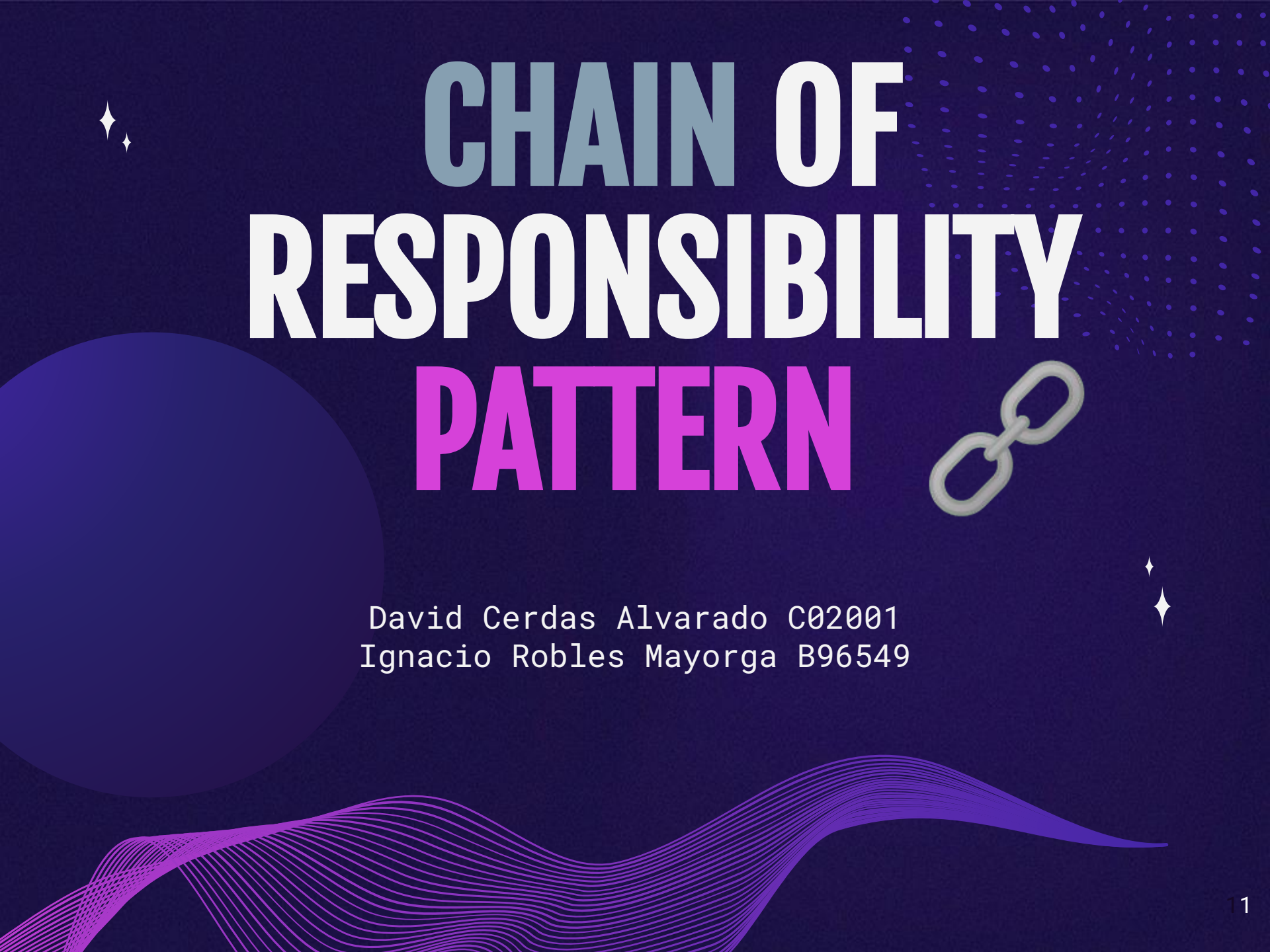


# CHAIN OF RESPONSIBILITY PATTERN



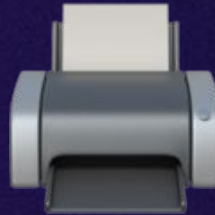
David Cerdas Alvarado C02001  
Ignacio Robles Mayorga B96549



# PROBLEMA



Solicitud



Manejadores  
(Handlers)





# PROBLEMA

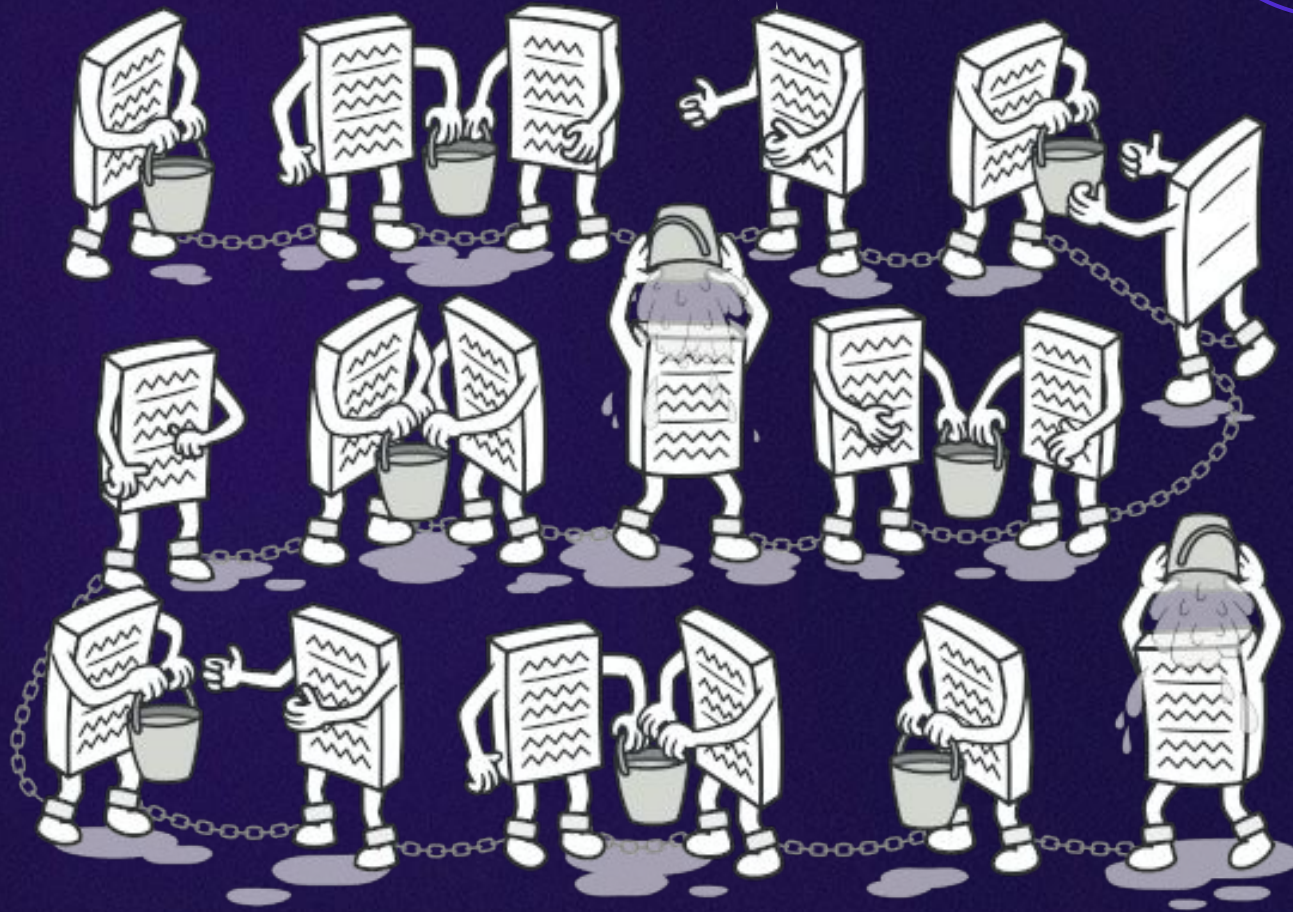


Solicitud



Manejadores  
(Handlers)

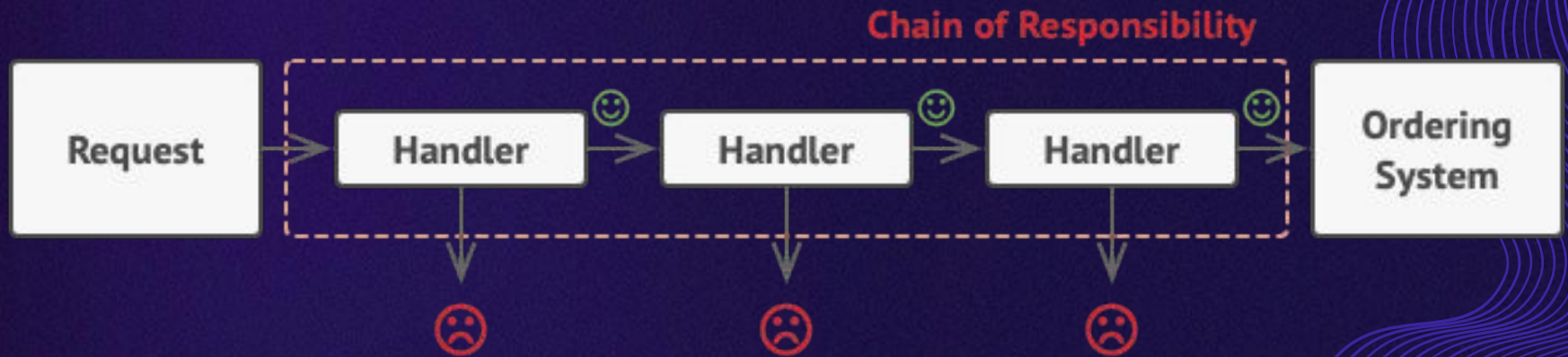
# Solución




*Imagen tomada de Refactoring Guru*



# Solución



*Imagen tomada de Refactoring Guru*



**¿Cuándo lo  
debo  
aplicar?**

# Aplicaciones



## Enrutamiento

Más de un objeto puede manejar la solicitud y el manejador(handler) no lo sabe de antemano. El handler debe aceptarlo automáticamente  
[Ejemplo Tickets]



## Flexibilidad

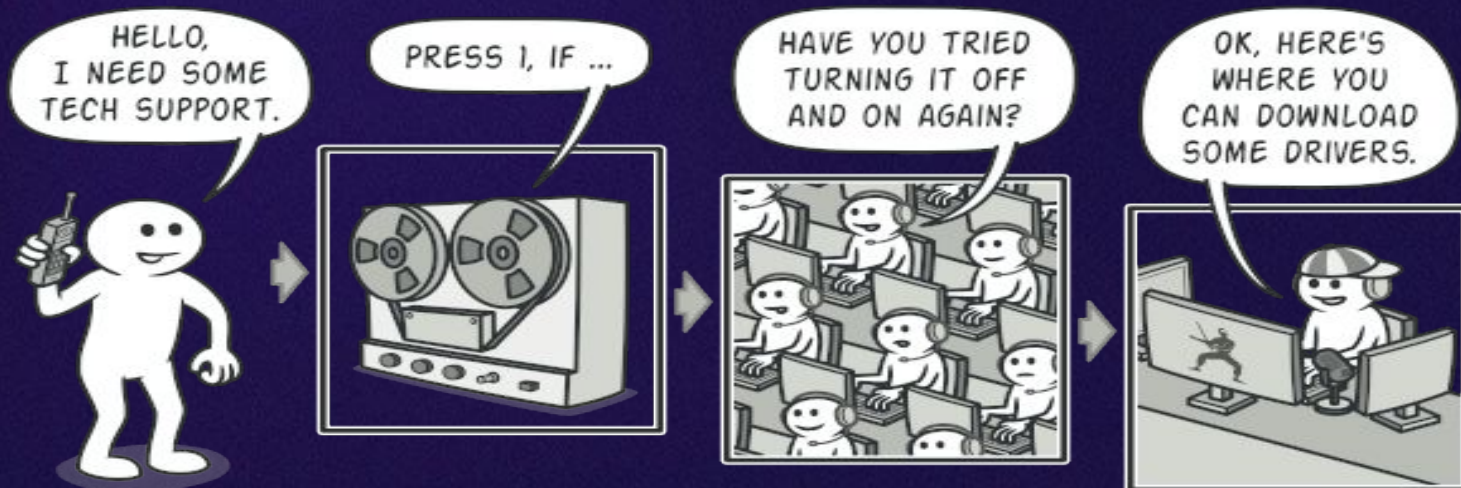
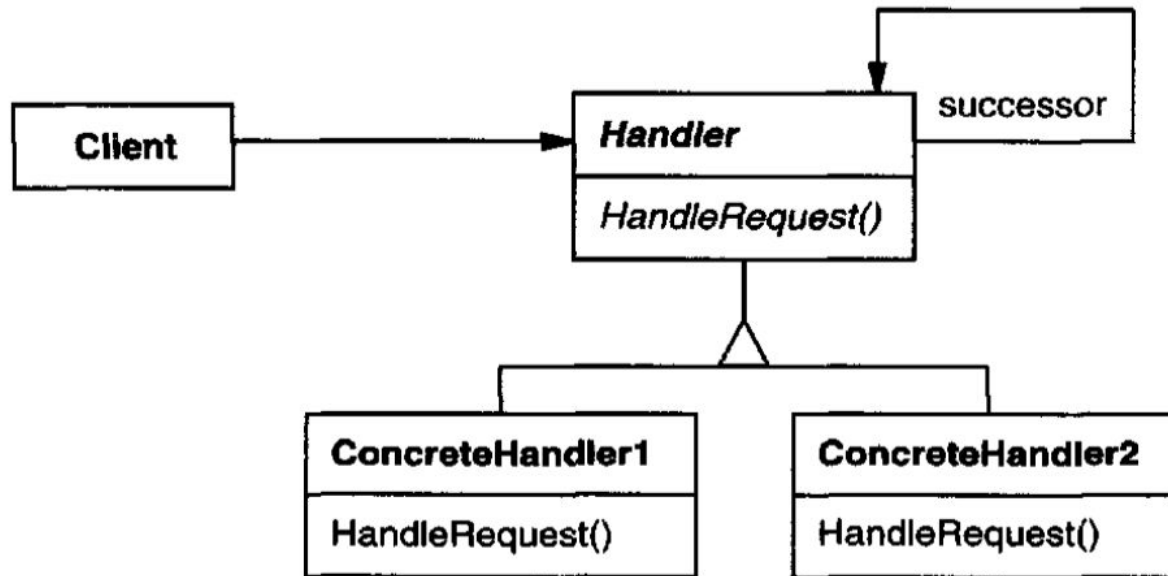
El conjunto de objetos que pueden manejar una solicitud debe especificarse dinámicamente. Se pueden agregar o eliminar objetos de la cadena según sea necesario  
[Ejemplo Flujo de trabajo según etapas]



## Desacoplamiento

Deseas enviar una solicitud a uno de varios objetos sin especificar explícitamente el receptor. [Ejemplo notificaciones]







# Ejemplo Sistema de la Universidad



**UCR**



**Registro**



**Oficina  
Administración  
Financiera**



**Oficina  
de Becas**



**Bienestar Y Salud**

# Ejemplo Sistema de la Universidad



**Registro**



**Oficina  
Administración  
Financiera**



**Oficina  
de Becas**



**Bienestar Y Salud**



**Handler**



**Handler**



**Handler**



**Handler**



# ¿Colaboraciones?



Cliente



Handler



Handler



Handler

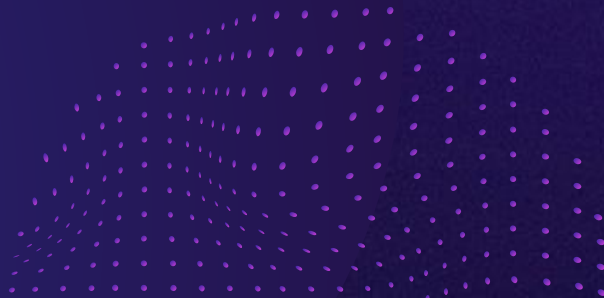


Handler



# IMPLEMENTACIÓN

A la hora de implementar una aplicación con un patrón de diseño chain of responsibility, se deben tomar en cuenta las siguientes consideraciones:

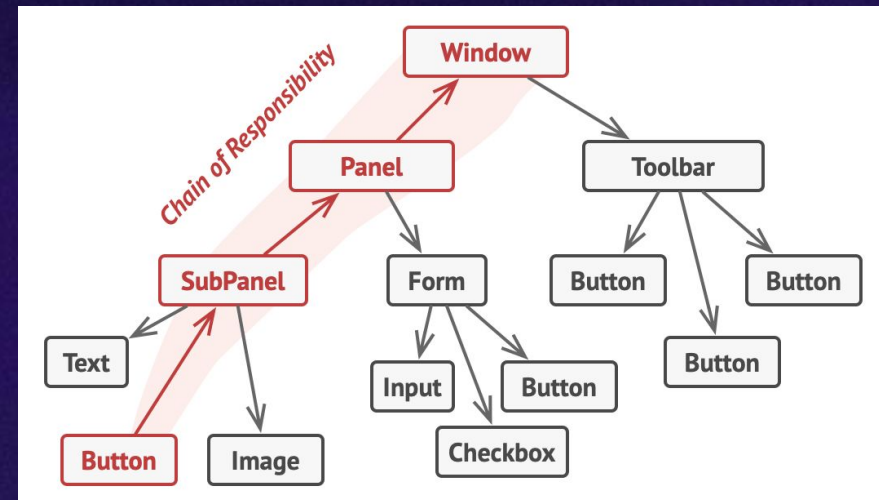
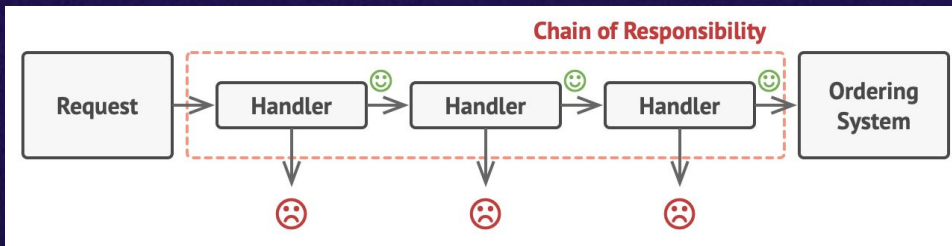




# HAY DOS VARIANTES DE IMPLEMENTACIÓN

CREAR NUEVOS LINKS

USAR LINKS EXISTENTES



# CONECTAR SUCESTORES

```
class HelpHandler {
public:
    HelpHandler(HelpHandler* s) : _successor(s) { }
    virtual void HandleHelp();
private:
    HelpHandler* _successor;
};

void HelpHandler::HandleHelp () {
    if (_successor) {
        _successor->HandleHelp();
    }
}
```



# REPRESENTACION DE REQUESTS

## OPCION 1: UN HANDLER POR CADA REQUEST

```
class Handler {  
    public:  
        Handler(Handler* s) : _successor(s) { }  
        virtual void HandleHelp();  
        virtual void HandlePrint();  
        virtual void HandlePayment();  
    private:  
        Handler* _successor;  
};
```

# REPRESENTACION DE REQUESTS

## OPCION 2: UN HANDLER PARA TODOS LOS REQUESTS

```
class Handler {  
    public:  
        Handler(Handler* s) : _successor(s) { }  
        virtual void HandleRequest(Request* theRequest);  
    private:  
        Handler* _successor;  
};
```



# REPRESENTACION DE REQUESTS

## IMPLEMENTACION DE LA OPCION 2:

```
void Handler::HandleRequest (Request* theRequest) {  
    switch (theRequest->GetKind()) {  
        case Help:  
            // cast argument to appropriate type  
            HandleHelp((HelpRequest*) theRequest);  
            break;  
        case Print:  
            HandlePrint((PrintRequest*) theRequest);  
            // . . .  
            break;  
        default:  
            // . . .  
            break;  
    }  
}
```



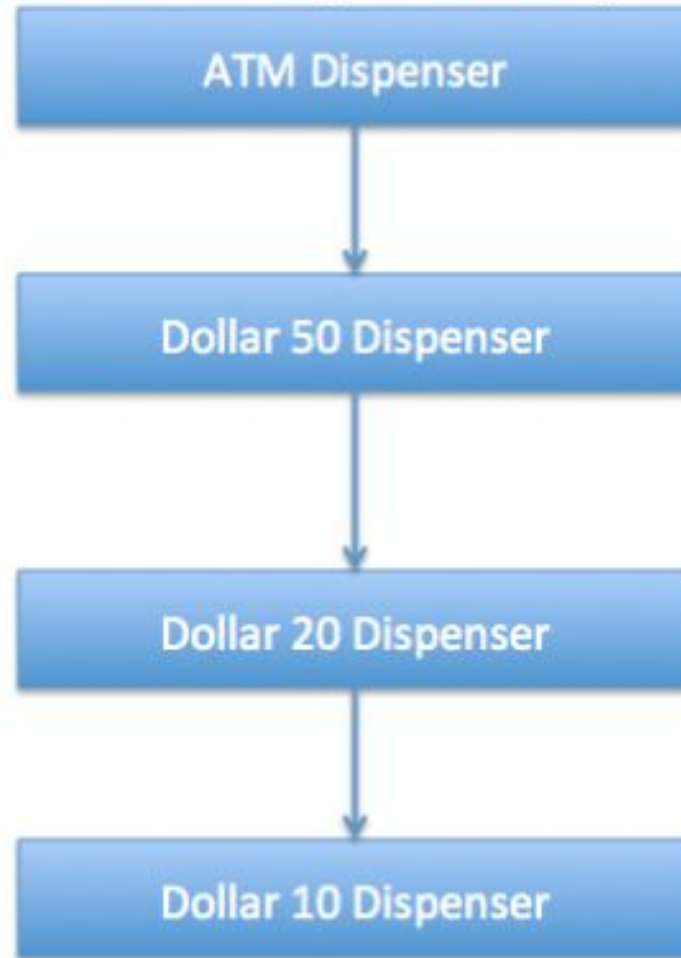
# EJEMPLO

Supongamos que se quiere programar un ATM. Este por dentro tiene la capacidad de dispensar 3 tipos de billetes: 50\$, 20\$ y 10\$. Para cada tipo de billete hay un dispensador diferente. Entonces, por dentro del ATM se tiene un dispensador de 50\$, uno de 20\$ y uno de 10\$. Por convención, el ATM quiere dispensar la menor cantidad de billetes posibles, o sea, entre más grande el billete, mejor.





Enter amount to dispense in multiples of 10



# CLASE WRAPPER PARA EL MONTO SOLICITADO

```
public class Currency {  
    private int amount;  
  
    public Currency(int amt){  
        this.amount=amt;  
    }  
  
    public int getAmount(){  
        return this.amount;  
    }  
}
```



# HANDLER

```
public interface DispenseChain {  
    void setNextChain(DispenseChain nextChain);  
    void dispense(Currency cur);  
}
```

# HANDLER CONCRETO DE 50\$

```
public class Dollar50Dispenser implements DispenseChain {

    private DispenseChain chain;

    @Override
    public void setNextChain(DispenseChain nextChain) {
        this.chain=nextChain;
    }

    @Override
    public void dispense(Currency cur) {
        if(cur.getAmount() >= 50){
            int num = cur.getAmount()/50;
            int remainder = cur.getAmount() % 50;
            System.out.println("Dispensing "+num+" 50$ note");
            if(remainder !=0) this.chain.dispense(new Currency(remainder));
        }else{
            this.chain.dispense(cur);
        }
    }
}
```



# HANDLER CONCRETO DE 20\$

```
public class Dollar20Dispenser implements DispenseChain{

    private DispenseChain chain;

    @Override
    public void setNextChain(DispenseChain nextChain) {
        this.chain=nextChain;
    }

    @Override
    public void dispense(Currency cur) {
        if(cur.getAmount() >= 20){
            int num = cur.getAmount()/20;
            int remainder = cur.getAmount() % 20;
            System.out.println("Dispensing "+num+" 20$ note");
            if(remainder !=0) this.chain.dispense(new Currency(remainder));
        }else{
            this.chain.dispense(cur);
        }
    }
}
```

# HANDLER CONCRETO DE 10\$

```
public class Dollar10Dispenser implements DispenseChain {

    private DispenseChain chain;

    @Override
    public void setNextChain(DispenseChain nextChain) {
        this.chain=nextChain;
    }

    @Override
    public void dispense(Currency cur) {
        if(cur.getAmount() >= 10){
            int num = cur.getAmount()/10;
            int remainder = cur.getAmount() % 10;
            System.out.println("Dispensing "+num+" 10$ note");
            if(remainder !=0) this.chain.dispense(new Currency(remainder));
        }else{
            this.chain.dispense(cur);
        }
    }
}
```

# CLIENTE

```
public class ATMDispenseChain {

    private DispenseChain c1;

    public ATMDispenseChain() {
        // initialize the chain
        this.c1 = new Dollar50Dispenser();
        DispenseChain c2 = new Dollar20Dispenser();
        DispenseChain c3 = new Dollar10Dispenser();

        // set the chain of responsibility
        c1.setNextChain(c2);
        c2.setNextChain(c3);
    }

    public static void main(String[] args) {
        ATMDispenseChain atmDispenser = new ATMDispenseChain();
        while (true) {
            int amount = 0;
            System.out.println("Enter amount to dispense");
            Scanner input = new Scanner(System.in);
            amount = input.nextInt();
            if (amount % 10 != 0) {
                System.out.println("Amount should be in multiple of 10s.");
                return;
            }
            // process the request
            atmDispenser.c1.dispense(new Currency(amount));
        }
    }
}
```





# CONSECUENCIAS



- El recibimiento de los requests no está garantizado



- Hay menos acoplamiento
- Más flexibilidad al asignar responsabilidades a los objetos ✨



# PATRONES RELACIONADOS

## Composite

En composite, el parent de un componente puede actuar como su sucesor.



# PRINCIPIOS SOLID

<b>Single Responsibility</b>	Cada handler en la cadena tiene una única responsabilidad, que es manejar un tipo específico de mensaje.
<b>Open/Close Principle</b>	El patrón permite extender el comportamiento de la cadena agregando nuevos handlers sin modificar los existentes
<b>Liskov's Substitution Principle</b>	Los handlers pueden ser sustituidos por subclases o implementaciones concretas sin alterar el funcionamiento del patrón.
<b>Dependency Inversion</b>	Permite que los clientes dependan de abstracciones (interfaz de handler) en lugar de depender de implementaciones concretas.





# PRINCIPIOS DE DISEÑO

**DRY**

**KISS**

**OOP**



# Gracias!

CREDITS: This presentation template was created by **Slidesgo**, including icons by **Flaticon** and infographics & images by **Freepik**

# Bibliografía

Gamma, Erich et.al. “Design Patterns”. Addison-Wesley, 1995.

Refactoring.Guru. (2023). Chain of Responsibility. *Refactoring.Guru*.

<https://refactoring.guru/design-patterns/chain-of-responsibility>

Pankaj. (2022). Chain of responsibility design pattern in java. *Digital Ocean*.

<https://www.digitalocean.com/community/tutorials/chain-of-responsibility-design-pattern-in-java>