

METODOLOGÍAS DE PROGRAMACIÓN I

Patrón estructural *Decorator*

Situación de ejemplo

- Una agencia de turismo posee un sistema que le permite administrar paquetes turísticos, el cual consiste en la estadía por un cierto período en un cierto lugar.
- Para ello cuenta con una clase llamada *PaqueteTuristico* que además de guardar el lugar y las fechas de llegada y partida posee métodos para saber el precio, consultar la disponibilidad (llamando al lugar de alojamiento) y hacer efectiva una reserva.

```
class PaqueteTuristico
    string fechaDeLlegada, fechaDePartida
    string lugar

    double precio()
    bool consultarDisponibilidad()
    void reservar()
```

Problema

Que sucedería si ahora la agencia de turismo quiere agregar adicionales al paquete turístico: alquiler de auto, pasaje de avión, venta de entradas a sitios de interés o de recitales, seguro médico, etc. Todos estos servicios son opcionales y el cliente puede contratar cualquiera de ellos.

Pensaríamos en una solución similar a esta:

```
class PaqueteTuristico{  
    boolean alquilerDeAuto  
    boolean pasajeDeAvion  
    boolean entradaRecital  
    boolean seguroMedico  
    . . .  
}
```

Problema

Que sucedería si ahora la agencia de turismo quiere agregar adicionales al paquete turístico: alquiler de auto, pasaje de avión, venta de entradas a sitios de interés o de recitales, seguro médico, etc. Todos estos servicios son opcionales y el cliente puede contratar cualquiera de ellos.

Pensaríamos en una solución similar a esta:

```
class PaqueteTuristico{  
    boolean alquilerDeAuto  
    boolean pasajeDeAvion  
    boolean entradaRecital  
    boolean seguroMedico
```

Además de estos "flags" hay que tener variables con el auto alquilado, los boletos de avión, las entradas, los datos del seguro, etc....

Problema

¿Cómo sería la implementación del método *precio*?

```
double precio()  
    double r = precio_del_lugar * cantidad_de_dias  
    if (alquilerDeAuto)  
        r = r + plus_auto  
    if (pasajeDeAvion)  
        r = r + plus_avion  
    if (entradaRecital)  
        r = r + plus_recital  
    if (seguroMedico)  
        r = r + plus_seguro  
    . . .  
    return r
```

Problema

¿Cómo sería la implementación del método *precio*?

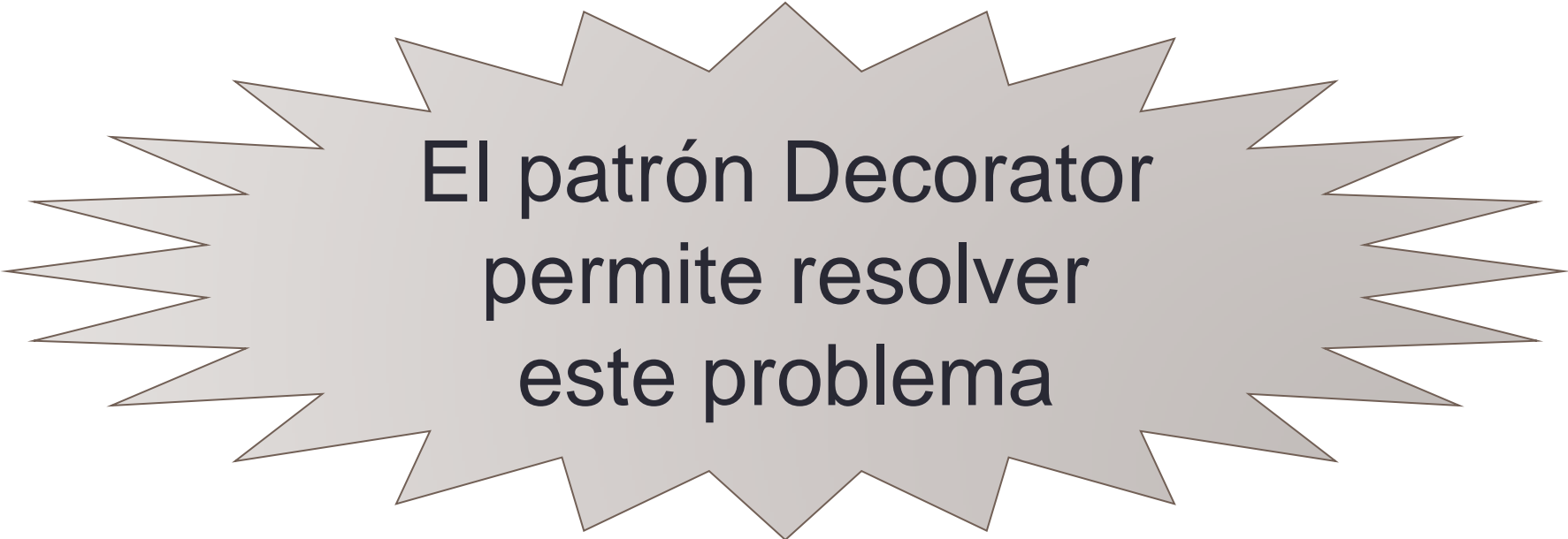
```
double precio()  
    double r = precio_del_lugar * cantidad_de_dias  
    if (alquilerDeAuto)  
        r = r + plus_auto  
    if (pasajeDeAvion)  
        r = r + plus_avion  
    if (entradaRecital)  
        r = r + plus_recital  
    if (seguroMedico)  
        r = r + plus_seguro  
    . . .  
    return r
```

Algo parecido hay que hacer en los métodos *consultarDisponibilidad* y *reservar*.

Con el agregado de que es más complejo ya que hay que averiguar mediante aerolíneas, alquileres de coches, teatros, cines, museos, aseguradoras, etc.

Motivación

Buscamos un diseño que permita agregar cualquier número de características o "decorados" a un objeto, sin alterar el propio objeto y que sea transparente para el cliente y que pueda tratar con el objeto simple o "decorado" de manera independiente.



**El patrón Decorator
permite resolver
este problema**

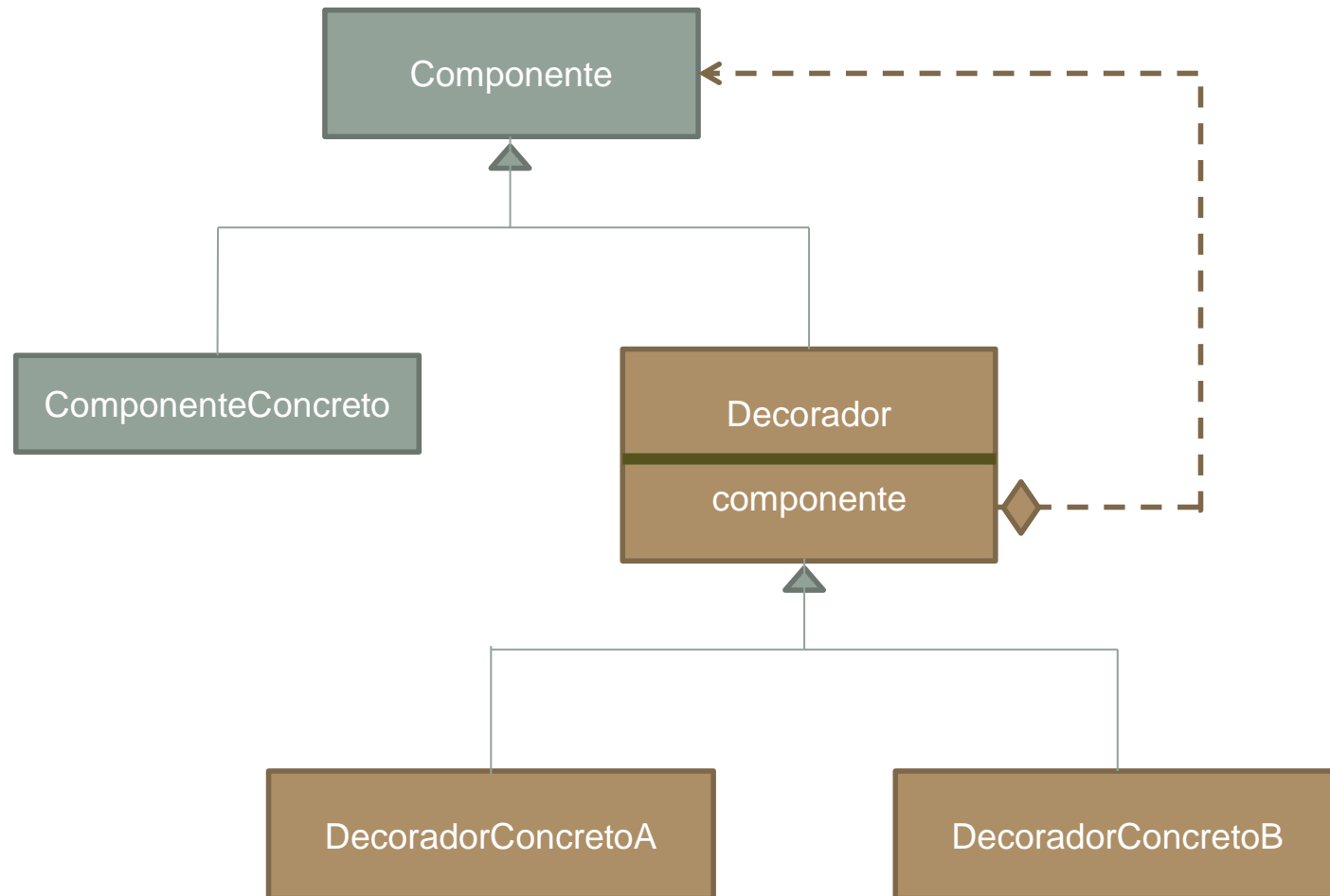
Decorator

Propósito: Asigna responsabilidades adicionales a un objeto dinámicamente, proporcionando una alternativa flexible a la herencia para extender la funcionalidad.

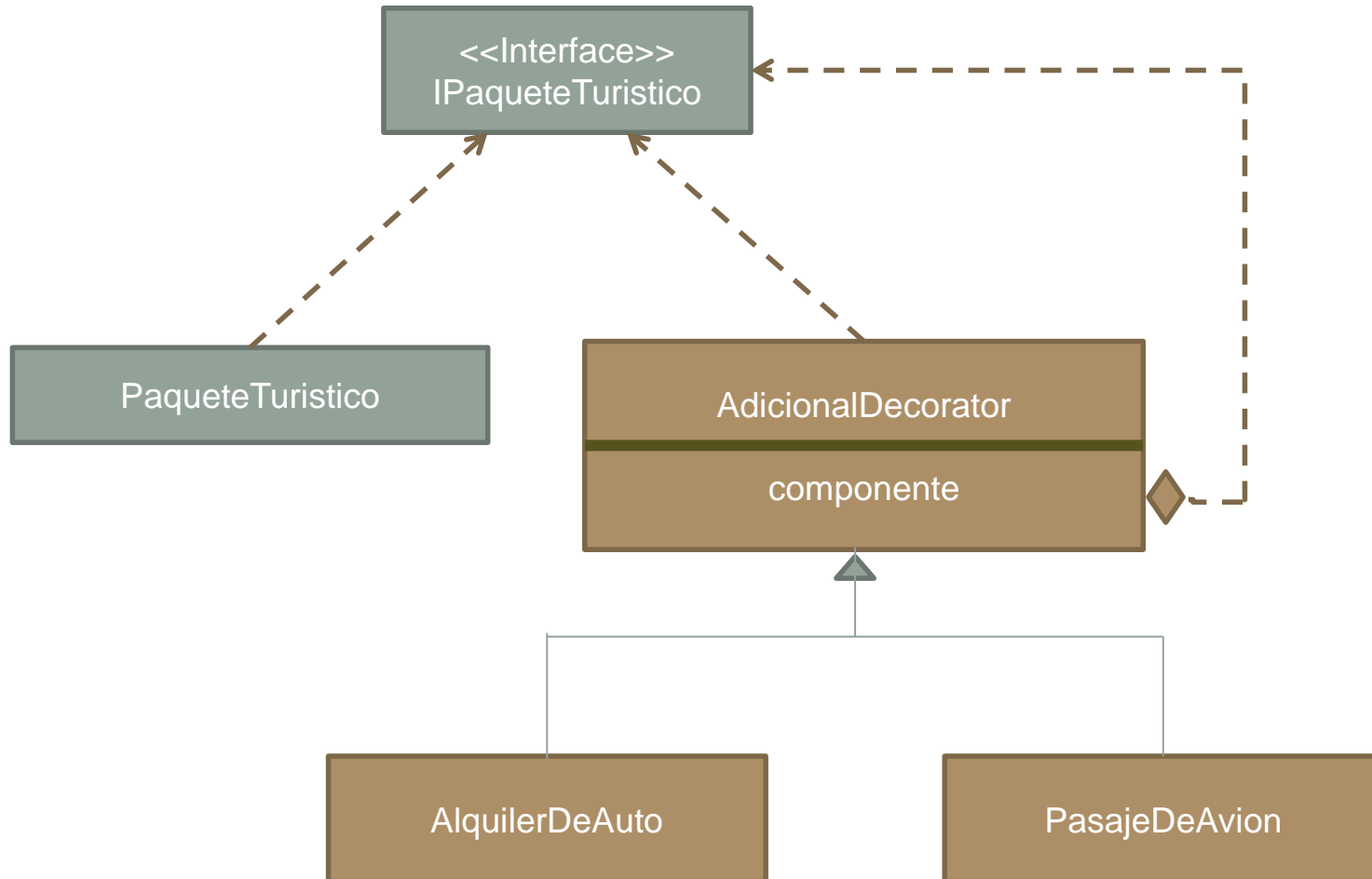
Aplicabilidad: usarlo cuando

- Quiera añadir responsabilidades individuales de forma dinámica y transparente.
- Sea necesario retirar responsabilidades
- La extensión por herencia no es viable

Decorator - Estructura



Decorator - Estructura



Decorator - Implementación

```
interface IPaqueteTuristico
```

```
    double precio()
```

```
    bool consultarDisponibilidad()
```

```
    void reservar()
```

Definimos una interface para las posibles características que puede tener un paquete turístico. Cada característica deberá implementar los métodos *precio*, *consultarDisponibilidad* y *reservar*.

Decorator - Implementación

```
class PaqueteTuristico : IPaqueteTuristico
    double precio()
        return precio_del_lugar * cantidad_de_dias
```

En la clase concreta *PaqueteTuristico* está el comportamiento "stándard" del paquete turístico.

Decorator - Implementación

```
abstract class AdicionalDecorator : IPaqueteTuristico
    IPaqueteTuristico adicional

    constructor(IPaqueteTuristico a)
        adicional = a

    double precio()
        return adicional.precio()
```

Decorator - Implementación

```
abstract class AdicionalDecorator : IPaqueteTuristico
    IPaqueteTuristico adicional

    constructor(IPaqueteTuristico a)
        adicional = a

    override fun precio()
        return adicional.precio()
```

Notar que la superclase *AdicionalDecorator* la declaramos como abstracta ya que no nos interesa instanciarla

Decorator - Implementación

```
abstract class AdicionalDecorator : IPaqueteTuristico
    IPaqueteTuristico adicional

    constructor(IPaqueteTuristico a)
        adicional = a

    double precio()
        return adicional.precio()
```



Todo decorador debe redirigir la petición a su componente asociado.

Decorator - Implementación

```
class AlquilerDeAuto : AdicionalDecorator

    double precio()
        return super.precio() + plus_auto
```


Decorator - Implementación

```
public metodoQueCrea ()
```

```
    pt = new PaqueteTuristico()
```

```
    pt = new AlquilerDeAuto(pt)
```

```
    pt = new PasajeDeAvion(pt)
```

```
    pt = new EntradaTeatro(pt)
```

```
    pt = new SeguroMedico(pt)
```

```
public metodoQueUsa ()
```

```
    if (pt.consultarDisponibilidad())
```

```
        pt.reservar()
```

```
        print(pt.precio())
```

Decorator - Implementación

```
public metodoQueCrea ()
```

```
    pt = new PaqueteTuristico()  
    pt = new AlquilerDeCarro(pt)  
    pt = new PasajeDeAvion(pt)  
    pt = new EntradaTeatro(pt)  
    pt = new SeguroMedico(pt)
```

Se referencia al último decorado creado
quien conoce al anterior, y así hasta
llegar al *PaqueteTuristico*

```
public metodoQueUsa ()
```

```
    if (pt.consultarDisponibilidad())  
        pt.reservar()  
        print(pt.precio())
```



SeguroMedico

EntradaTeatro

PasajeDeAvion

AlquilerDeAuto

PaqueteTuristico

```
public metodoQueUsa()
```

```
    pt = new PaqueteTuristico()
```

```
    pt = new AlquilerDeAuto(pt)
```

```
    pt = new PasajeDeAvion(pt)
```

```
    pt = new EntradaTeatro(pt)
```

```
    pt = new SeguroMedico(pt)
```

```
public metodoQueUsa()
```

```
    if (pt.consultarDisponibilidad())
```

```
        pt.reservar()
```

```
        print(pt.precio())
```

Decorator - Implementación

```
public metodoQueCrea ()
```

```
    pt = new PaqueteTuristico()
```

```
    pt = new AlquilerDeAuto(pt)
```

```
    pt = new PasajeDeAvion(pt)
```

```
    pt = new EntradaTeatro(pt)
```

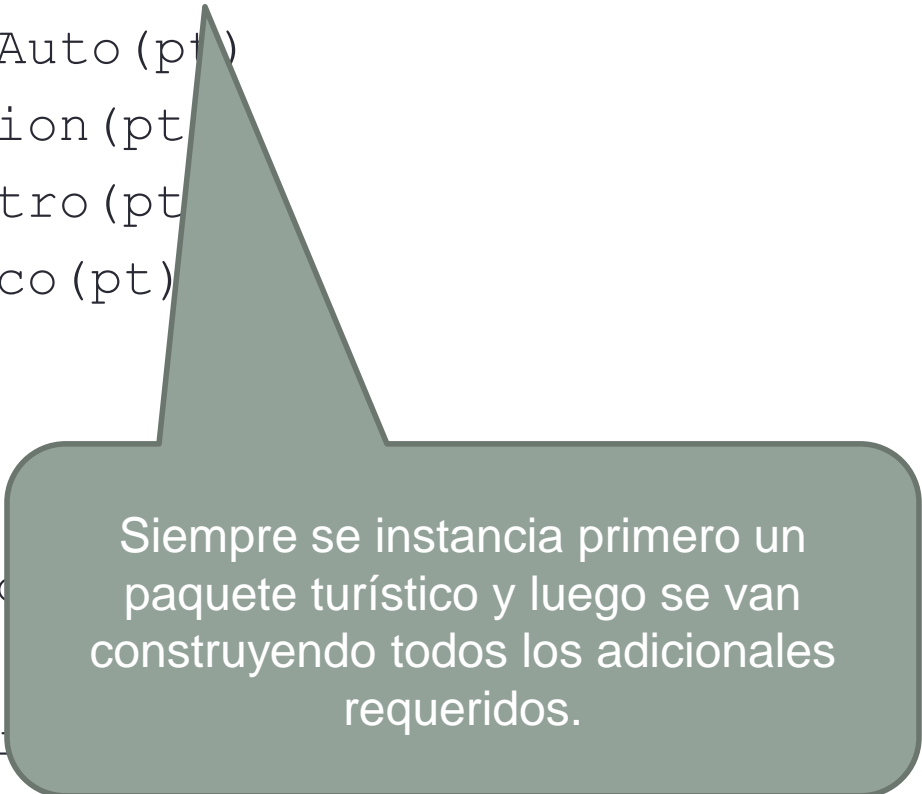
```
    pt = new SeguroMedico(pt)
```

```
public metodoQueUsa ()
```

```
    if (pt.consultarDispo
```

```
        pt.reservar ()
```

```
        print (pt.preci
```

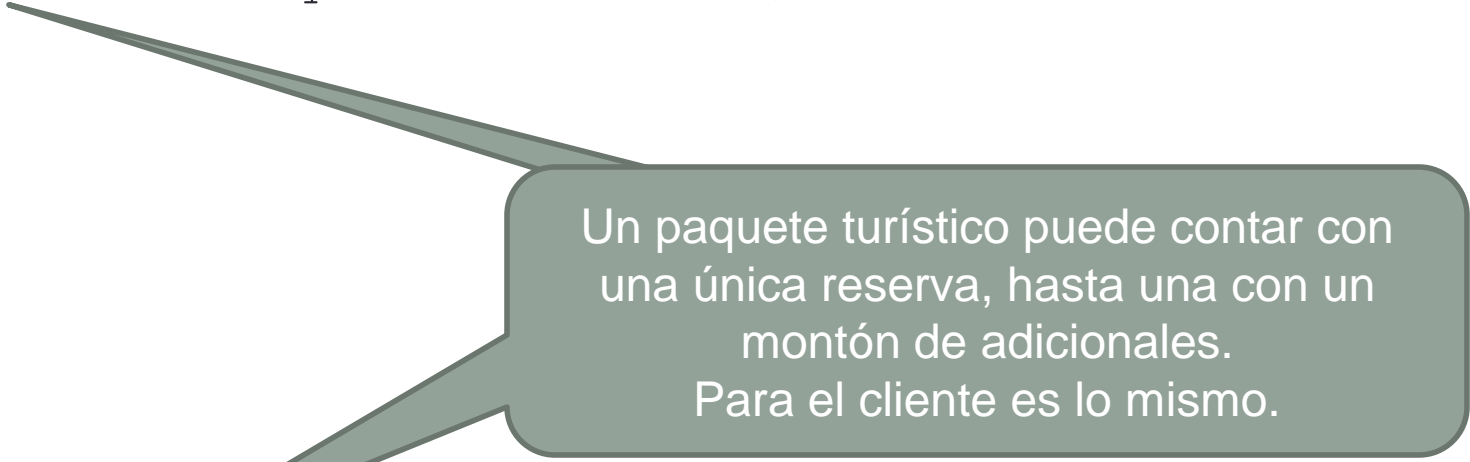


Siempre se instancia primero un paquete turístico y luego se van construyendo todos los adicionales requeridos.

Decorator - Implementación

```
public metodoQueCrea ()
```

```
    pt = new PaqueteTuristico()
```



Un paquete turístico puede contar con una única reserva, hasta una con un montón de adicionales. Para el cliente es lo mismo.

```
public metodoQueUsa ()
```

```
    if (pt.consultarDisponibilidad())
```

```
        pt.reservar ()
```

```
        print (pt.precio())
```

Decorator – Ventajas

- Más flexibilidad que la herencia estática. El patrón Decorator proporciona una manera más flexible de añadir responsabilidades a los objetos. Es posible añadir y quitar responsabilidades en tiempo de ejecución.
- Evita clases cargadas de funciones en las partes superiores de las jerarquías.
- Muchos objetos pequeños. El uso del patrón Decorator da como resultado sistemas formados por muchos objetos pequeños muy parecidos entre sí.