

# METODOLOGÍAS DE PROGRAMACIÓN I

---

Patrón de comportamiento *Chain of responsibility*

# Situación de ejemplo

- Una empresa organizadora de eventos posee un equipo de soporte capaces de solucionar cualquier inconveniente que ocurra durante el desarrollo del evento.
- Este equipo es capaz de solucionar problemas de sonido, de luces, de electricidad en general, gasistas para la cocina, carpinteros para reparar mesas y sillas, costureras para arreglar vestidos o camisas, etc.
- Cada problema es resuelto por una persona experta en ese tipo de problemas.

# Situación de ejemplo

Esta empresa cuenta con la siguiente clase:

```
class Evento
    Electricista electricista
    Carpintero carpintero
    Costurera costurera
    Gasista gasista
    Sonidista sonidista

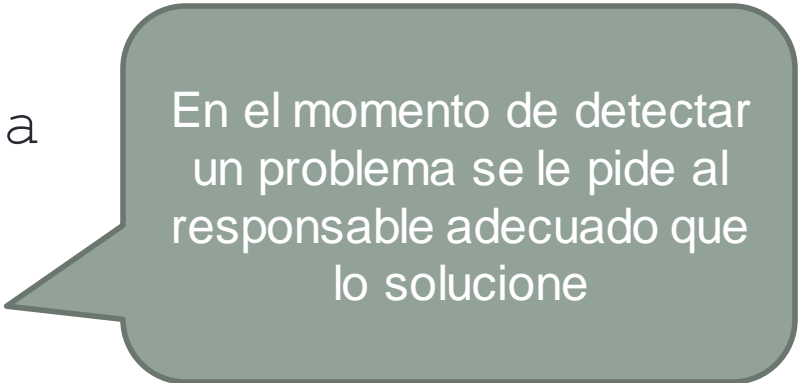
    void hacerFiesta()
        . . .
```

# Situación de ejemplo

Esta empresa cuenta con la siguiente clase:

```
class Evento
    Electricista electricista
    Carpintero carpintero
    Costurera costurera
    Gasista gasista
    Sonidista sonidista

    void hacerFiesta()
    . . .
```




En el momento de detectar un problema se le pide al responsable adecuado que lo solucione

# Problema

- ¿Qué sucede si ahora la empresa cuenta con médicos y bomberos como parte del equipo?
- ¿Qué sucedería si contratamos un bombero capaz de solucionar problemas de electricidad o de pérdidas de gas? Es decir, un bombero capaz de actuar como electricista o gasista.
- Nuestra clase *Evento* se vuelve muy dependiente de muchos otros objetos y este tipo de situación no es deseable ya que no es fácil de mantener objetos con alto grado de acoplamiento.

# Motivación

Es deseable tener un único responsable capaz de solucionar todos los problemas, pero al mismo tiempo, dejando que cada problema puntual lo solucione el responsable adecuado.



**El patrón Chain of  
Responsability  
permite resolver  
este problema**

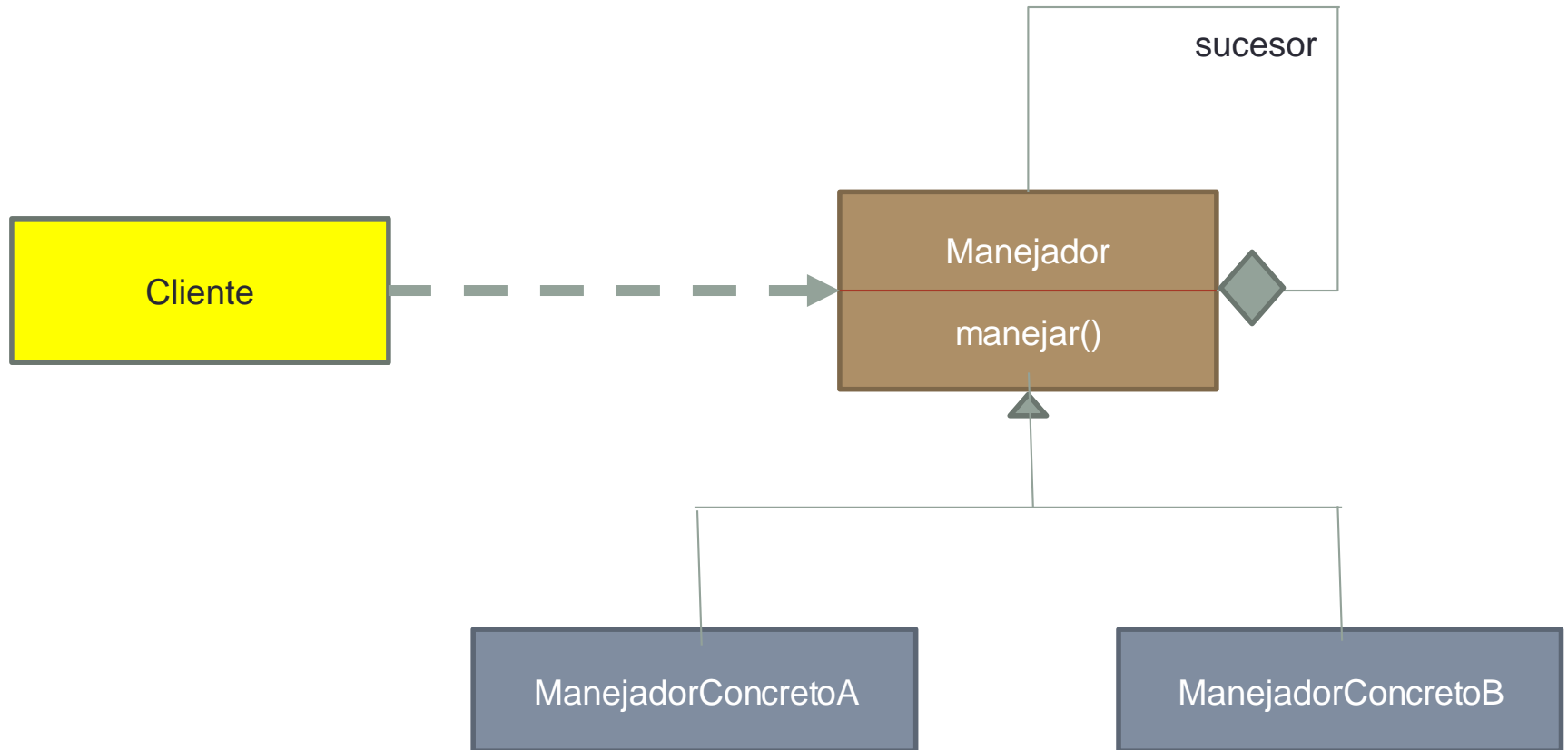
# Chain of responsibility

**Propósito:** Evita acoplar el emisor de una petición a su receptor, dando a más de un objeto la responsabilidad de responder a la petición. Encadena los objetos receptores y pasa la petición a través de la cadena hasta que es procesada por un objeto.

**Aplicabilidad:** usarlo cuando

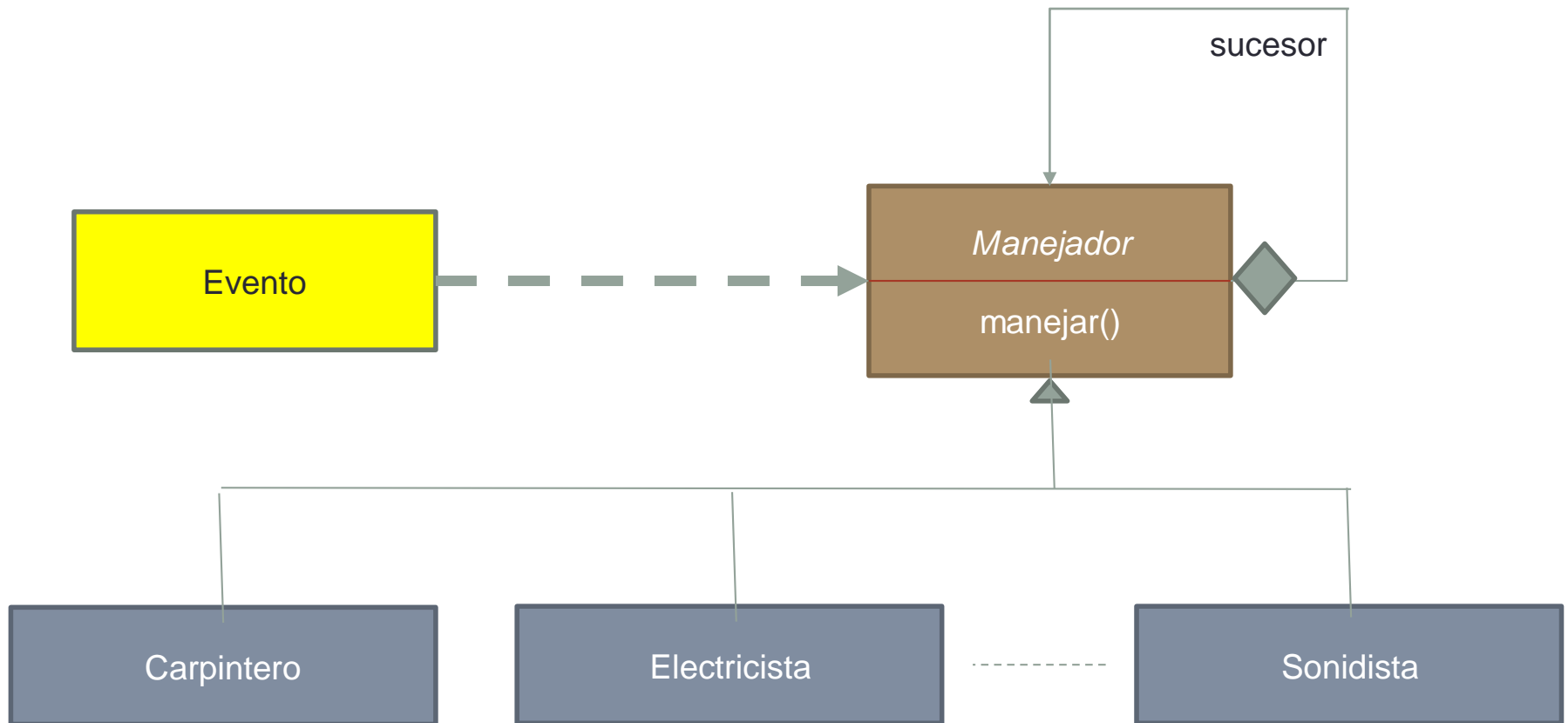
- Hay más de un objeto que puede manejar una petición, y el manejador no se conoce a priori, sino que debería determinarse dinámicamente.
- Se quiere enviar una petición a un objeto entre varios sin especificar explícitamente el receptor.

# Chain of responsibility - Estructura





# Chain of responsibility - Estructura

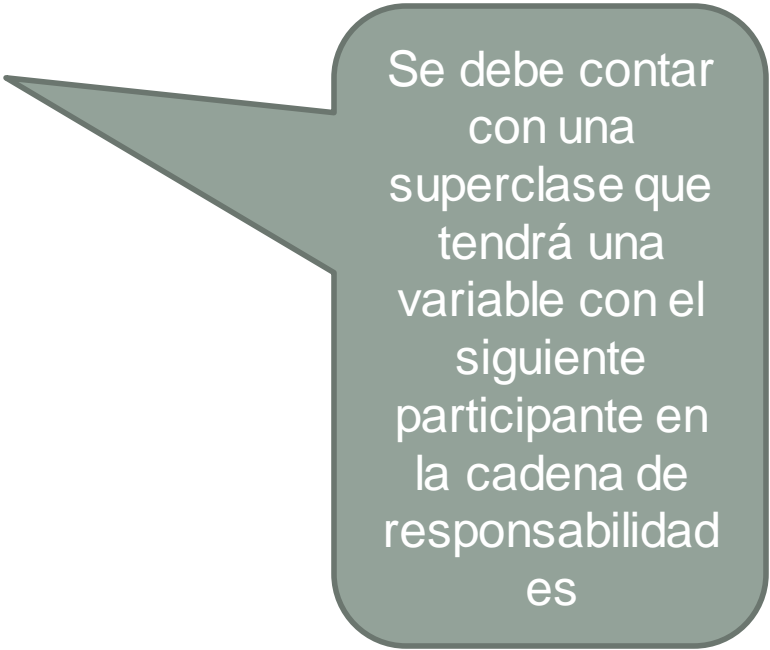


# Chain of responsibility - Ejemplo



# Chain of responsibility - Implementación

```
abstract class Manejador  
  
    Manejador sucesor = null  
  
    constructor (Manejador s)  
        sucesor = s  
  
    . . .
```



Se debe contar con una superclase que tendrá una variable con el siguiente participante en la cadena de responsabilidad es

# Chain of responsibility - Implementación

```
abstract class Manejador
{
    . . .
    void arreglarLuces() . . .
    void corregirSonido() . . .
    void arreglarMesa() . . .
    void encolarSilla() . . .
    void cocerCamisa() . . .
    void apagarFocoDeIncendio() . . .
    void atenderUnDesmayo() . . .
    void arreglarPerdidaDeGas() . . .
}
```

Necesitamos que la superclase abstracta *Manejador* tenga la interface de todos los responsables

# Chain of responsibility - Implementación

```
abstract class Manejador
```

```
    . . .
```

```
void arreglarLuces() . . .
```

```
void corregirSonido() . . .
```

```
void arreglarMesa() . . .
```

```
void encolarSilla() . . .
```

```
void cocerCamisa() . . .
```

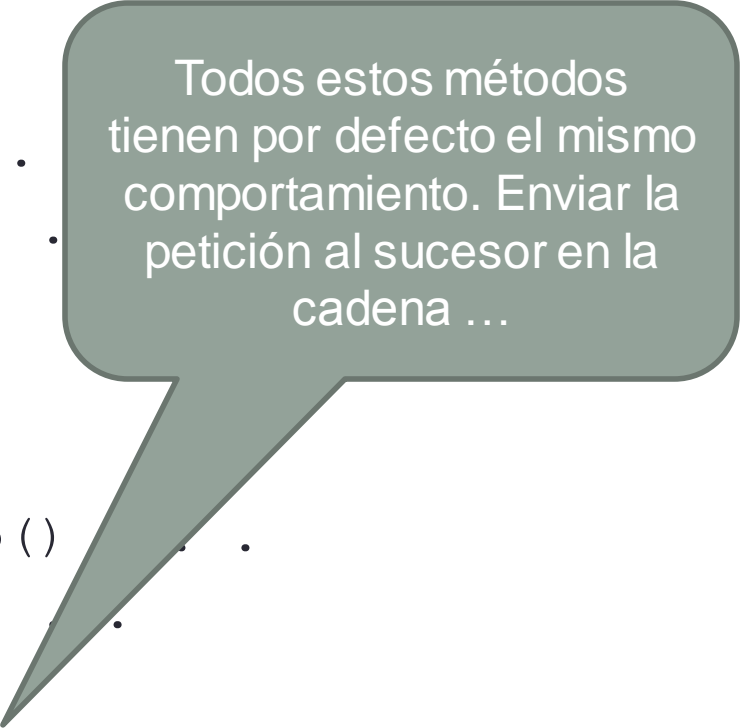
```
void apagarFocoDeIncendio() . . .
```

```
void atenderUnDesmayo() . . .
```

```
void arreglarPerdidaDeGas()
```

```
    if(sucesor != null)
```

```
        sucesor.arreglarPerdidaDeGas()
```



Todos estos métodos tienen por defecto el mismo comportamiento. Enviar la petición al sucesor en la cadena ...

# Chain of responsibility - Implementación

```
class Carpintero : Manejador
```

```
    void arreglarMesa()  
        print("Arreglando la mesa rota")
```

```
    void encolarSilla()  
        print("Encolando la silla rota")
```

# Chain of responsibility - Implementación

```
class Carpintero : Manejador
```

```
    void arreglarMesa()  
        print("Arreglando la mesa rota")
```

```
    void encolarSilla()  
        print("Encolando la silla rota")
```

Cada manejador concreto implementa solo los métodos que sabe manejar.

# Chain of responsibility - Implementación

```
class Carpintero : Manejador
```

```
    void arreglarMesa()  
        print("Arreglando la mesa rota")
```

```
    void encolarSilla()  
        print("Encolando la silla rota")  
        base.encolarSilla()
```

Dependiendo del problema, un manejador concreto podría, además de hacer su parte, invocar al siguiente manejador antes o después de actuar.



# Chain of responsibility - Implementación

```
class Evento
```

```
    Manejador manejadorDeProblemas
```

```
    void hacerFiesta()
```

```
        . . .
```

```
        manejadorDeProblemas.corregirSonido()
```

```
        . . .
```

# Chain of responsibility - Implementación

```
class Evento
```

```
    Manejador manejadorDeProblemas
```

```
    void hacerFiesta()
```

```
        . . .
```

```
        manejadorDeProblemas.corregirSonido()
```

```
        . . .
```



El cliente solo conoce a un responsable:  
*manejadorDeProblemas*.

Para el cliente es transparente que este  
responsable sea en realidad una cadena  
de responsables

# Chain of responsibility - Implementación

```
class Evento
```

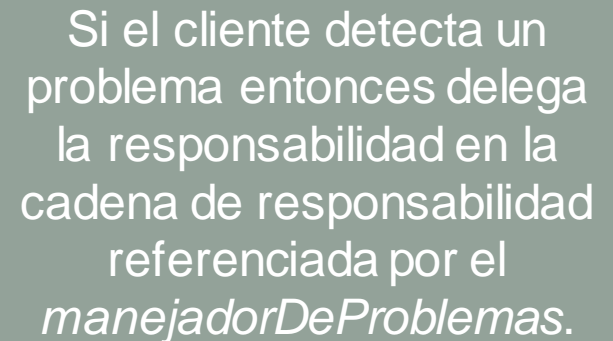
```
    Manejador manejadorDeProblemas
```

```
    void hacerFiesta()
```

```
        . . .
```

```
        manejadorDeProblemas.corregirSonido()
```

```
        . . .
```



Si el cliente detecta un problema entonces delega la responsabilidad en la cadena de responsabilidad referenciada por el *manejadorDeProblemas*.

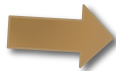
# Chain of responsibility - Implementación

void metodo

```
manejador = new TecnicoEnSonido (null)
manejador = new Electricista (manejador)
manejador = new Bombero (manejador)
manejador = new Medico (manejador)
manejador = new TecnicoEnSonido (manejador)
manejador = new Gasista (manejador)
manejador = new Carpintero (manejador)
```



evento.setManejador (manejador)



# Chain of responsibility - Implementación

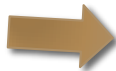
void metodo

```
manejador = new TecnicoEnSonido(null)
manejador = new Electricista(manejador)
manejador = new Bombero(manejador)
manejador = new Medico(manejador)
manejador = new TecnicoEnSonido(manejador)
manejador = new Gasista(manejador)
manejador = new Carpintero(manejador)
```

En algún lado se construye la cadena de responsabilidades y se las pasa al cliente



```
evento.setManejador (manejador)
```



# Chain of responsibility – Ventajas

- Reduce el acoplamiento. Este patrón libera a un objeto de tener que saber que otro objeto maneja una petición.
- Permite flexibilidad para agregar responsabilidades a objetos.

# Chain of responsibility – Desventajas

- No se garantiza que una petición sea procesada. La petición puede alcanzar el final de la cadena sin haber sido procesada.
- Una petición puede quedar sin ser atendida por un determinado manejador si la cadena está mal configurada.