



Tecnologías de Programación

Paradigma Orientado a Objetos

IV – Patrones de Diseño



Patrones de Diseño

- Los patrones de diseño (design patterns) son la base para la búsqueda de soluciones a problemas comunes en el desarrollo de software y otros ámbitos referentes al diseño de interacción o interfaces.
- Un patrón de diseño es ***una solución a un problema de diseño***. Para que una solución sea considerada un patrón debe poseer ciertas características. Una de ellas es que debe haber comprobado su efectividad resolviendo problemas similares en ocasiones anteriores. Otra es que debe ser reusable, lo que significa que es aplicable a diferentes problemas de diseño en distintas circunstancias.



Objetivos

- Proporcionar catálogos de elementos reusables en el diseño de sistemas software.
- Evitar la reiteración en la búsqueda de soluciones a problemas ya conocidos y solucionados anteriormente.
- Formalizar un vocabulario común entre diseñadores.
- Estandarizar el modo en que se realiza el diseño.
- Facilitar el aprendizaje de las nuevas generaciones de diseñadores condensando conocimiento ya existente.

No es obligatorio utilizar los patrones, solo es aconsejable en el caso de tener el mismo problema o similar que soluciona el patrón, siempre teniendo en cuenta que en un caso particular puede no ser aplicable.

Abusar o forzar el uso de los patrones puede ser un error.



Patrones de Diseño

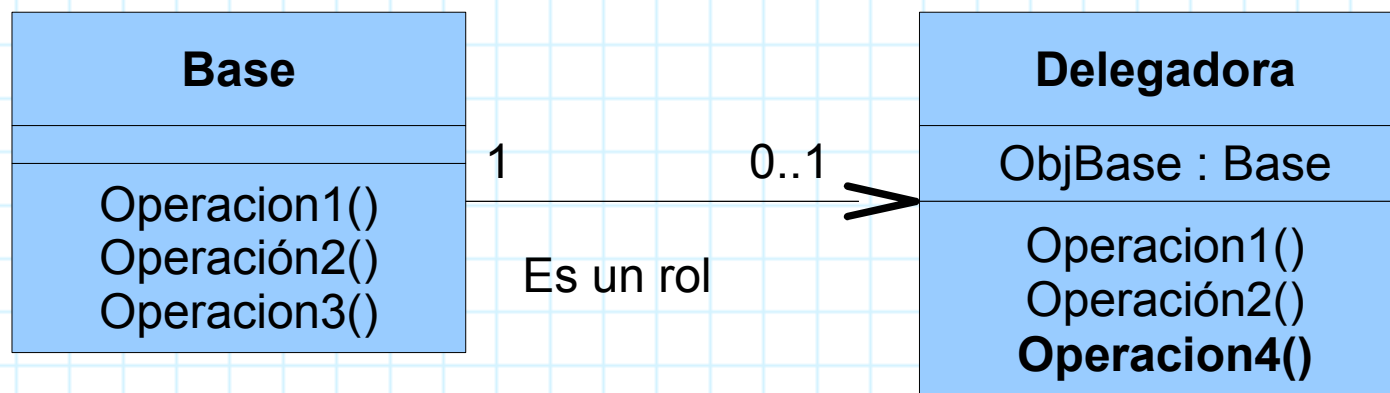
- Delegator
- Composite
- Adapter
- Decorator
- Observer
- State
- Strategy

Delegación I

Este es un patrón fundamental de tipo estructural. **Indica cuándo no usar herencia.**

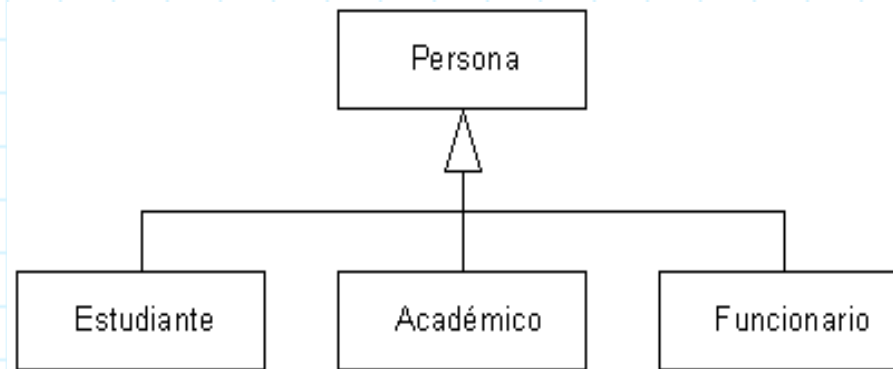
La herencia es útil para modelar relaciones de tipo **es-un** o **es-una**, ya que estos tipos de relaciones son de naturaleza estática.

Sin embargo, relaciones de tipo **es-un-rol-ejecutado-por** son mal modeladas con herencia. En este tipo de relaciones, instancias de una clase pueden jugar múltiples roles.



Delegación II

Por ejemplo, supongamos que tenemos el caso de una universidad donde se tienen tres tipos de roles: estudiantes, académicos y funcionarios. Es posible representar esta situación mediante una clase llamada **Persona** que tiene las subclases correspondientes a cada uno de los roles.

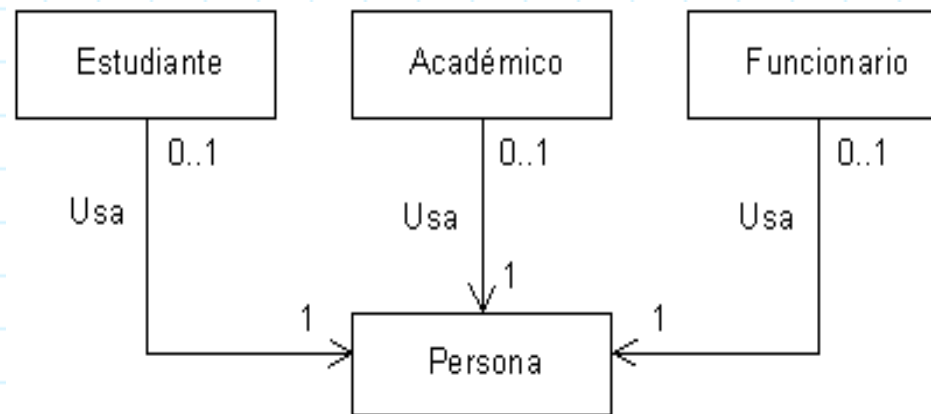


El problema con este diagrama, es que una misma persona puede jugar más de un rol al mismo tiempo.

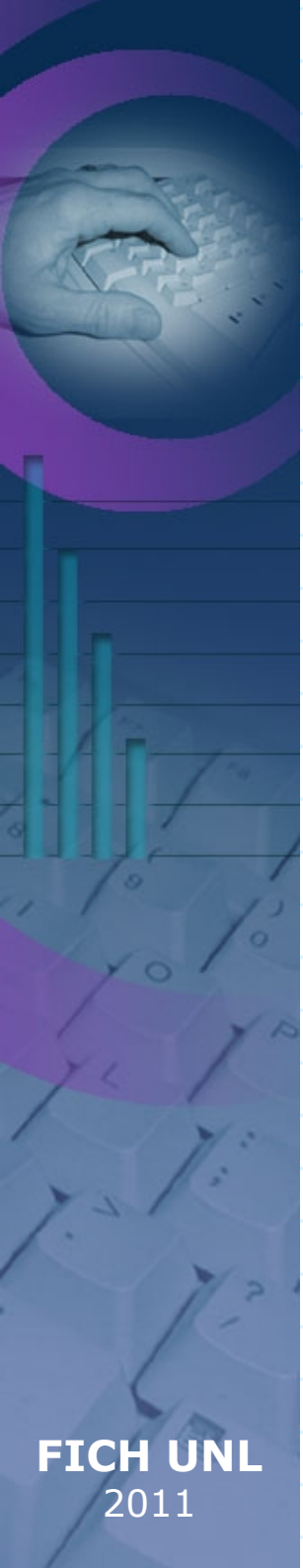
La herencia es una relación estática que no cambia con el tiempo.

Delegación III

Para resolver esta situación, es posible representar personas en diferentes roles usando *delegación*, como se muestra en la siguiente figura.



La solución general propuesta en este patrón es: incorporar la funcionalidad de la clase original usando una instancia de la clase original y llamando sus métodos.



```
import java.util.Date;

public class Principal {

    public static void main(String[] args) {

        Persona oPersonal = new Persona( "Juan",
                                           "Perez",
                                           new Date("1976/03/02"),
                                           "1 de mayo");

        System.out.println("Edad:"  + oPersonal.calcularEdad());

        Estudiante oEstudiante1 = new Estudiante(oPersonal, "Informatica");
        System.out.println( oEstudiante1.listar() +
                            " Edad:"  +
                            oEstudiante1.calcularEdad());

        Academico oAcademico1 = new Academico(oPersonal, "TECPROG");
        System.out.println( oAcademico1.listar() +
                            " Edad:"  +
                            oAcademico1.calcularEdad());

    }

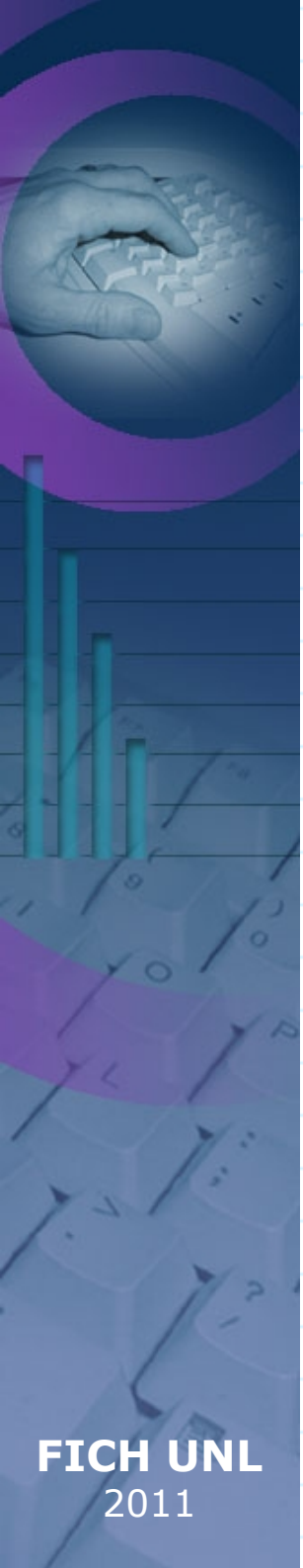
}
```

CONSOLA

Edad:37

Estudiante: Perez, Juan - Rol estudiante: Informatica Edad:37

Academico: Perez, Juan - Rol docente: TECPROG Edad:37



```
import java.util.Date;

public class Persona {

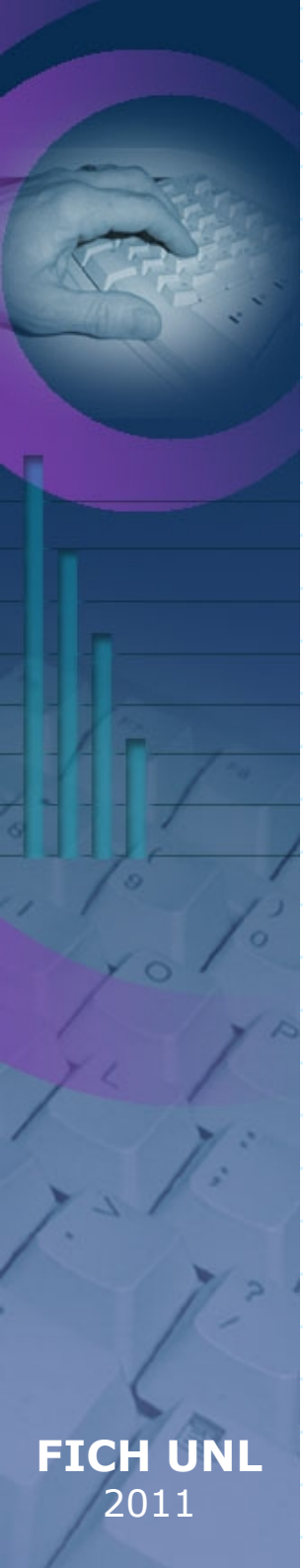
    private String nombre;
    private String apellido;
    private Date fecha_nacimiento;
    private String domicilio;

    public Persona( String nombre,
                    String apellido,
                    Date fecha_nacimiento,
                    String domicilio) {
        this.nombre = nombre;
        this.apellido = apellido;
        this.fecha_nacimiento = fecha_nacimiento;
        this.domicilio = domicilio;
    }

    public String listar() {
        return this.apellido + ", " + this.nombre;
    }

    public int calcularEdad() {
        Date hoy = new Date();
        long diferencia= ( hoy.getTime() - this.fecha_nacimiento.getTime());
        int anios = ((int) (diferencia/(1000*60*60*24))) / 365;

        return anios;
    }
}
```



```
public class Estudiante {  
  
    private Persona oPersona;  
    private String carrera;  
  
    public Estudiante(Persona oPersona, String carrera) {  
        this.oPersona = oPersona;  
        this.carrera = carrera;  
    }  
  
    public int calcularEdad() {  
        return this.oPersona.calcularEdad();  
    }  
  
    public String listar() {  
        return this.oPersona.listar() +  
            " - Rol estudiante: " +  
            this.carrera;  
    }  
  
}
```

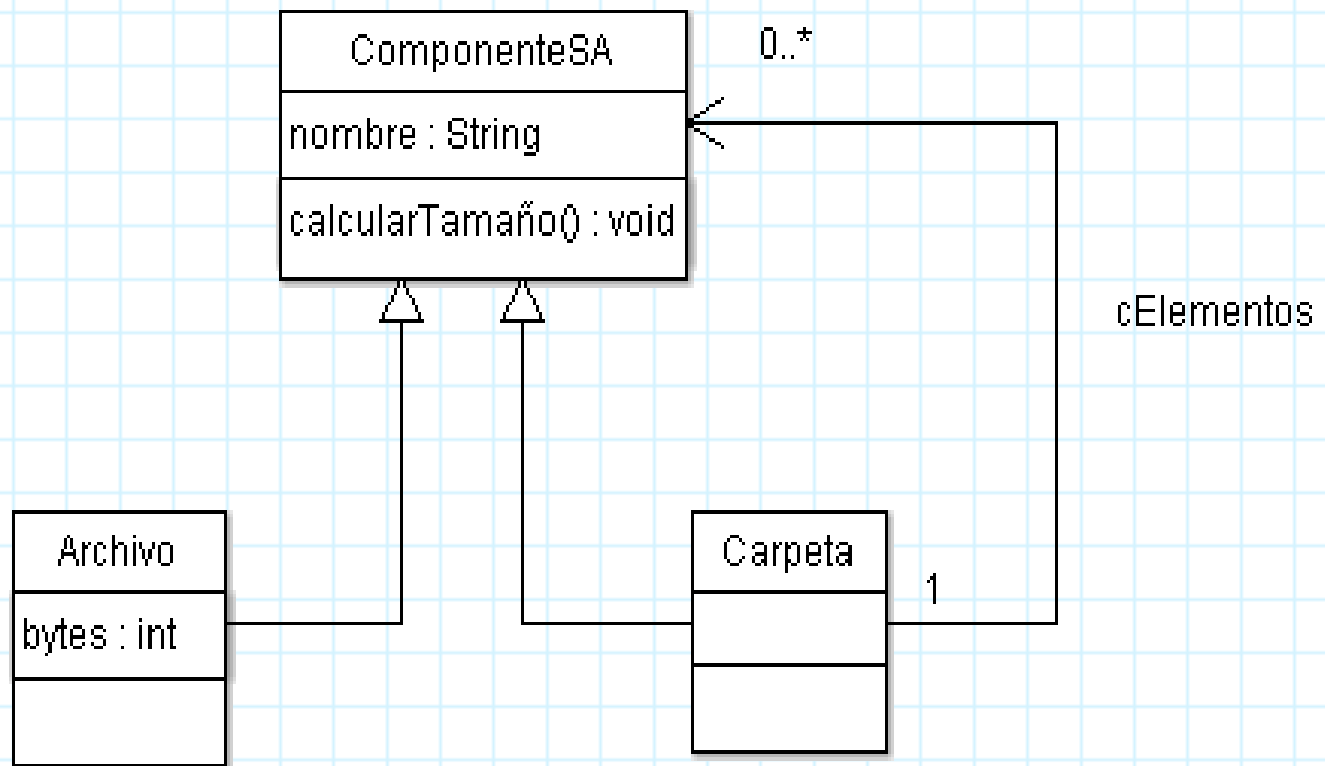


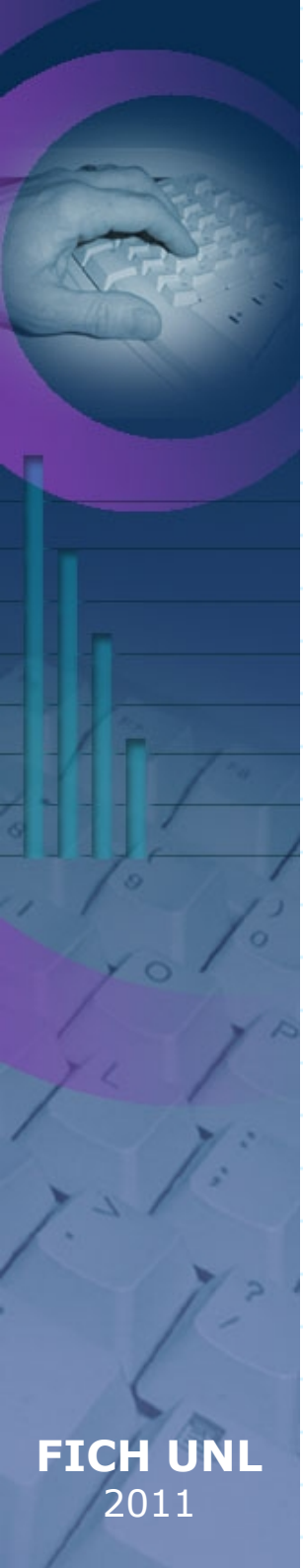
Composite I

- El patrón **Composite** sirve para construir objetos complejos a partir de otros más simples y similares entre sí, gracias a la composición recursiva y a una estructura en forma de árbol.
- Esto simplifica el tratamiento de los objetos creados, ya que al poseer todos ellos una interfaz común, se tratan todos de la misma manera.

Composite II

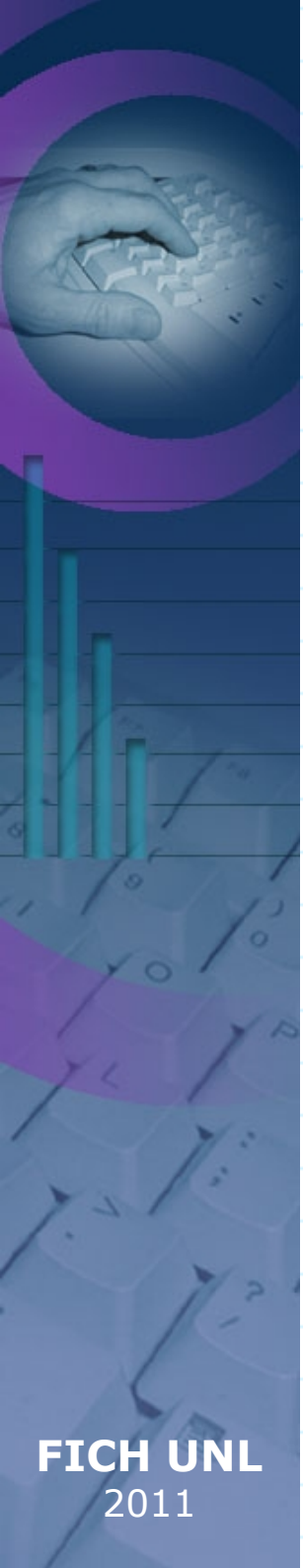
Imaginemos que necesitamos crear una serie de clases para guardar información de un Sistema de Archivos.





```
public abstract class ComponenteSA {  
  
    private String nombre;  
  
    public ComponenteSA(String nombre) {  
        this.nombre = nombre;  
    }  
  
    public abstract int calcularTamaño();  
  
    public String toString() {  
        return this.nombre;  
    }  
  
    public abstract void agregar(ComponenteSA oComponente);  
    public abstract void remover(ComponenteSA oComponente);  
}
```

```
public class Archivo extends ComponenteSA {  
  
    private int bytes;  
  
    public Archivo(String nombre, int bytes) {  
        super(nombre);  
        this.bytes = bytes;  
    }  
  
    public int calcularTamaño() {  
        return this.bytes;  
    }  
  
    public void agregar(ComponenteSA c) {}  
    public void remover(ComponenteSA c) {}  
}
```



```
import java.util.Vector;

public class Carpeta extends ComponenteSA {

    private Vector<ComponenteSA> cElementos;

    public Carpeta(String nombre) {
        super(nombre);
        this.cElementos = new Vector<ComponenteSA>();
    }

    public int calcularTamaño() {

        // Recorre la colección de elementos que lo componen
        // y llama a calcularTamaño de cada uno
        int total = 0;
        for(ComponenteSA oComponenteSA: this.cElementos) {
            total += oComponenteSA.calcularTamaño();
        }

        return total;
    }

    public void agregar(ComponenteSA componente) {
        cElementos.add(componente);
    }

    public void remover(ComponenteSA componente) {
        cElementos.remove(componente);
    }

}
```