujo {  
  
    private Rectángulo[] rectángulos = new Rectángulo[30];  
    private int contador = 0;  
  
    public void añadir(Rectángulo rectángulo) {  
        rectángulos[contador++] = rectángulo;  
    }  
  
    public void dibujar() {  
        for (int i = 0; i < contador; i++) {  
            System.out.println(rectángulos[i].x1 + ", " + rectángulos[i].y1);  
            System.out.println(rectángulos[i].x2 + ", " + rectángulos[i].y2);  
        }  
    }  
}
```

Nótese que no es solo porque estemos accediendo directamente a los atributos.



Dibujo.java

```
class Dibujo {  
  
    private Rectángulo[] rectángulos = new Rectángulo[30];  
    private int contador = 0;  
  
    public void añadir(Rectángulo rectángulo) {  
        rectángulos[contador++] = rectángulo;  
    }  
  
    public void dibujar() {  
        for (int i = 0; i < contador; i++) {  
            System.out.println(rectángulos[i].x1 + ", " + rectángulos[i].y1);  
            System.out.println(rectángulos[i].x2 + ", " + rectángulos[i].y2);  
        }  
    }  
}
```



Dibujo.java

```
class Dibujo {  
  
    private Rectángulo[] rectángulos = new Rectángulo[30];  
    private int contador = 0;  
  
    public void añadir(Rectángulo rectángulo) {  
        rectángulos[contador++] = rectángulo;  
    }  
  
    public void dibujar() {  
        for (int i = 0; i < contador; i++) {  
            System.out.println(rectángulos[i].x1() + ", " + rectángulos[i].y1());  
            System.out.println(rectángulos[i].x2() + ", " + rectángulos[i].y2());  
        }  
    }  
}
```

Por otro, que el dibujo tiene
que **conocer** a los objetos
con los que colabora.

Y tratar a cada uno por separado.

Lo cual, dicho sea de paso, es también consecuencia de lo anterior: tienen **distintos atributos**.

La pregunta que debemos hacernos, la forma de diseñar
correctamente, es...

¿Qué necesita el
dibujo de los
objetos con los
que colabora?

**Que sepan cómo
dibujarse.**



```
class Dibujo {  
  
    private Rectángulo[] rectángulos = new Rectángulo[30];  
    private int contador = 0;  
  
    public void añadir(Rectángulo rectángulo) {  
        rectángulos[contador++] = rectángulo;  
    }  
  
    public void dibujar() {  
        for (int i = 0; i < contador; i++) {  
            System.out.println(rectángulos[i].x1 + ", " + rectángulos[i].y1);  
            System.out.println(rectángulos[i].x2 + ", " + rectángulos[i].y2);  
        }  
    }  
}
```



```
class Dibujo {  
  
    private Figura[] figuras = new Figura[30];  
    private int contador = 0;  
  
    public void añadirFigura(Figura figura) {  
        figuras[contador++] = figura;  
    }  
  
    public void dibujar() {  
        for (int i = 0; i < contador; i++) {  
            figuras[i].dibujar();  
        }  
    }  
}
```

¿Cómo se lo exige?



```
class Dibujo {  
  
    private Figura[] figuras = new Figura[30];  
    private int contador = 0;  
  
    public void añadirFigura(Figura figura) {  
        figuras[contador++] = figura;  
    }  
  
    public void dibujar() {  
        for (int i = 0; i < contador; i++) {  
            figuras[i].dibujar();  
        }  
    }  
}
```

¿Qué es
«Figura»?

¿Tiene sentido crear
objetos «Figura»?

¿El dibujo necesita
una determinada
implementación de
dichos objetos?

**A través de una
interfaz.**



Figura.java

```
interface Figura {  
    void dibujar();  
}
```



Rectángulo.java

```
class Rectángulo implements Figura {  
  
    private int x1, y1; // Esquina superior izquierda  
    private int x2, y2; // Esquina inferior derecha  
  
    public void dibujar() {  
        System.out.println(x1 + ", " + y1);  
        System.out.println(x2 + ", " + y2);  
    }  
}
```



Figura.java

```
interface Figura {  
    void dibujar();  
}
```



Rectángulo.java

```
class Rectángulo implements Figura {  
  
    private int x1, y1; // Esquina superior izquierda  
    private int x2, y2; // Esquina inferior derecha  
  
    public void dibujar() {  
        System.out.println(x1 + ", " + y1);  
        System.out.println(x2 + ", " + y2);  
    }  
}
```

¿Por qué una interfaz?

(En vez de, por ejemplo, una clase abstracta).

**Interfaces
frente a clases
abstractas**

**Permiten comunicar objetos
que no se conocen.**

Ejemplo

Un objeto jefe necesita delegar en otro, empleado,
parte de su trabajo.

En este caso, bailar.

¿Tiene que ser alguien en concreto?

¿Tiene que ser alguien en concreto?

No. Hay varios empleados que saben bailar.

¿Tiene que ser alguien en concreto?

No. Hay muchas personas que saben bailar.

Pero, ¿puede ser cualquiera?

¿Tiene que ser alguien en concreto?

No. Hay varios empleados que saben bailar.

Pero, ¿puede ser cualquiera?

Tampoco: tiene que saber bailar.

¿Cómo indica el jefe las condiciones que debe cumplir el empleado para que le sea útil?



Jefe.java

```
class Jefe {  
    // ...  
  
    public void contrata(                         empleado) {  
        // ...  
    }  
  
    public void trabaja() {  
        // ...  
        empleado.baila();  
        // ...  
    }  
}
```

¿Qué ponemos aquí?



Las clases no sirven.

Se quiere poder contratar a empleados de distintas clases.

El jefe especifica en un contrato qué operaciones requiere.



EmpleadoBailón.java

```
interface EmpleadoBailón {  
    void baila();  
}
```

Y utiliza dicho contrato como el tipo del objeto requerido.



Jefe.java

```
class Jefe {  
    // ...  
  
    public void contrata(                         empleado) {  
        // ...  
    }  
  
    public void trabaja() {  
        // ...  
        empleado.baila();  
        // ...  
    }  
}
```



Jefe.java

```
class Jefe {  
    // ...  
  
    public void contrata(EmpleadoBailón empleado) {  
        // ...  
    }  
  
    public void trabaja() {  
        // ...  
        empleado.baila();  
        // ...  
    }  
}
```

Cualquier objeto que quiera ser contratado debe cumplir el contrato.



Británico.java

```
class Británico implements EmpleadoBailón {  
    public void baila() {  
        System.out.println("Y con salero, como buen inglés.");  
    }  
}
```



Catedrático.java

```
class Catedrático implements EmpleadoBailón {  
    public void baila() {  
        System.out.println("¡Ahora se lo mando a mi ayudante!");  
    }  
}
```



Británico.java

```
class Británico implements EmpleadoBailón {  
    public void baila() {  
        System.out.println("Y con salero, como buen inglés.");  
    }  
}
```



Catedrático.java

```
class Catedrático implements EmpleadoBailón {  
    public void baila() {  
        System.out.println("¡Ahora se lo mando a mi ayudante!");  
    }  
}
```



Británico.java

```
class Británico implements EmpleadoBailón {  
    public void baila() {  
        System.out.println("Y con salero, como buen inglés.");  
    }  
}
```



Catedrático.java

```
class Catedrático implements EmpleadoBailón {  
    public void baila() {  
        System.out.println("¡Ahora se lo mando a mi ayudante!");  
    }  
}
```

Nótese que «Británico» y «Catedrático» pueden no formar parte de la misma jerarquía de clases (no compartir ninguna relación de herencia de implementación).

Y sin embargo, en el contexto actual, se pueden sustituir objetos de ambos tipos sin tener conocimiento previo de cada uno (o de una nueva clase que pudiera surgir en un futuro, siempre y cuando también implementase la misma interfaz).

La revisión del contrato se hará de manera
estática.



```
Jefe jefe = new Jefe();
```

```
EmpleadoBailón peter = new Británico();
```

```
EmpleadoBailón pepe = new Catedrático();
```

```
jefe.contrata(peter);
```

```
jefe.contrata(pepe);
```

¿PETER Y PEPE
IMPLEMENTAN LA
MISMA INTERFAZ?

// Y aquí es donde quiebra el negocio

// (a pesar del arte de sus bailarines)

Sí, y se sabe en tiempo
de compilación.

Tipos de herencia

De interfaz

Declaramos el compromiso de implementar unos tipos.

De interfaz

Define tipos y subtipos.

De implementación

Creamos una clase a partir de la implementación de otra.

De implementación

Es un mecanismo de reutilización
de código.

criterio de diseño

*Cuando un objeto deba
delegar ciertas operaciones
en otro deberá exigírselas
mediante una interfaz.*

Criterio de diseño

Importan los mensajes que
acepta el objeto.

Criterio de diseño

No de quién ha obtenido su
implementación.

Interfaces

Reutilización sin acoplamiento

**Veamos cómo las interfaces
son el medio para participar
en colaboraciones ya
implementadas.**

En una aplicación se necesita ordenar facturas.



Factura.java

```
class Factura {  
    // ...  
    public int getNúmero() { ... }  
    // ...  
}
```



Ordenatriz.java

```
class Ordenatriz {  
    public void ordena(Factura[] facturas) {  
        for (int i = 0; i < facturas.length - 1; i++) {  
            for (j = i + 1; j < facturas.length; j++) {  
                if (facturas[j].getNúmero() < facturas[i].getNúmero()) {  
                    Factura temp = facturas[i];  
                    facturas[i] = facturas[j];  
                    facturas[j] = temp;  
                }  
            }  
        }  
    }  
}
```

¿Y si ahora se quiere ordenar personas?



Persona.java

```
class Persona {  
    // ...  
    public String getNombre() { ... }  
    // ...  
}
```



Ordenatriz.java

```
class Ordenatriz {  
    public void ordena(Persona[] personas) {  
        for (int i = 0; i < personas.length - 1; i++) {  
            for (j = i + 1; j < personas.length; j++) {  
                if (personas[j].getNombre() < personas[i].getNombre()) {  
                    Persona temp = personas[i];  
                    personas[i] = personas[j];  
                    personas[j] = temp;  
                }  
            }  
        }  
    }  
}
```

Solución



Ordenatriz.java

```
class Ordenatriz {  
    public void ordena(Ordenable[] elementos)  
    {  
        for (int i = 0; i < elementos.length - 1; i++) {  
            for (j = i + 1; j < elementos.length; j++) {  
                if (elementos[j].anteriorA(elementos[i])) {  
                    Object temp = elementos[i];  
                    elementos[i] = elementos[j];  
                    elementos[j] = temp;  
                }  
            }  
        }  
    }  
}
```



Ordenable.java

```
interface Ordenable {
    // ...
    public boolean anteriorA() { ... }
    // ...
}
```



Ordenable.java

```
interface Ordenable {  
    // ...  
    public boolean anteriorA() { ... }  
    // ...  
}
```

```
class Ordenatriz {  
    public void ordena(Ordenable[] elementos)  
    {  
        for (int i = 0; i < elementos.length - 1; i++) {  
            for (j = i + 1; j < elementos.length; j++) {  
                if (elementos[j].anteriorA(elementos[i])) {  
                    Object temp = elementos[i];  
                    elementos[i] = elementos[j];  
                    elementos[j] = temp;  
                }  
            }  
        }  
    }  
}
```

```
class Ordenatriz {  
    public void ordena(Ordenable[] elementos)  
    {  
        for (int i = 0; i < elementos.length - 1; i++) {  
            for (j = i + 1; j < elementos.length; j++) {  
                if (elementos[j].anteriorA(elementos[i])) {  
                    Object temp = elementos[i];  
                    elementos[i] = elementos[j];  
                    elementos[j] = temp;  
                }  
            }  
        }  
    }  
}
```

Otra aplicación
quiere hacer una
búsqueda
dicotómica



Detective.java

```
class Detective {  
    public int buscar(Buscable[] elementos, Object identificador) {  
        int inferior = 0;  
        int superior = elementos.length - 1;  
  
        while (inferior <= superior) {  
            int medio = (inferior + superior) / 2;  
            if (elementos[medio].igualA(identificador))  
                return medio;  
            if (elementos[medio].menorQue(identificador))  
                inferior = medio + 1;  
            else  
                superior = medio - 1;  
        }  
        return -1;  
    }  
}
```



Buscable.java

```
interface Buscable {  
    boolean igualA(Object id);  
    boolean menorQue(Object id);  
}
```



Detective.java

```
class Detective {  
    public int buscar(Buscable[] elementos, Object identificador) {  
        int inferior = 0;  
        int superior = elementos.length - 1;  
  
        while (inferior <= superior) {  
            int medio = (inferior + superior) / 2;  
            if (elementos[medio].igualA(identificador))  
                return medio;  
            if (elementos[medio].menorQue(identificador))  
                inferior = medio + 1;  
            else  
                superior = medio - 1;  
        }  
        return -1;  
    }  
}
```



Buscable.java

```
interface Buscable {  
    boolean igualA(Object id);  
    boolean menorQue(Object id);  
}
```

Ahora queremos ordenar un array de personas y luego buscar en él.



```
public static void main(String[] args) {  
    Persona[] personas = new Persona[100];  
  
    for (int i=0; i < 100; i++)  
        personas[i] = new Persona(); // al azar  
  
    ordenatriz.ordena(personas);  
    detective.busca(personas, "pepe");  
}
```



Persona.java

```
class Persona implements Buscable, Ordenable {  
    public String getName() {...}  
  
    public boolean igualA(Object id) {  
        return getName().equals((String) id);  
    }  
  
    public boolean menorQue(Object id) {  
        return getName().compareTo((String) id) < 0;  
    }  
  
    public boolean anteriorA(Ordenable o) {  
        Persona otra = (Persona) o;  
        return getName().compareTo(otra.getName()) < 0;  
    }  
}
```

Reutilización con interfaces

En el ejemplo anterior han colaborado tres módulos distintos que podrían haber sido escritos:

Por distintas personas

En distinto momento

Sin conocimiento previo

Persona

Búsquedas

Ordenación

No se podría haber hecho si
ordena hubiera exigido parámetros
de una clase concreta.

¡La clase Persona ni siquiera existía todavía!

Clases abstractas

Y sus dos únicas aplicaciones

Supongamos un objeto que delega responsabilidades en otros que cumplen una determinada interfaz.





Secta.java

```
class Secta {  
    public void atrapa(Adepto persona) {  
        persona.dameTusPosesiones();  
        persona.trabaja();  
        persona.habla();  
        persona.duerme();  
    }  
}
```



Adepto.java

```
interface Adepto {  
    void dameTusPosesiones();  
    void duerme();  
    void trabaja();  
    void habla();  
}
```

Cada clase a conectar deberá proveer una implementación para todos los métodos de la interfaz.



Jubilado.java

```
class Jubilado implements Adepto {  
  
    public void duerme() {  
        System.out.println("zzzZZZZzz");  
    }  
  
    public void trabaja() {  
        System.out.println("zzzZZZZzz");  
    }  
  
    public void habla() {  
        System.out.println("¡Esta obra va muy lenta!");  
    }  
  
    public void dameTusPosesiones() {  
        System.out.println("Ni un duro");  
    }  
}
```



Catedrático.java

```
class Catedrático implements Adepto {  
  
    public void duerme() {  
        System.out.println("zzzZZZZzz");  
    }  
  
    public void trabaja() {  
        System.out.println("zzzZZZZzz");  
    }  
  
    public void habla() {  
        System.out.println("¡El COBOL es el futuro, queridos alumnos!");  
    }  
  
    public void dameTusPosesiones() {  
        System.out.println("Ni un duro");  
    }  
}
```



Jubilado.java

```
class Jubilado implements Adepto {  
  
    public void duerme() {  
        System.out.println("zzzzZZzz");  
    }  
  
    public void trabaja() {  
        System.out.println("zzzZZZZzz");  
    }  
  
    public void habla() {  
        System.out.println("¡Esta obra va muy  
lenta!");  
    }  
  
    public void dameTusPosesiones() {  
        System.out.println("Ni un duro");  
    }  
}
```



Catedrático.java

```
class Catedrático implements Adepto {  
  
    public void duerme() {  
        System.out.println("zzzZZZzz");  
    }  
  
    public void trabaja() {  
        System.out.println("zzzZZZzz");  
    }  
  
    public void habla() {  
        System.out.println("¡El COBOL es el  
futuro, queridos alumnos!");  
    }  
  
    public void dameTusPosesiones() {  
        System.out.println("Ni un duro");  
    }  
}
```

¿Qué observamos?

Jubilado y catedrático hacen prácticamente lo mismo.

¿Hay algún modo de evitar esa duplicación de código?

Efectivamente, mediante
las clases abstractas.



Adepto.java

```
interface Adepto {  
    void dameTusPosesiones();  
    void duerme();  
    void trabaja();  
    void habla();  
}
```



AbstractAdepto.java

```
abstract class AbstractAdepto implements Adepto {  
  
    public void duerme() {  
        System.out.println("zzzZZZzz");  
    }  
  
    public void trabaja() {  
        System.out.println("zzzzZZZzz");  
    }  
  
    public void dameTusPosesiones() {  
        System.out.println("Ni un duro");  
    }  
}
```



Jubilado.java

```
class Jubilado extends AbstractAdepto {  
    public void habla() {  
        System.out.println("¡Esta obra va muy lenta!");  
    }  
}
```



Catedrático.java

```
class Catedrático extends AbstractAdepto {  
    public void habla() {  
        System.out.println("¡El COBOL es el futuro, queridos alumnos!");  
    }  
}
```

```
class Catedratico extends AbstractAdepto {  
    public void habla() {  
        System.out.println("¡El COBOL es el futuro, queridos alumnos!");  
    }  
}
```



Adepto.java

```
interface Adepto {  
    void dameTusPosesiones();  
    void duerme();  
    void trabaja();  
    void habla();  
}
```



Jubilado.java

```
class Jubilado extends AbstractAdepto {  
    public void habla() {  
        System.out.println("¡Esta obra va muy lenta!");  
    }  
}
```



AbstractAdepto.java

```
abstract class AbstractAdepto implements Adepto {  
    public void dameTusPosesiones() {...}; // Ni un duro  
    public void duerme() {...}; // zzzZZZZZZZ  
    public void trabaja() {...}; // zzzZZZZZZZ  
}
```



Catedrático.java

```
class Catedrático extends AbstractAdepto {  
    public void habla() {  
        System.out.println("¡El COBOL es el  
futuro, queridos alumnos!");  
    }  
}
```

**Ese es el primer uso de las
clases abstractas: factorizar
código propio.**

(Para eliminar el código repetido).

AbstractAdepto es la clase base abstracta.

Quedan mensajes sin implementar: lo que no es común.

Por tanto, no se puede «instanciar».

Se usa como implementación base para crear otras clases.

Una clase base también
puede implementar todos los
mensajes de la interfaz.

En ese caso se denomina clase base
concreta.

Se suelen llamar «Default-» o «Standard-».

Se pueden «instanciar» directamente.

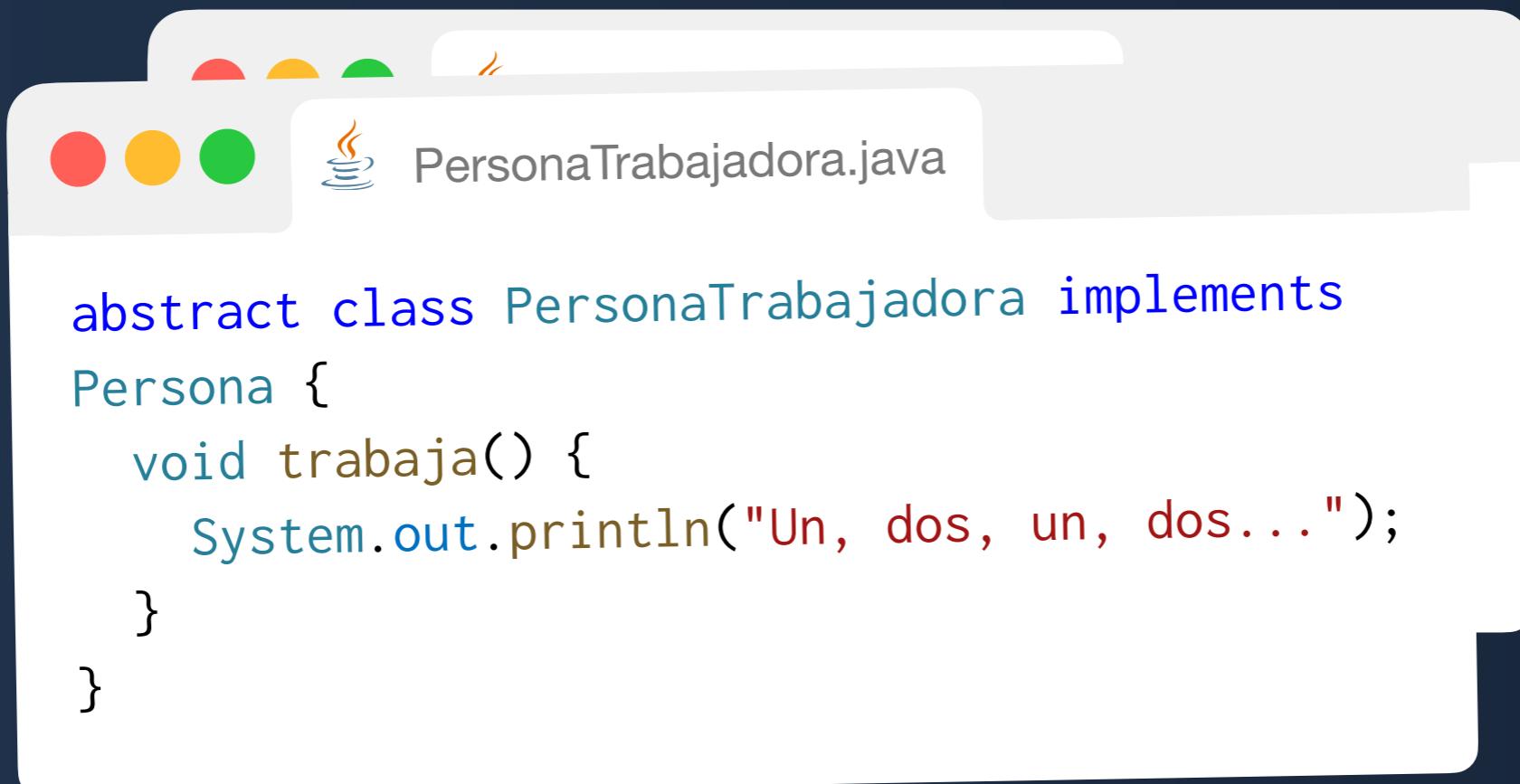
**El segundo uso principal es
facilitar la extensión.**

Son proporcionadas por el autor
de la biblioteca o el framework.



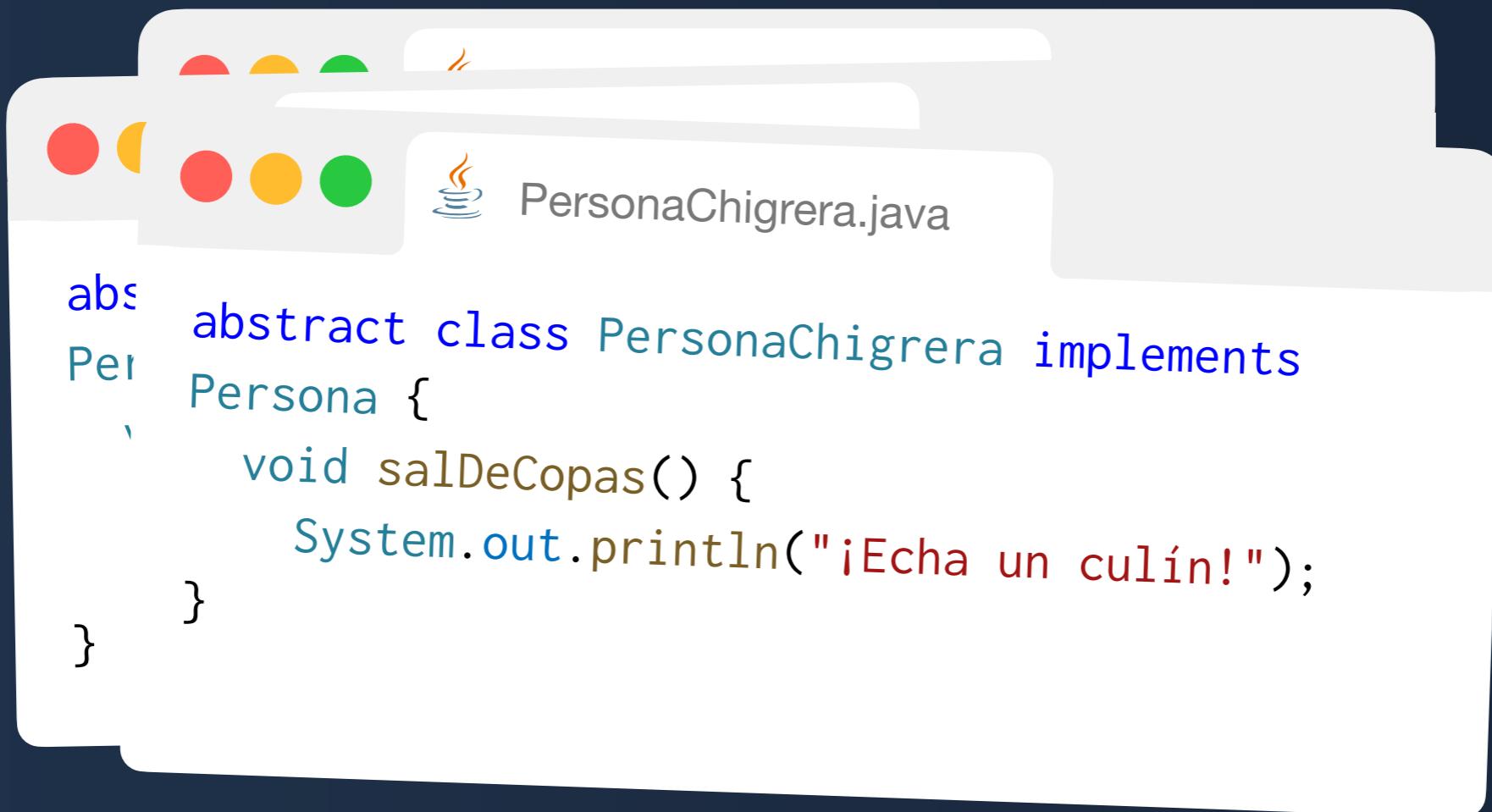
Persona.java

```
interface Persona {  
    void trabaja();  
    void salDeCopas();  
}
```



A Scratch script window titled "PersonaTrabajadora.java". The script contains the following Java code:

```
abstract class PersonaTrabajadora implements  
Persona {  
    void trabaja() {  
        System.out.println("Un, dos, un, dos...");  
    }  
}
```



A screenshot of a Java code editor showing an abstract class named `PersonaChigrera.java`. The code defines an abstract class `PersonaChigrera` that implements the `Persona` interface. It contains a single method `salDeCopas()` which prints the message "¡Echa un culín!" to the console.

```
abs abstract class PersonaChigrera implements
Per Persona {
    void salDeCopas() {
        System.out.println("¡Echa un culín!");
    }
}
```



Persona.java

```
interface Persona {  
    void trabaja();  
    void salDeCopas();  
}
```



PersonaTrabajadora.java

```
abstract class PersonaTrabajadora implements  
Persona {  
    void trabaja() {  
        System.out.println("Un, dos, un, dos...")  
    }  
}
```



PersonaChigrera.java

```
abstract class PersonaChigrera implements  
Persona {  
    void salDeCopas() {  
        System.out.println("¡Echa un culín!");  
    }  
}
```

«interface»
Persona

trabaja()
salDeCopas()

Persona
Trabajadora

trabaja()

Persona
Chigrera

salDeCopas()





EnanoDeBlancanieves.java

```
class EnanoDeBlancanieves extends  
PersonaTrabajadora {  
    void salDeCopas() {  
        System.out.println("¡Blancanieves, guapa!,  
¿una copita?");  
    }  
}
```



CantanteDeRock.java

```
class CantanteDeRock extends PersonaChigrera {  
    void trabaja() {  
        System.out.println("...");  
    }  
}
```



Persona.java

```
interface Persona {
```



PersonaTrabajadora.java

Incluido con el
framework

```
abstract class PersonaTrabajadora implements
```



PersonaChigrera.java

```
abstract class PersonaChigrera implements  
Persona {
```



EnanoDeBlancanieves.java

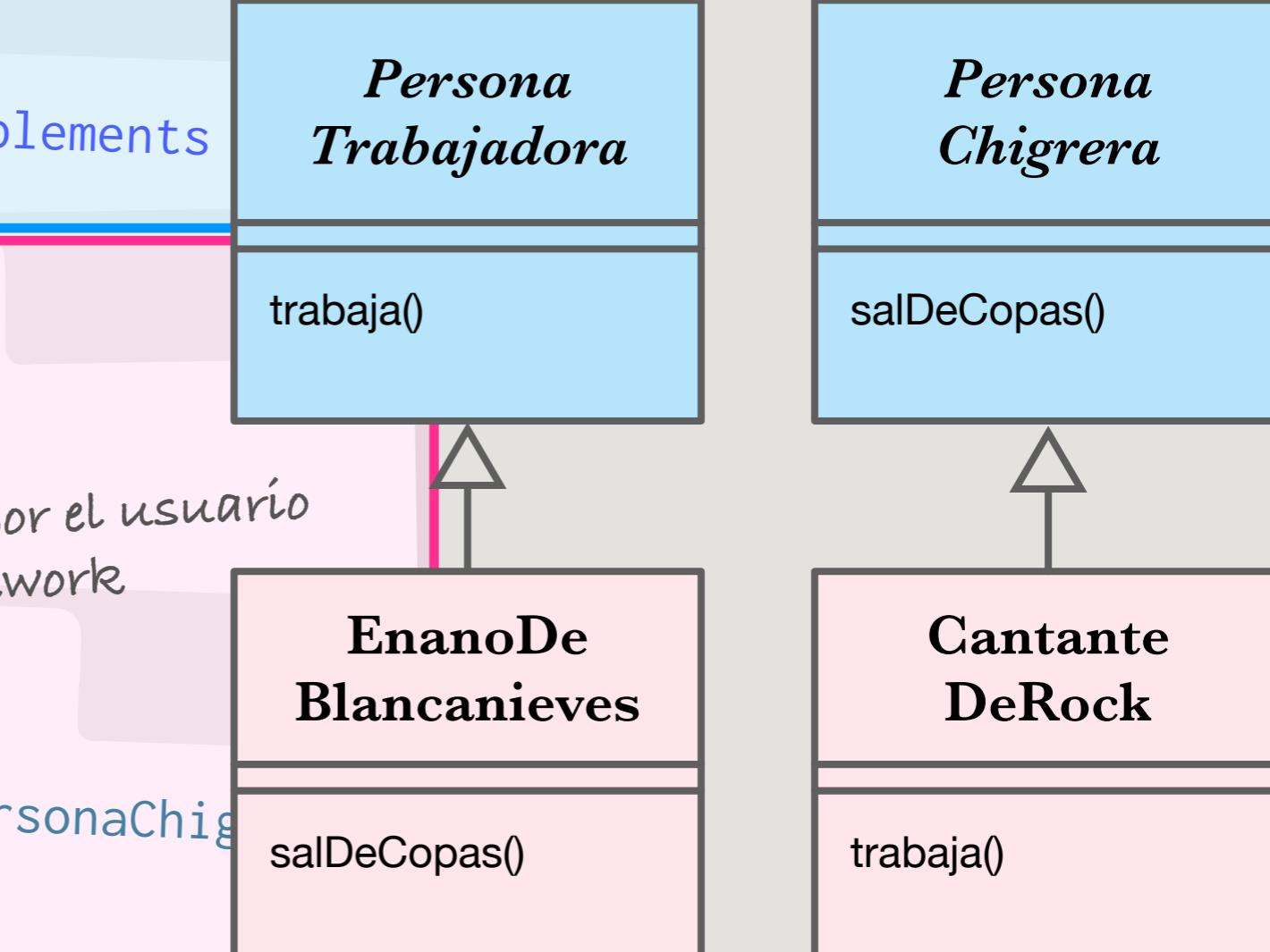
```
class EnanoDeBlancanieves extends  
PersonaTrabajadora {
```



CantanteDeRock.java

```
class CantanteDeRock extends PersonaChig  
void trabaja() {  
    System.out.println("...");  
}
```

Creadas por el usuario
del framework



El creador de la interfaz suele acompañarla de una o varias implementaciones predeterminadas.

A la hora de implementarla, miraremos primero qué implementaciones, totales (clases concretas listas para ser usadas) o parciales (es decir, abstractas), nos vienen ya dadas.

El creador de la interfaz suele acompañarla de una o varias implementaciones predeterminadas.

Escogeremos la más cercana que nos sirva.

En caso contrario, la implementamos desde cero.

Ejemplos

Tool y AbstractTool

Drawing y StandardDrawing



Java SE 22 & JDK 22

SEARCH



Search

**Module** java.base**Package** java.util

Interface List<E>

Type Parameters:

E - the type of elements in this list

All Superinterfaces:

[Collection<E>](#), [Iterable<E>](#), [SequencedCollection<E>](#)

All Known Subinterfaces:

[ClassPrinter.ListNode](#)PREVIEW

All Known Implementing Classes:

[AbstractList](#), [AbstractSequentialList](#), [ArrayList](#), [AttributeList](#), [CopyOnWriteArrayList](#), [LinkedList](#), [RoleList](#), [RoleUnresolvedList](#), [Stack](#), [Vector](#)

```
public interface List<E>
extends SequencedCollection<E>
```

An ordered collection, where the user has precise control over where in the list each element is inserted. The user can access elements by their integer index (position in the list), and search for elements in the list.

Unlike sets, lists typically allow duplicate elements. More formally, lists typically allow pairs of elements e1 and e2 such that e1.equals(e2), and they typically allow multiple null elements if they allow null elements at all. It is

Module java.base
Package java.util

Class AbstractList<E>

java.lang.Object
 java.util.AbstractCollection<E>
 java.util.AbstractList<E>

Type Parameters:
E - the type of elements in this list

All Implemented Interfaces:
[Iterable<E>](#), [Collection<E>](#), [List<E>](#), [SequencedCollection<E>](#)

Direct Known Subclasses:
[AbstractSequentialList](#), [ArrayList](#), [Vector](#)

```
public abstract class AbstractList<E>
extends AbstractCollection<E>
implements List<E>
```

This class provides a skeletal implementation of the [List](#) interface to minimize the effort required to implement this interface backed by a "random access" data store (such as an array). For sequential access data (such as a linked list), [AbstractSequentialList](#) should be used in preference to this class.

To implement an unmodifiable list, the programmer needs only to extend this class and provide implementations for the `set(int, E)` and `remove()` methods.



SEARCH



Search

**Module** java.base**Package** java.util

Class ArrayList<E>

[java.lang.Object](#)[java.util.AbstractCollection<E>](#)[java.util.AbstractList<E>](#)[java.util.ArrayList<E>](#)**Type Parameters:**

E - the type of elements in this list

All Implemented Interfaces:[Serializable](#), [Cloneable](#), [Iterable<E>](#), [Collection<E>](#), [List<E>](#), [RandomAccess](#), [SequencedCollection<E>](#)**Direct Known Subclasses:**[AttributeList](#), [RoleList](#), [RoleUnresolvedList](#)

```
public class ArrayList<E>
extends AbstractList<E>
implements List<E>, RandomAccess, Cloneable, Serializable
```

Resizable-array implementation of the [List](#) interface. Implements all optional list operations, and permits all elements, including `null`. In addition to implementing the [List](#) interface, this class provides methods to manipulate the size of the array that is used internally to store the list. (This class is roughly equivalent to [Vector](#), except that it is unsynchronized.)



SEARCH



Search

**Module** java.base**Package** java.util

Class LinkedList<E>

[java.lang.Object](#)[java.util.AbstractCollection<E>](#)[java.util.AbstractList<E>](#)[java.util.AbstractSequentialList<E>](#)[java.util.LinkedList<E>](#)**Type Parameters:**

E - the type of elements held in this collection

All Implemented Interfaces:[Serializable](#), [Cloneable](#), [Iterable<E>](#), [Collection<E>](#), [Deque<E>](#), [List<E>](#), [Queue<E>](#),[SequencedCollection<E>](#)

```
public class LinkedList<E>
extends AbstractSequentialList<E>
implements List<E>, Deque<E>, Cloneable, Serializable
```

Doubly-linked list implementation of the List and Deque interfaces. Implements all optional list operations, and permits all elements (including null).

All of the operations perform as could be expected for a doubly-linked list. Operations that index into the list will traverse the list from the beginning or the end, whichever is closer to the specified index.

The screenshot shows a web browser window with the URL docs.oracle.com. The page title is "Java SE 22 & JDK 22". The main content area displays information about the `AbstractSequentialList<E>` class, including its module (`java.base`), package (`java.util`), inheritance chain, type parameters, implemented interfaces, subclasses, and a code snippet. A search bar is visible at the top right.

Module `java.base`

Package `java.util`

Class `AbstractSequentialList<E>`

`java.lang.Object`

`java.util.AbstractCollection<E>`

`java.util.AbstractList<E>`

`java.util.AbstractSequentialList<E>`

Type Parameters:

`E` - the type of elements in this list

All Implemented Interfaces:

`Iterable<E>, Collection<E>, List<E>, SequencedCollection<E>`

Direct Known Subclasses:

`LinkedList`

```
public abstract class AbstractSequentialList<E>
extends AbstractList<E>
```

This class provides a skeletal implementation of the `List` interface to minimize the effort required to implement this interface backed by a "sequential access" data store (such as a linked list). For random access data (such as an array), `AbstractList` should be used in preference to this class.

This class is the opposite of the `AbstractList` class in the sense that it implements the "random access" methods (`get(int index)`, `set(int index, E element)`, `add(int index, E element)` and `remove(int index)`) on top

**Nótese el papel secundario
de la herencia de clases.**

Nótese el papel secundario de la herencia de clases.

No es más que un mecanismo para evitar la duplicación de código, en el primer caso.

Nótese el papel secundario de la herencia de clases.

O para hacer más cómoda la implementación de las interfaces,
en el segundo.

Si todos los métodos fuesen distintos
en cada clase final, no habría clases
abstractas en el programa.

Redefinición de métodos

**Hasta ahora, al derivar
de una clase abstracta:**

O se deja el método heredado.

**Hasta ahora, al derivar
de una clase abstracta:**

O se redefine.

¿Qué ocurre si, en vez de sustituir
toda la implementación heredada,
solo se quiere ampliarla?



Adepto.java

```
interface Adepto {  
    void dameTusPosesiones();  
    void duerme();  
    void trabaja();  
    void habla();  
}
```



AbstractAdepto.java

```
abstract class AbstractAdepto implements Adepto {  
  
    public void duerme() {  
        System.out.println("zzzZZZzz");  
    }  
  
    public void trabaja() {  
        System.out.println("zzzzZZZzz");  
    }  
  
    public void dameTusPosesiones() {  
        System.out.println("Ni un duro");  
    }  
}
```



AbstractAdepto.java

```
abstract class AbstractAdepto implements Adepto {  
    // ...  
    public void trabaja() {  
        System.out.println("zzzZZZZzz");  
    }  
}
```



Catedrático.java

```
class Catedrático extends AbstractAdepto {  
    public void trabaja() {  
        System.out.println("¡Que alguien escriba un artículo!");  
        System.out.println("zzzZZZzz");  
        System.out.println("¡Ponme como autor!");  
    }  
}
```



AbstractAdepto.java

```
abstract class AbstractAdepto implements Adepto {  
    // ...  
    public void trabaja() {  
        System.out.println("zzzZZZzzz");  
    }  
}
```



Catedrático.java

```
class Catedrático extends AbstractAdepto {  
    public void trabaja() {  
        System.out.println("¡Que alguien escriba un artículo!");  
        System.out.println("zzzZZZzzz");  
        System.out.println("¡Ponme como autor!");  
    }  
}
```



AbstractAdepto.java

```
abstract class AbstractAdepto implements Adepto {  
    // ...  
    public void trabaja() {  
        System.out.println("zzzZZZzzz");  
    }  
}
```



Catedrático.java

```
class Catedrático extends AbstractAdepto {  
    public void trabaja() {  
        System.out.println("¡Que alguien escriba un artículo!");  
        super.trabaja();  
        System.out.println("¡Ponme como autor!");  
    }  
}
```

Hasta ahora, una clase abstracta podía hacer dos cosas con un método:

Dejarlo sin implementación para que lo hagan las clases derivadas.

Hasta ahora, una clase abstracta podía hacer dos cosas con un método:

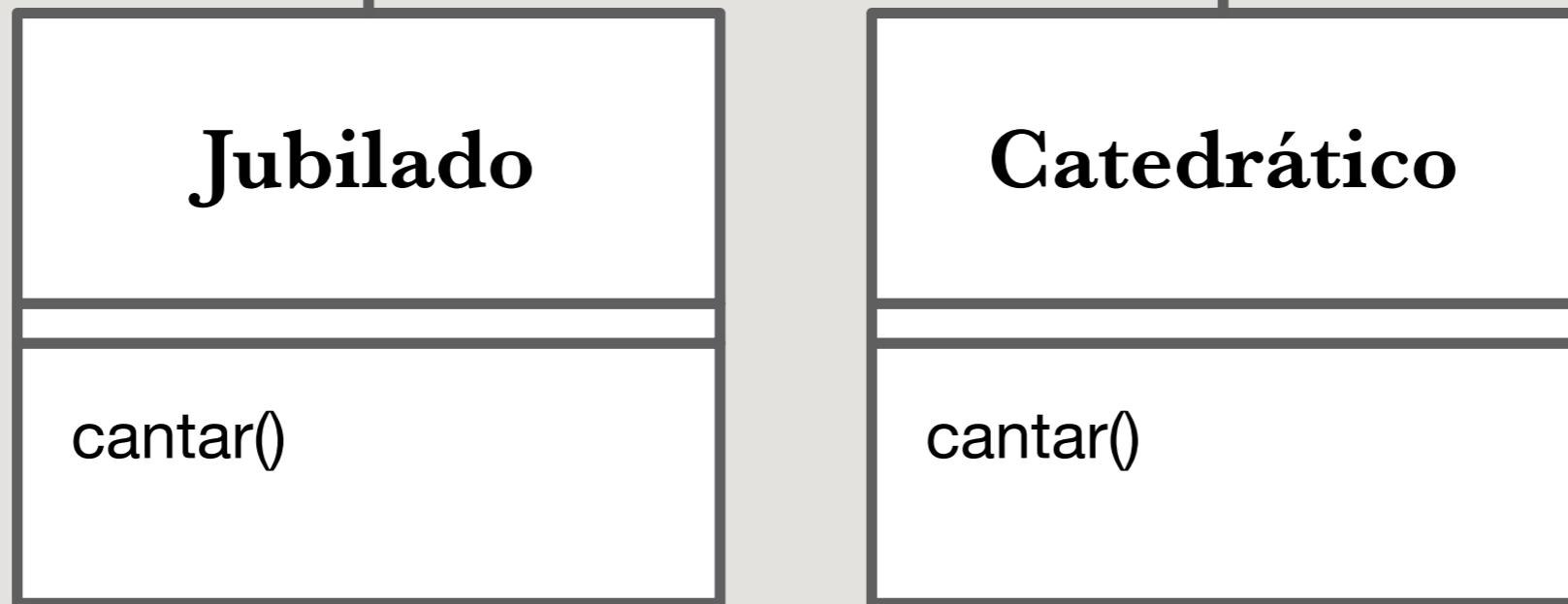
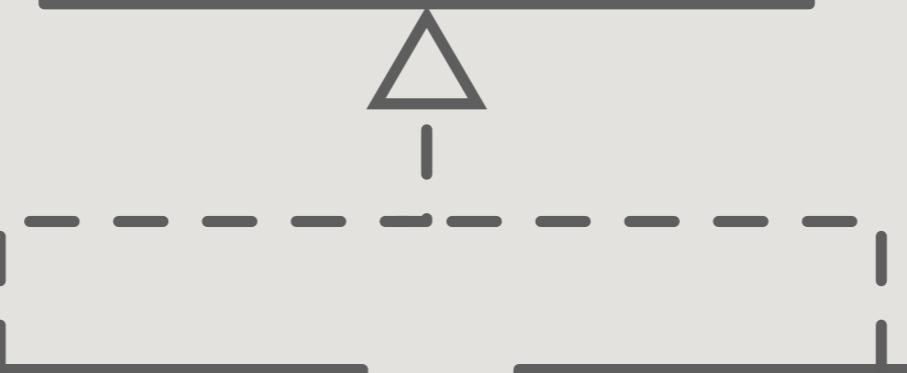
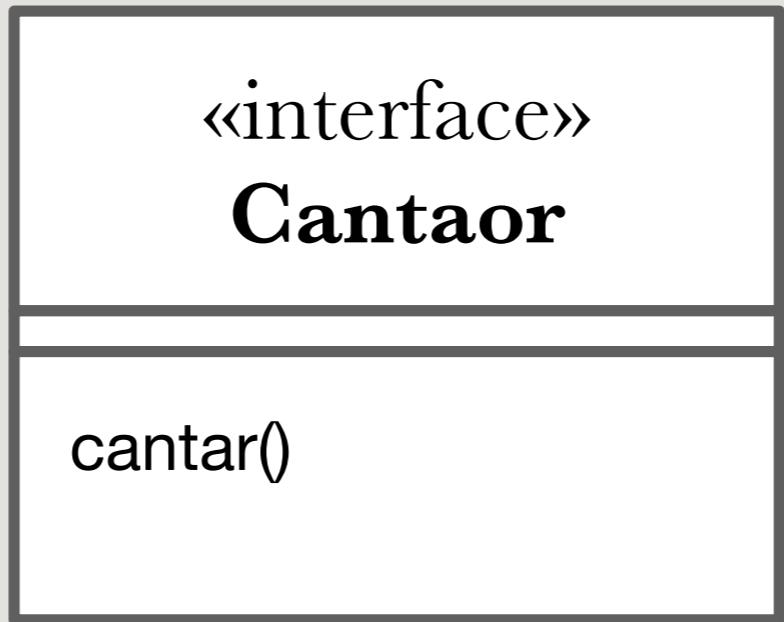
Dar la implementación completa del mensaje.

¿Tiene sentido dar partes de la implementación de un mensaje?

Es decir, acciones obligatorias que deben hacer todos.

Queremos que al añadir nuevas clases se cambie solo una parte de la implementación, no toda.





Aunque cada uno cante a su manera,
todos deben carraspear primero y
esquivar los tomates al terminar.

**¿Cómo conseguir que
cambie solo parte de
la implementación?**



Cantaor.java

```
interface Cantaor{  
    void canta();  
}
```



Jubilado.java

```
class Jubilado implements Cantaor {  
    public void canta() {  
        System.out.println("Carraspea");  
        System.out.println("Soy minero...");  
        System.out.println("Esquiva tomates");  
    }  
}
```



Catedrático.java

```
class Catedrático implements Cantaor {  
    public void canta() {  
        System.out.println("Carraspea");  
        System.out.println("Reloj no marques las horas...");  
        System.out.println("Esquiva tomates");  
    }  
}
```



Cantaor.java

```
interface Cantaor{  
    void canta();  
}
```



Jubilado.java

```
class Jubilado implements Cantaor {  
    public void canta() {  
        System.out.println("Carraspea");  
        System.out.println("Soy minero...");  
        System.out.println("Esquiva tomates");  
    }  
}
```



Catedrático.java

```
class Catedrático implements Cantaor {  
    public void canta() {  
        System.out.println("Carraspea");  
        System.out.println("Reloj no marques las horas...");  
        System.out.println("Esquiva tomates");  
    }  
}
```

¿Cómo podemos hacerlo sin tener que copiar y pegar código en las clases hijas?



AbstractCantaor.java

```
abstract class AbstractCantaor implements Cantaor {  
    public void canta() {  
        System.out.println("Carraspea");  
        cuandoQuieraMaestro();  
        System.out.println("Esquiva tomates");  
    }  
    protected abstract void cuandoQuieraMaestro();  
}
```



Jubilado.java

```
class Jubilado extends AbstractCantaor {  
    void cuandoQuieraMaestro() {  
        System.out.println("Soy mineroooo000ooo... ");  
    }  
}
```



Catedrático.java

```
class Catedrático extends AbstractCantaor {  
    void cuandoQuieraMaestro() {  
        System.out.println("Reloj, no marques las horas...");  
    }  
}
```



AbstractCantaor.java

```
abstract class AbstractCantaor implements Cantaor {  
    public void canta() {  
        System.out.println("Carraspea");  
        cuandoQuieraMaestro();  
        System.out.println("Esquiva tomates");  
    }  
    protected abstract void cuandoQuieraMaestro();
```



Jubilado.java

```
class Jubilado extends AbstractCantaor {  
    void cuandoQuieraMaestro() {  
        System.out.println(" Soy minerooo000ooo... ");  
    }  
}
```



Catedrático.java

```
class Catedrático extends AbstractCantaor {  
    void cuandoQuieraMaestro() {  
        System.out.println("Reloj, no marques las horas...");  
    }  
}
```



AbstractCantaor.java

```
abstract class AbstractCantaor implements Cantaor {  
    public void canta() {  
        System.out.println("Carraspea");  
        cuandoQuieraMaestro();  
        System.out.println("Esquiva tomates");  
    }  
    protected abstract void cuandoQuieraMaestro();  
}
```



```
class Jubilado extends AbstractCantaor {  
    void cuandoQuieraMaestro() {  
        System.out.println(" Soy minerooo000ooo... ");  
    }  
}
```



Catedrático.java

```
class Catedrático extends AbstractCantaor {  
    void cuandoQuieraMaestro() {  
        System.out.println("Reloj, no marques las horas...");  
    }  
}
```

**Se separa la parte común y
la parte específica de la
implementación.**

La clase base contendrá la parte
obligatoria.

**Se separa la parte común y
la parte específica de la
implementación.**

Las clases derivadas aportarán la
parte específica.

¡Para esto es el protected!

Indica que hay que completar una implementación parcial.

Extensión

Sabemos que en una clase derivada podemos añadir nuevos métodos que no estén en la clase base.



```
interface Adepto {...}

class StandardAdepto implements Adepto {...}

class Catedrático extends StandardAdepto {
    // ...
    public void hacerMuchosMuchosArtículos() {
        // ...
    }
}
```

La ventaja de la orientación a objetos es que para crear una clase se puede escoger otra que ya esté hecha y añadirle las operaciones que le faltan.

¿Es eso cierto?

¿Es eso cierto?

En la práctica, no mucho.

Las clases, en una jerarquía, lo normal es que sean implementaciones de una determinada interfaz.

Normalmente los clientes interactuarán con los objetos a través de dicha interfaz.

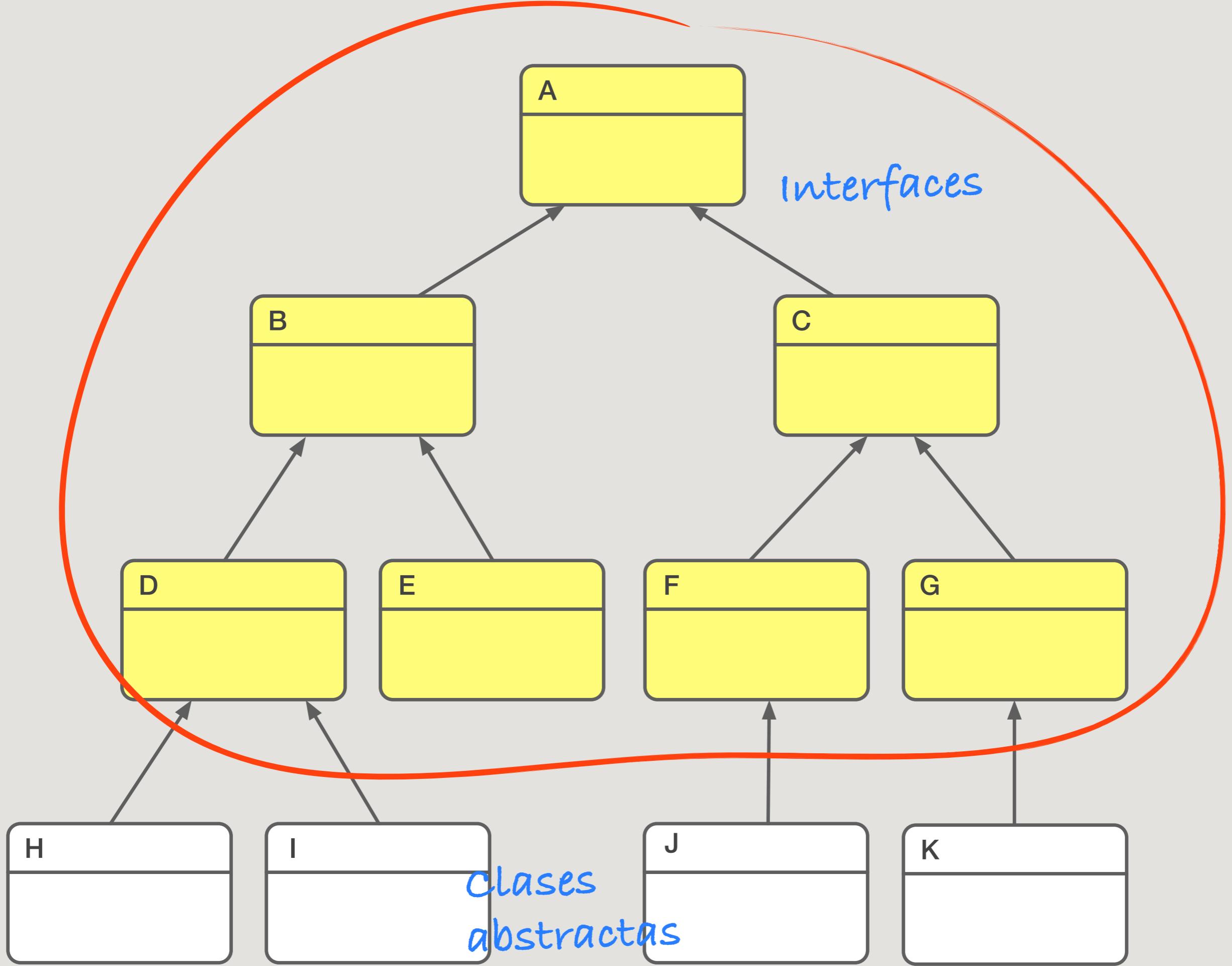
Ni la interfaz tiene los métodos añadidos ni el cliente sabe de qué tipo concreto es el objeto.

**Y esa sí es la grandísima
ventaja de la OO:**

Poder sustituir unos objetos por
otros.

Jerarquías de interfaces

O de tipos



Cada paradigma dirige a una forma particular de plasmar el diseño.

... bla, bla, bla, empleados, bla, bla, empresa, bla, bla...

Cada paradigma dirige a una forma particular de plasmar el diseño.

... bla, bla, bla, empleados, bla, bla, empresa, bla, bla...

Entidad-relación, funcional, orientado a objetos...

Cada paradigma dirige a una forma particular de plasmar el diseño.

... bla, bla, bla, empleados, bla, bla, empresa, bla, bla...

Se extraen las entidades.

Cada paradigma dirige a una forma particular de plasmar el diseño.

... bla, bla, bla, empleados, bla, bla, empresa, bla, bla...

Se extraen las entidades.

Cada paradigma dirige a una forma particular de plasmar el diseño.

... bla, bla, bla, empleados, bla, bla, empresa, bla, bla...

Se extraen las entidades.

Cada paradigma dirige a una forma particular de plasmar el diseño.

... bla, bla, bla, empleados, bla, bla, empresa, bla, bla...

Se buscan las relaciones entre ellas (herencia, agregación...): «es un», «tiene un», «parece casi como un» ...

... bla, bla, ventana, bla, bla, rectángulo, bla, bla, bla...

... bla, bla, bla, círculo, bla, bla, bla, punto, bla, bla...

¿Es eso correcto para llegar a un
buen diseño?

Ya hemos visto, a propósito del ES-UN, que los
métodos tradicionales no sirven.

El círculo es un punto al que se le añade un radio (herencia).

El círculo tiene un punto central y un radio (agregación).

¿La ventana es un rectángulo?

¿El rectángulo hereda de cuadrado o es al revés?

¿El rectángulo es una extensión del cuadrado al que se le permite ampliarse en ambas direcciones?

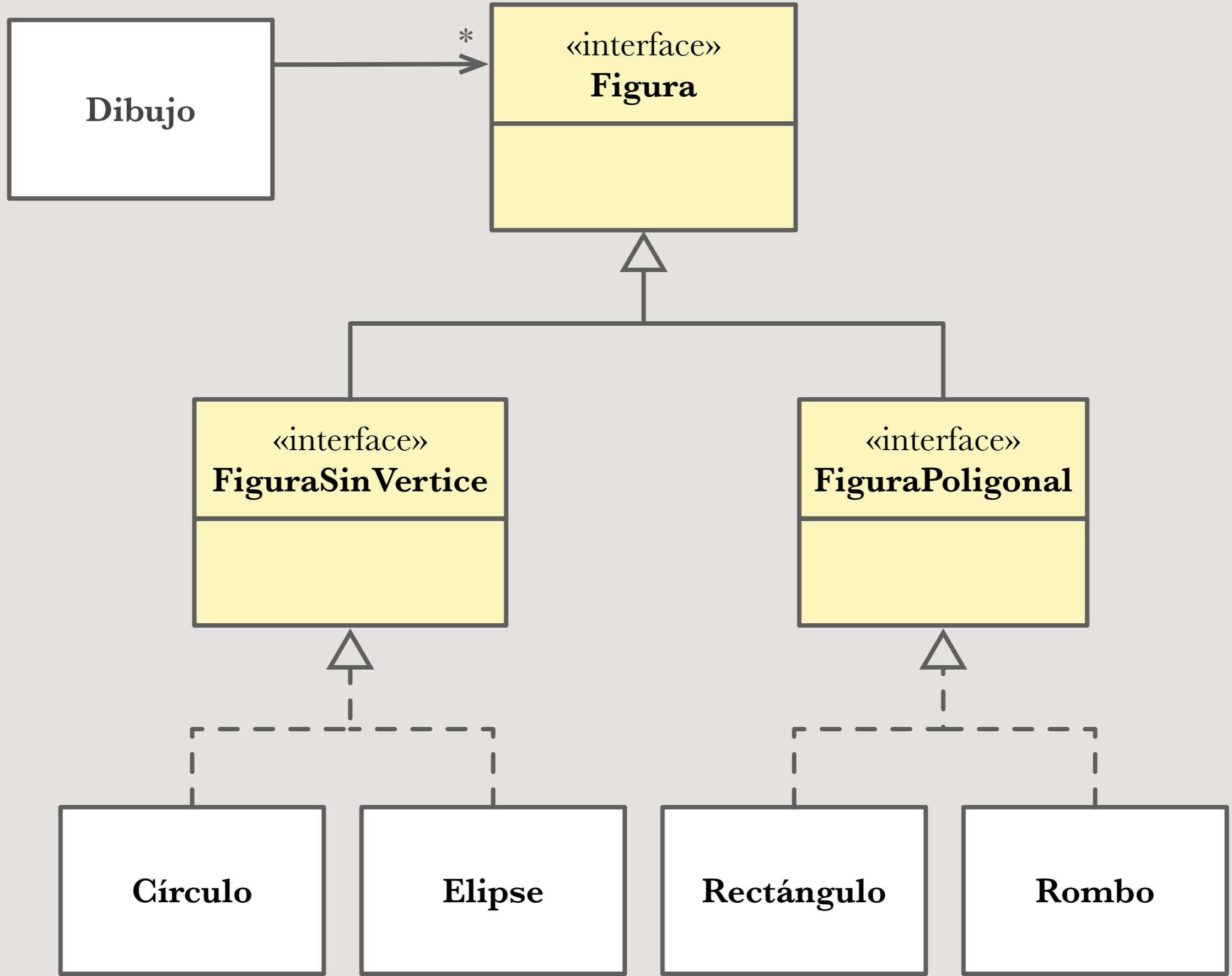
¿El cuadrado es un rectángulo en el que alto y ancho son iguales?

¿Qué fue primero, el huevo o la gallina?

Si no la siento, ¿debo fingir la herencia?

Ejemplo

¿Es correcto el siguiente diseño?



Recordemos

Las interfaces surgen por la
necesidad de declarar unas
responsabilidades.

Por tanto

Nunca debe surgir una jerarquía
como una mera clasificación.

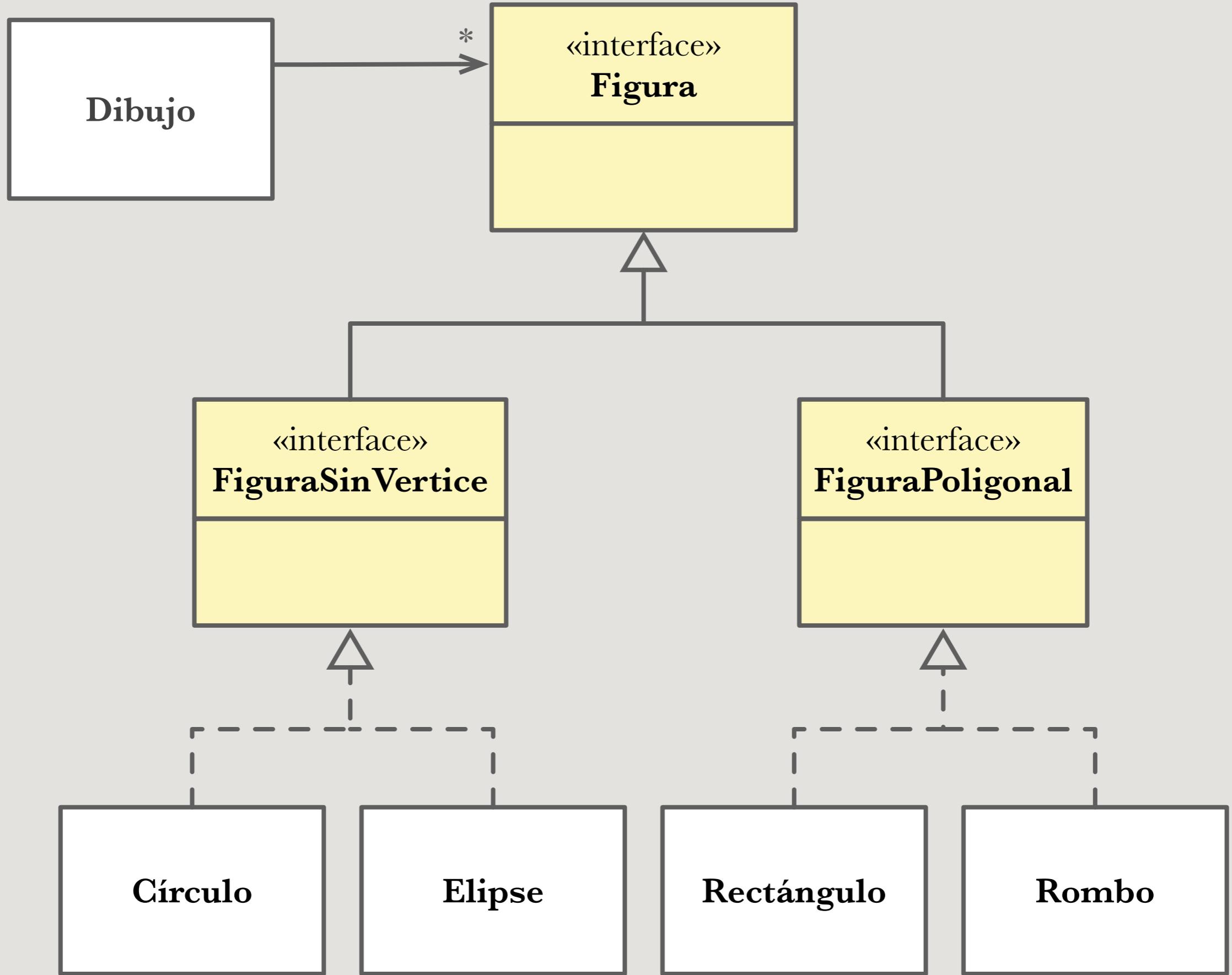
Por tanto

No se trata de buscar un árbol
equilibrado.

Por tanto

Tan solo hay que comprobar si
toda interfaz tiene al menos un
cliente que la use.

Volviendo a nuestro ejemplo...



¿Es correcto?

Depende.

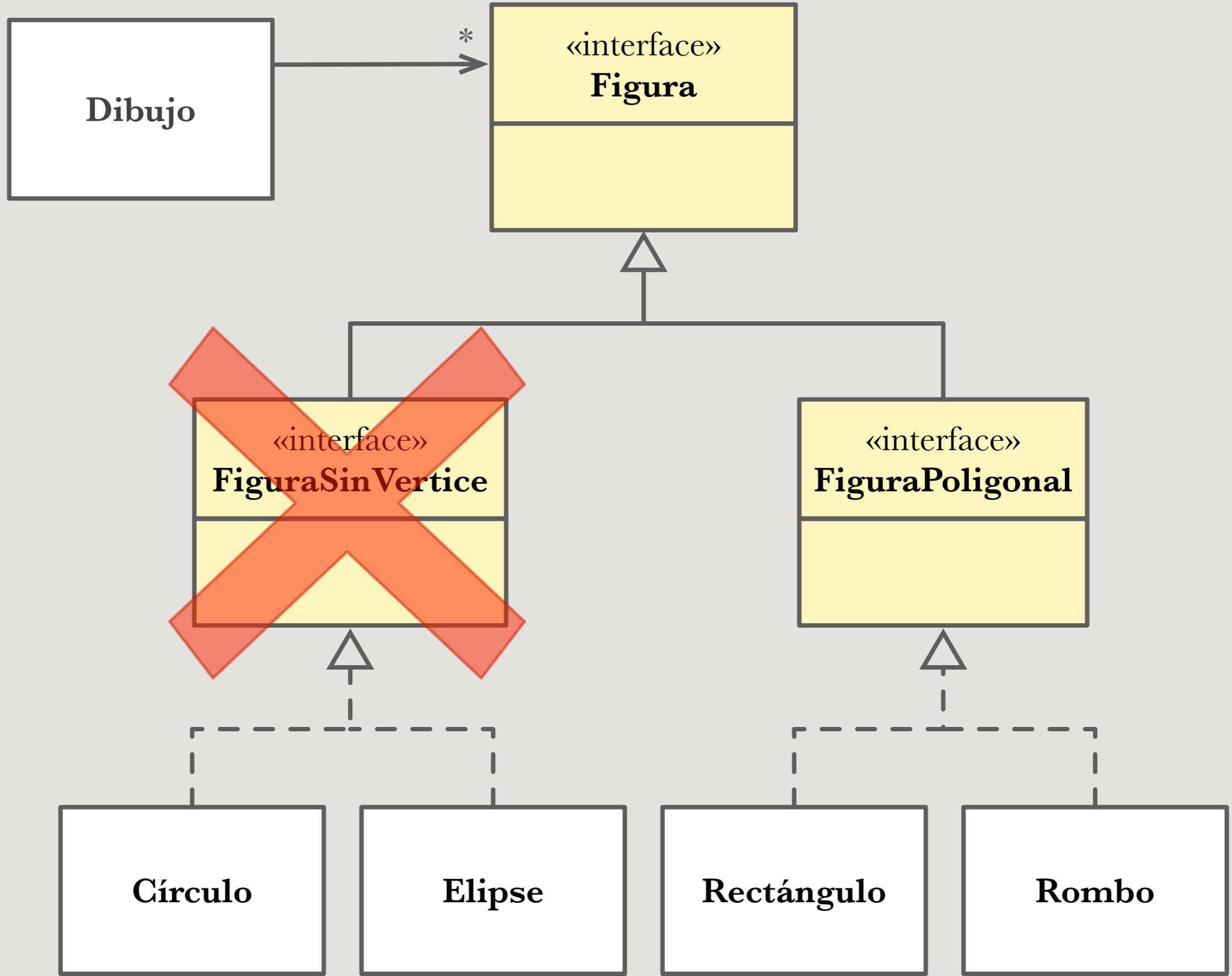
Las interfaces FiguraSinVértice y FiguraPoligonal tendrían sentido solo si va a haber algún cliente que las use (que necesite tratar a los objetos de uno u otro tipo polimórficamente).

Por ejemplo, si hay una HerramientaX que trabaje solo con figuras poligonales.



```
class Dibujo {  
    private List<Figura> figuras;  
    // ...  
    dibujar() {  
        for (Figura figura : figuras) {  
            figura.dibujar();  
        }  
    }  
  
class HerramientaX {  
    // ...  
    manipula(FiguraPoligonal polígono) {  
        List<Punto> puntos = polígono.getPuntos();  
        // ...  
    }  
}
```

Pero que en el ejemplo anterior FiguraPoligonal tenga plenamente sentido no significa que haya que introducir su equivalente FiguraSinVértice para buscar un árbol equilibrado.



No hace falta tener jerarquías para que un diseño sea orientado a objetos.

¡Aunque sea lo primero (y, a veces, lo único) que se enseña de la orientación a objetos!

No es que no haga falta, es que está mal comenzar un diseño orientado a objetos haciendo jerarquías (aunque sea lo más fácil).

Cuando se tiene un objeto que necesite delegar en otros una cierta responsabilidad se saca esta a una interfaz de manera que otros la puedan implementar.

A partir de esta necesidad se va creando una jerarquía de objetos (generalmente de un solo nivel) que la solucionan de distinta manera (esto es, como alternativas de implementación).

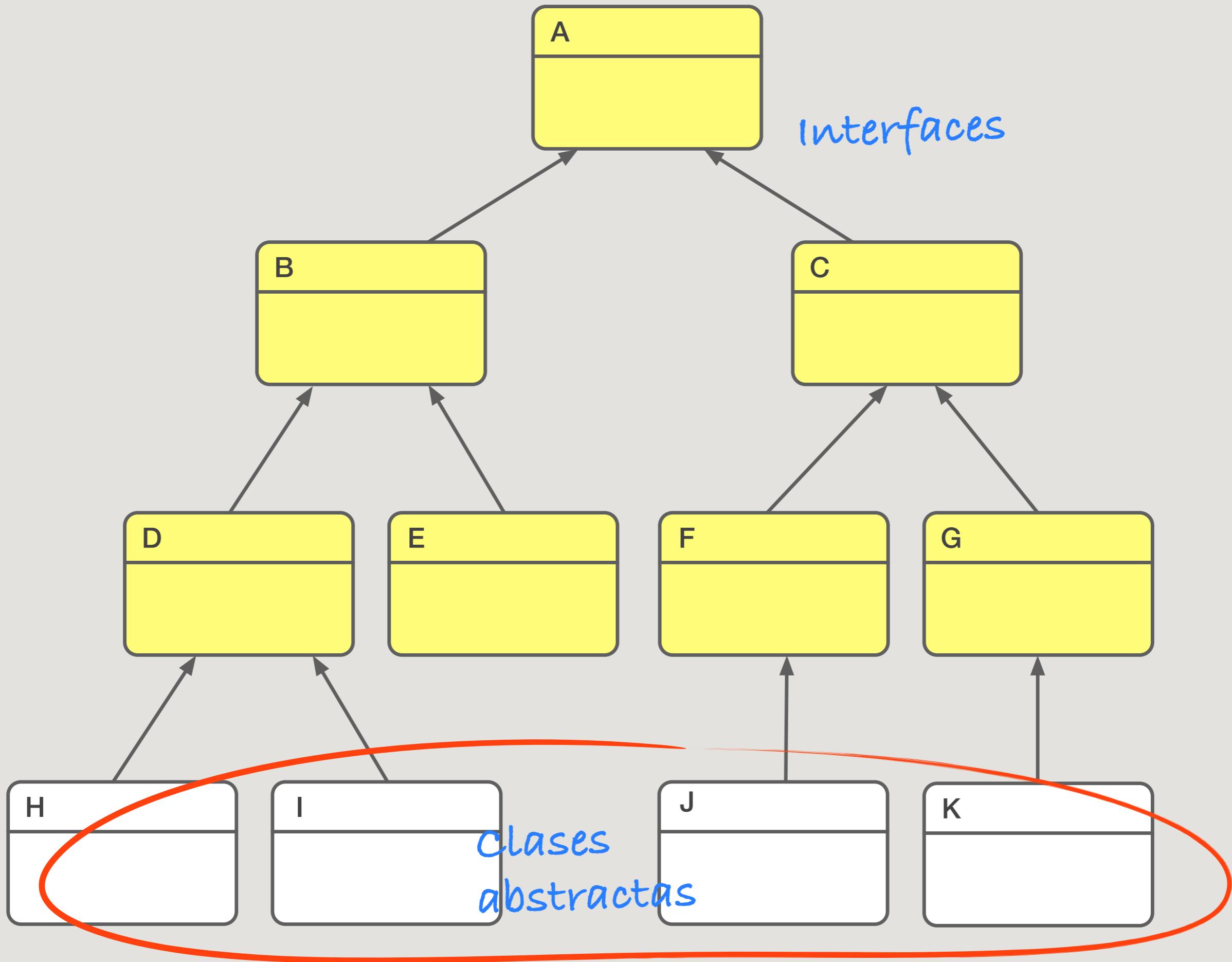
Si no hay ningún cliente que haya planteado dicha necesidad inicial, la jerarquía carece de todo sentido.

**SÍN NECESIDAD
NO INTERFAZ!!**



**¡Nunca debe
surgir una
jerarquía como
una mera
clasificación!**

Jerarquías de clases abstractas



Recordemos

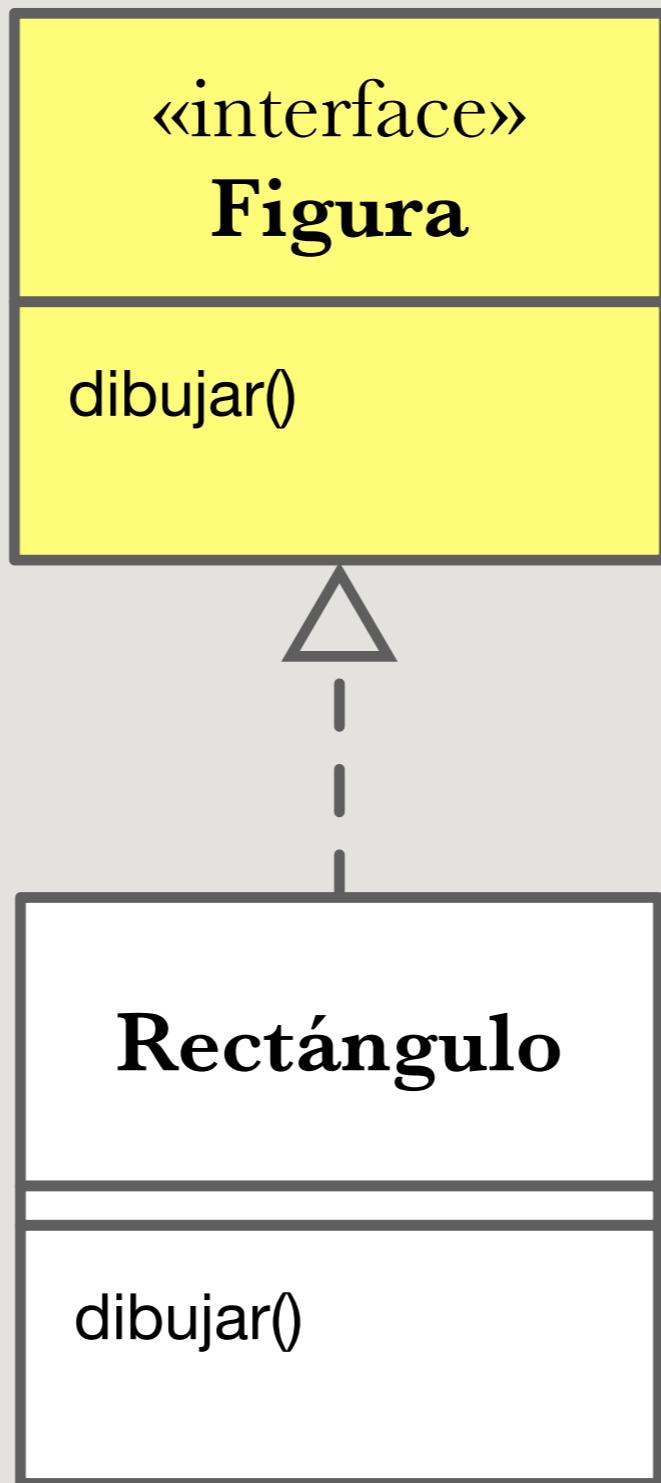
¿De dónde surgen las clases abstractas?

Del implementador de la interfaz.

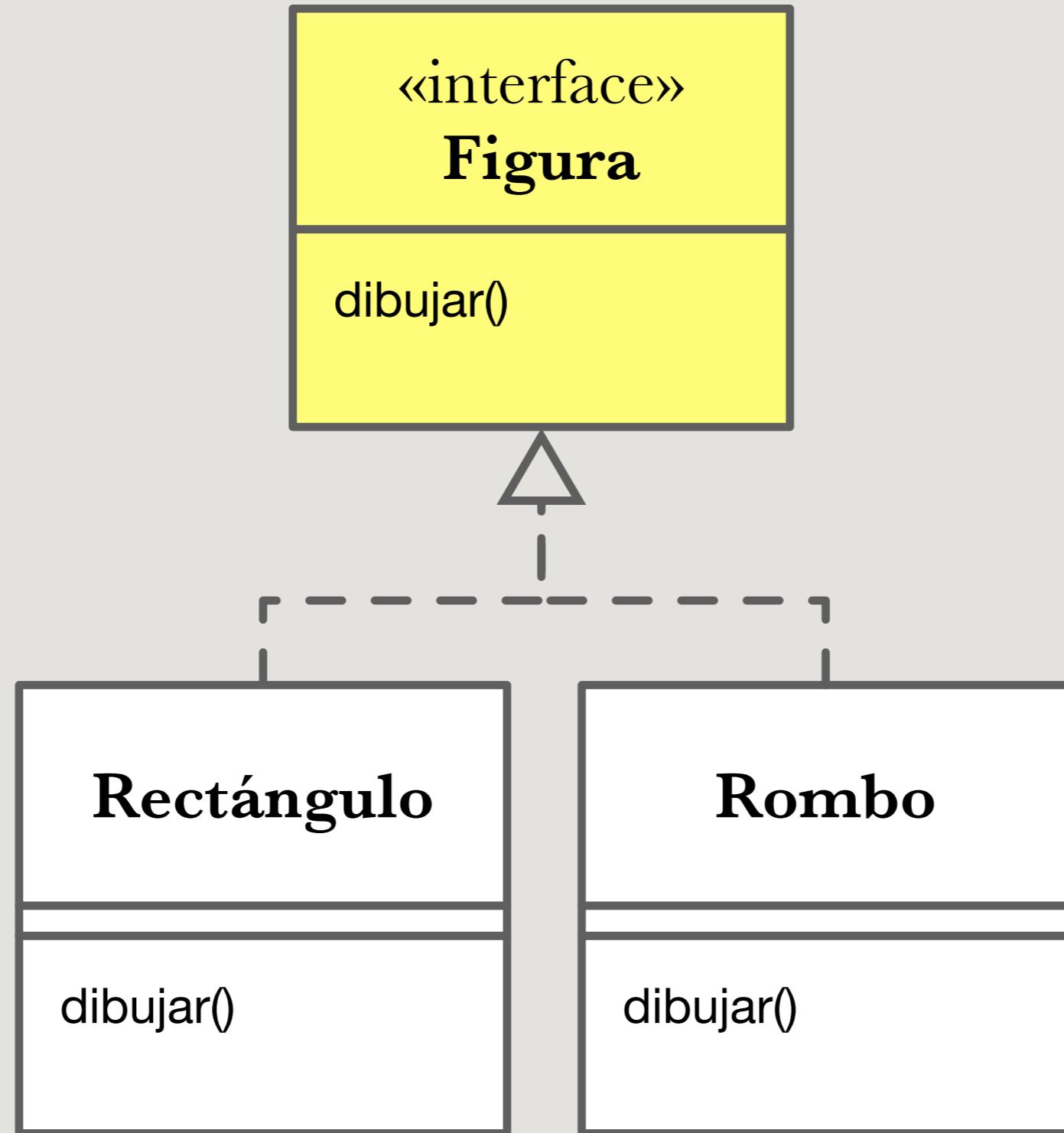
Del implementador de las clases concretas, al factorizar.

Ejemplo

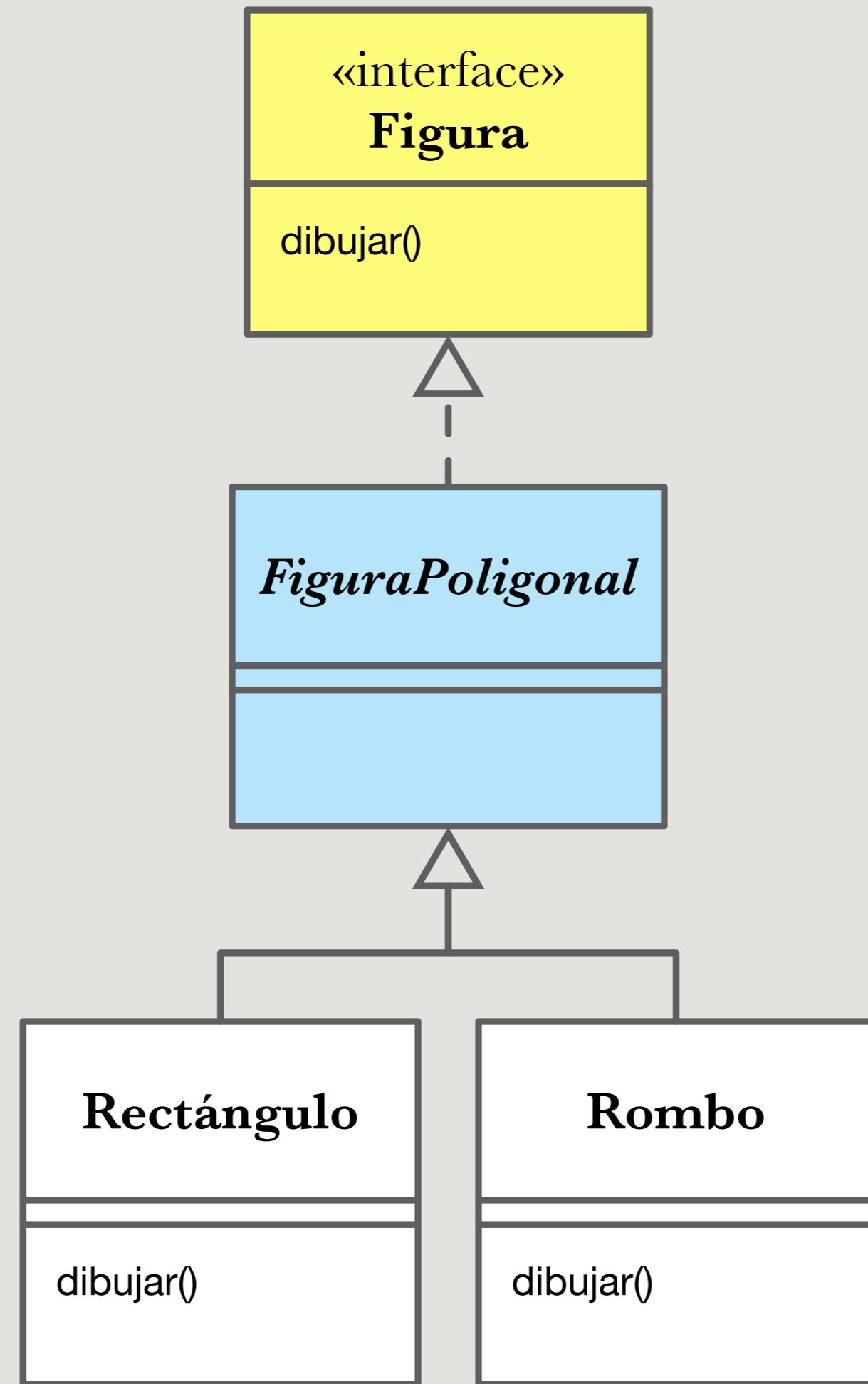
Inicialmente tenemos una interfaz (se supone que con al menos algún cliente que la está usando) y una clase concreta que la implementa.



Ahora aparece una nueva clase y nos damos cuenta (es un decir) de que tienen código duplicado.



Movemos dicha funcionalidad común hacia arriba en la jerarquía, creando una nueva clase abstracta.



Pero...

¿El introducir una figura debería
cambiar otras ya existentes?

Por otro lado, al factorizar evitamos la duplicación de código.

¿Código duplicado o propagar
los cambios?

**Una jerarquía de
implementación (de
clases abstractas) no es
esencial que sea estable.**

El cliente solo conoce la interfaz.

No sabe si debajo de esa ella hay una jerarquía de clases abstractas para ahorrar código o clases concretas que implementan directamente la interfaz: no trata directamente con ellas.

No le afectan, por tanto, las reorganizaciones que se hagan de dichas implementaciones.

**Las jerarquías de
implementación son para
compartir código común y
pueden reestructurarse
cuantas veces se necesite.**

Si más adelante desaparece Rombo fusionaremos FiguraPoligonal y Rectángulo.

No hace falta pensar en el futuro lejano: reorganizaremos el árbol en función de las necesidades actuales.

¿Cuál es el papel de una clase abstracta en un diagrama de análisis o de diseño preliminar?

En el análisis no importa el código; tampoco en el diseño
(sí los métodos, pero no su implementación interna).

Si lo que importan son las clases e interfaces y sus mensajes (las responsabilidades), pero no las implementaciones...

¿Por qué mostrar clases que surgen
por la implementación?

Conclusiones

¿Tenemos claras las diferencias entre interfaces y clases abstractas?

Repasemos sus diferencias atendiendo a:

En qué momento se crea cada una

¿Quién es su creador en el proceso de desarrollo?

Analista, diseñador o programador

Si pueden ser raíz de una jerarquía o no

Importancia dentro del paradigma

¿Qué consecuencias tendría para el lenguaje de programación quitar cada una de ellas?

Qué hacen con respecto a las operaciones de sus ancestros

Añadir, redefinir, eliminar

Impacto de sus cambios en el resto del diseño

	Interfaz	Clase abstracta
Situación en la que se originan	Extracción de responsabilidades	Facilitar nuevas implementaciones No repetir código
Quién las crea	Analista/Diseñador	Programador
Raíz de la jerarquía	Sí	No
Importancia	Fundamental: es el enlace dinámico, lo que nos permite eliminar lógica condicional	Comodidad (es un mero cortar y pegar)
Operaciones	Añaden a las de sus ancestros	Implementan las de sus ancestros Añaden operaciones de implementaciones parciales
Impacto de los cambios	Alto: es un contrato que afecta a varios programadores	Bajo: la interfaz hace de cortafuegos