Patrón Chain of Responsibility

Angel E. Concepción Capellán ID 1124530

ING-IDS309-01ARQUITECTURA DE SOFTWARE

Docente: GABRIEL ELIVAN VILLALONA EUSEBIO

¿Qué es Chain of Responsibility?

Patróndediseñode comportamiento

Desacoplael emisorde una solicitud de su receptor

Encadena múltiples manejadores que procesan la solicitud secuencialmente

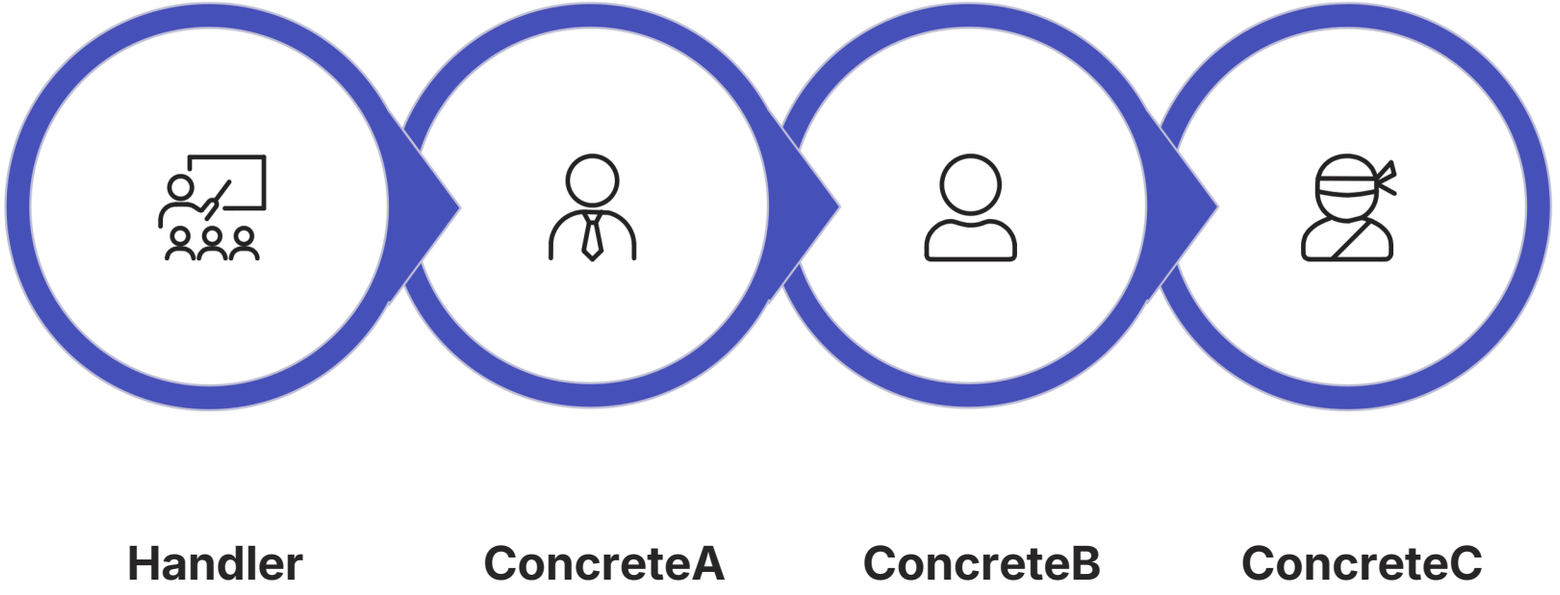
Cada manejador decide si procesa la solicitud o la pasa al siguiente

Permite crear sistemas flexibles sin conocer el destino final de la solicitud

Diagrama UML del Patrón

Mostrar un diagramaque represente:

- Handler (clase abstracta) con método handle()
- Tres ConcreteHandlers (ConcreteHandlerA, B, C) que heredan de Handler
- Relación de composición: cada Handler tiene una referencia al siguiente Handler
- Flujo de solicitud pasando de un handler al siguiente



Usar un diagrama visual que muestre la cadena de responsabilidad.

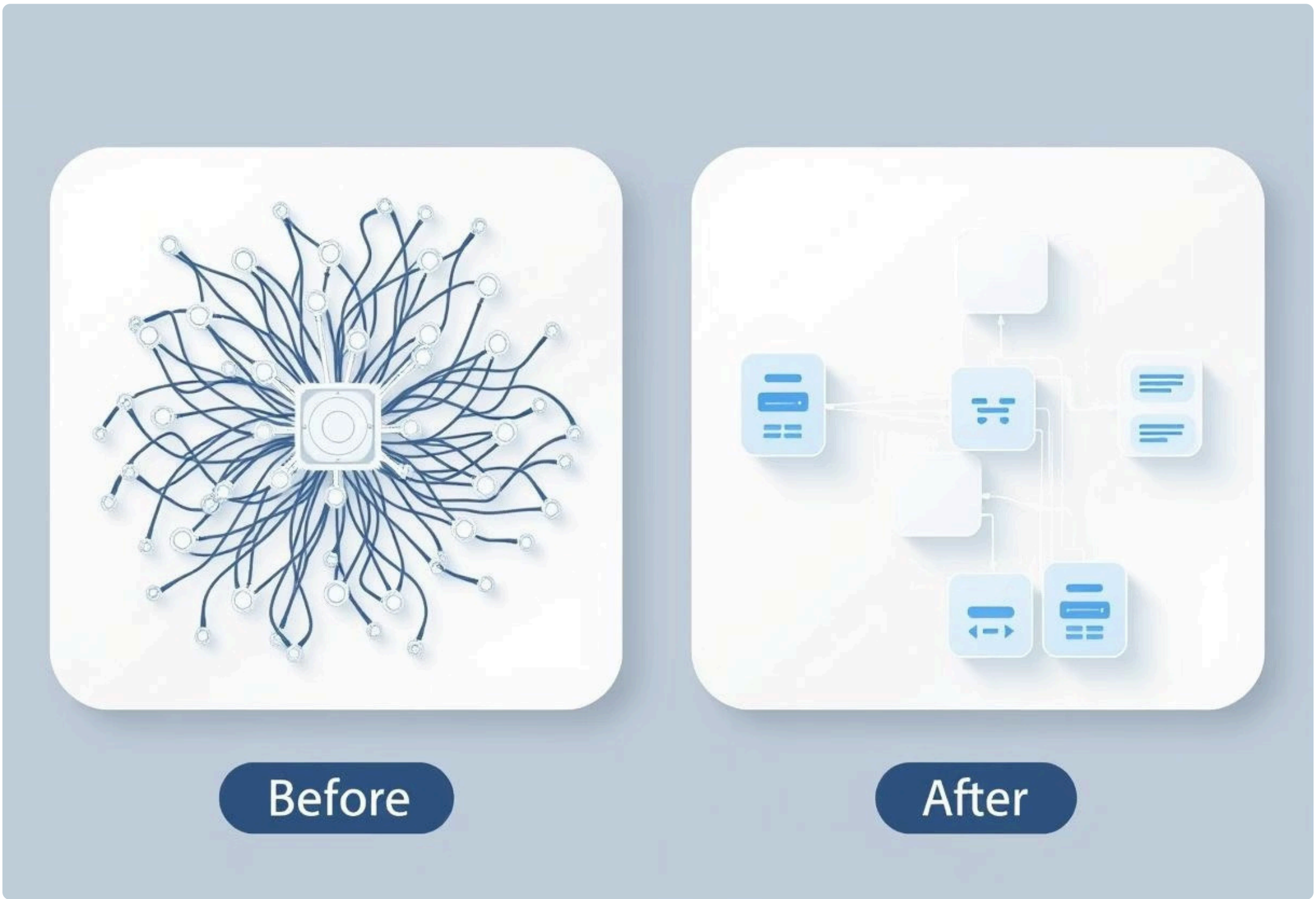
Problema que Resuelve

Sin Chain of Responsibility

- Acoplamiento fuerte entrecliente y manejadores
- Necesidad de conocer todos los posibles receptores
- Difícil de mantener y extender
- Lógica compleja en el cliente

Con Chain of Responsibility

- Desacoplamiento total
- Fácil agregar nuevos manejadores
- Lógica distribuida entre manejadores
- Sistema flexible y mantenible



Componentes Principales

Handler (Manejador Abstracto)

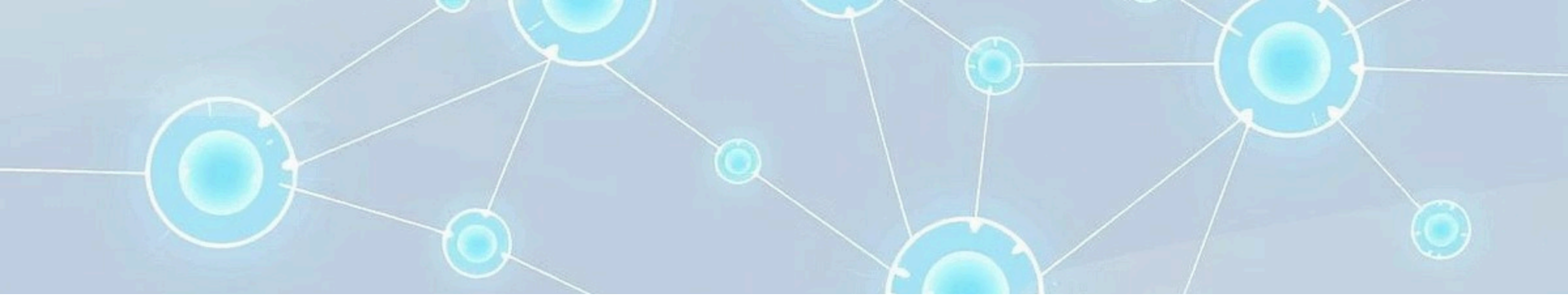
- Definela interfazpara procesar solicitudes
- Mantiene referencia al siguiente manejador
- Implementa la lógica de encadenamiento

ConcreteHandler(Manejador Concreto)

- Implementalalógica específica de procesamiento
- Decide si procesa o pasa al siguiente
- Puede haber múltiples implementaciones

Client (Cliente)

- Crea la cadena de manejadores
- Envía la solicitud al primer manejador
- No conoce los detalles de procesamiento



Implementación: Cliente y Construcción de la Cadena

Función principal que demuestra el uso del patrón Chain of Responsibility. Crea la cadena de manejadores y procesa solicitudes.

```
def main():
    """
 Función principal que demuestra el uso del patrón Chain of Responsibility.
Crea la cadena de manejadores y procesa solicitudes.
    """
    # Crear los manejadores
    basic_support = BasicSupport()
    specialized_tech = SpecializedTechnician()

 support_manager = SupportManager()

    # Construir la cadena: BasicSupport ³ SpecializedTechnician ³ SupportManager
 basic_support.set_next(specialized_tech).set_next(support_manager)

    # Crear solicitudes de prueba
    requests = [
        SupportRequest(1, "Olvidé mi contraseña", "low"),
        SupportRequest(2, "Error en la configuración del servidor", "high"),
        SupportRequest(3, "Fallo crítico del sistema", "critical"),
    ]
    #  P r o c e s a r  c a d a  s o l i c i t u d
    f o r  r e q u e s t  i n  r e q u e s t s :

        print(f"\n--- Procesando {request} ---")
        result = basic_support.handle(request)
        print(f"Resultado: {result}\n")

if __name__ == "__main__":
 main()
```

Implementación: SupportManager Handler

```
class SupportManager(SupportHandler):
    """
 Manejador de gerente de soporte.
Resuelve problemas críticos o no resueltos por otros niveles.
    """
    def handle(self, request):

        # Este es el último nivel, resuelve todo
        print(f"7 SupportManager resolviendo: {request.description}")
        request.resolved = True
        request.handler_name = "SupportManager"
        # Si hay un siguiente manejador, también lo notifica
        if self._next_handler:

            return self._next_handler.handle(request)

        return request
```

Implementación: SpecializedTechnician Handler

```
class SpecializedTechnician(SupportHandler):
    """
 Manejador de soporte técnico especializado.
 Resuelve problemas técnicos complejos.
    """
    def handle(self, request):

        # Problemas que puede resolver este nivel
        if request.priority in ['medium', 'high']:
            print(f"7 SpecializedTechnician resolviendo: {request.description}")
            request.resolved = True
            request.handler_name = "SpecializedTechnician"
            return request
        else:
            # Si no puede resolver, pasa al siguiente manejador
            if self._next_handler:
                print(f"³ SpecializedTechnician escalando a siguiente nivel...")
                return self._next_handler.handle(request)
            else:
                print(f"  No hay manejador disponible para: {request}")
                return request
```

Implementación: BasicSupport Handler

```
class BasicSupport(SupportHandler):
    """
 Manejador de soporte básico.
Resuelve problemas simples como contraseñas y acceso básico.
    """
    def handle(self, request):

        # Problemas que puede resolver este nivel
        if request.priority in ['low', 'medium']:
            print(f"7 BasicSupport resolviendo: {request.description}")
            request.resolved = True
            request.handler_name = "BasicSupport"
            return request
        else:
            # Si no puede resolver, pasa al siguiente manejador
            if self._next_handler:
                print(f"³ BasicSupport escalando a siguiente nivel...")
                return self._next_handler.handle(request)
            else:
                print(f"  No hay manejador disponible para: {request}")
                return request
```

Implementación: Clase SupportRequest

Representa una solicitud de soporte técnico. Contiene información sobre el problema a resolver.

```
class SupportRequest:
    """
    Representa una solicitud de soporte técnico.
 Contiene información sobre el problema a resolver.
    """
    def __init__(self, request_id, description, priority):

        self.request_id = request_id
        self.description = description
        # Prioridad: 'low', 'medium', 'high', 'critical'
        self.priority = priority
        self.resolved = False
        self.handler_name = None

    def __str__(self):
        return (f"ID: {self.request_id} | "
                f"Prioridad: {self.priority} | "
                f"Descripción: {self.description} | "
                f"Resuelto: {self.resolved}")
```

Implementación: Clase Base Handler

```
from abcimportABC,abstractmethod

class SupportHandler(ABC):
    """
 Clase abstracta que define la interfaz para los manejadores.
 Cada manejador tiene una referencia al siguiente en la cadena.
    """

    def __init__(self):

        # Referencia al siguiente manejador en la cadena
        self._next_handler = None

    def set_next(self, handler):
        """Establece el siguiente manejador en la cadena"""
        self._next_handler = handler
        return handler # Permite encadenamiento

    @abstractmethod
    def handle(self, request):
        """Método abstracto que debe implementar cada manejador concreto"""
        pass
```

Estructura del Código Python

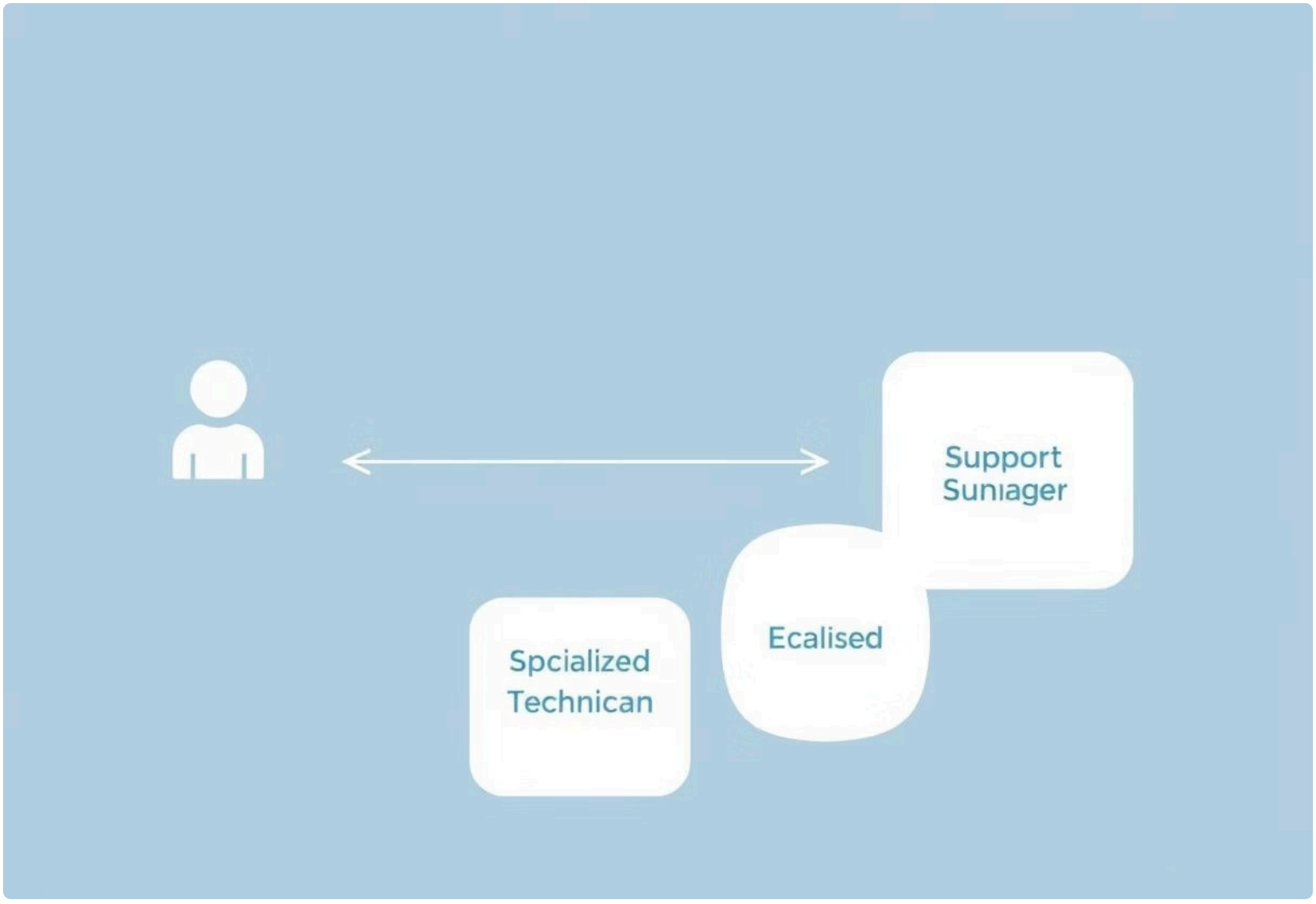
Clases principales:

- SupportHandler(clase abstracta)
  - Método: handle(request)
  - Atributo: next_handler
- BasicSupport (ConcreteHandler)
  - Maneja problemas simples
- SpecializedTechnician (ConcreteHandler)
  - Maneja problemas técnicos
- SupportManager (ConcreteHandler)
  - Maneja problemas críticos

Clases de soporte:

- SupportRequest
  - Atributos: id, description, priority
- Cliente
  - Crea la cadena
  - Envía solicitudes

Caso Práctico: Sistema de Soporte Técnico



Una empresa de tecnología necesita un sistema de gestión de soporte técnico donde las solicitudes de los clientes pasan por diferentes niveles de atención:

Soporte Básico

Resuelve problemas simples (contraseñas, acceso básico)

Técnico Especializado

Maneja problemas técnicos complejos (configuración, bugs)

Gerente de Soporte

Escala problemas críticos o no resueltos

Cada nivel intenta resolver el problema. Si no puede, lo pasa al siguiente nivel.

El cliente solo envía la solicitud al primer nivel sin conocer los detalles internos.

Desventajas del Patrón

- 7 **Complejidad:** Puede ser excesivo para sistemas simples
- 7 **Rendimiento:** Múltiples iteraciones en la cadena
- 7 **Depuración:** Difícil rastrear el flujo de ejecución
- 7 **Solicitud no procesada:** Si ningún manejador procesa, la solicitud se pierde
- 7 **Orden importante:** El orden de los manejadores es crítico
- 7 **Overhead:** Creación y mantenimiento de la cadena

Ventajas del Patrón

Desacoplamiento

Elclientenoconocelosdetalles de los manejadores

Flexibilidad

Fácilagregar,remover o reordenar manejadores

Responsabilidad única

Cadamanejadortieneunaresponsabilidad específica

Reutilización

Losmanejadorespueden usarse en diferentes contextos

Dinámico

Lacadenapuede construirse en tiempo de ejecución

Mantenibilidad

Cambioslocalizadossin afectar el resto del sistema