Decorator

Objective

Implementing additional responsibilities to an object, so dynamic, and provide a flexible alternative to extend the functionality of a subclass.

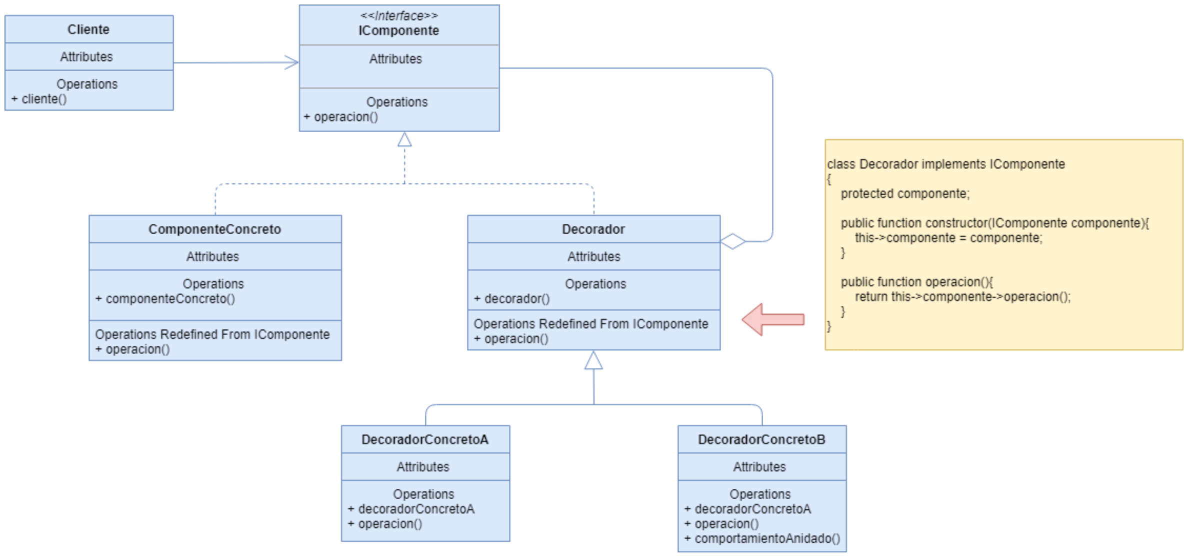
Function

Facilitate the addition of functionality to a class dynamically.

Structure

The client is always interested in CentralFunctionality.doSomething(). The client may, or may not, be interested in OptionOne.doSomething() and OptionTwo.doSomething(). Each of these classes always delegates to the base class Decorator, and that class always delegates to the object 'wrappee' content.

The structure that meets this pattern is shown in Figure 1



```
class Decorador implements IComponente
{
    protected componente;

    public function constructor(IComponente componente){
        this->componente = componente;
    }

    public function operacion(){
        return this->componente->operacion();
    }
}
```

Figure 1: UML Diagram Decorator Pattern

Applications

The use of the Decorator pattern is recommended when:

- The system requires that responsibilities be added dynamically and transparently to individual objects; that is, without affecting the other objects.

- Withdrawal of responsibility without affecting operation is required.

- Extensibility to subclass is impractical. Sometimes a large number of independent extensions is possible and would result in an explosion of subclasses for to support each combination.

Design Patterns Collaborators

- A Decorator pattern is normally considered a single-component generated composite pattern; as a Decorator adds responsibilities additional.

- The Decorator and Strategy patterns work together, in the area of variation of an object; one is in charge of its presentation while the other their form, respectively.

Scope of action

Applied at the object level.

Problem

To add a specific behavior or state to the individually and at run time, it is required to implement the inheritance; without However, this applies to an entire class in a static manner and the customer loses control.

Solution

The Decorator pattern forms the interface of the component that works as the boundary between a class and its respective subclasses, which is transparent to clients, the decorator class refers the order to the component and performs while transparency allows for recursively nesting decorators so that an unlimited number of aggregate liabilities are satisfied dynamically.

Diagram or Implementation

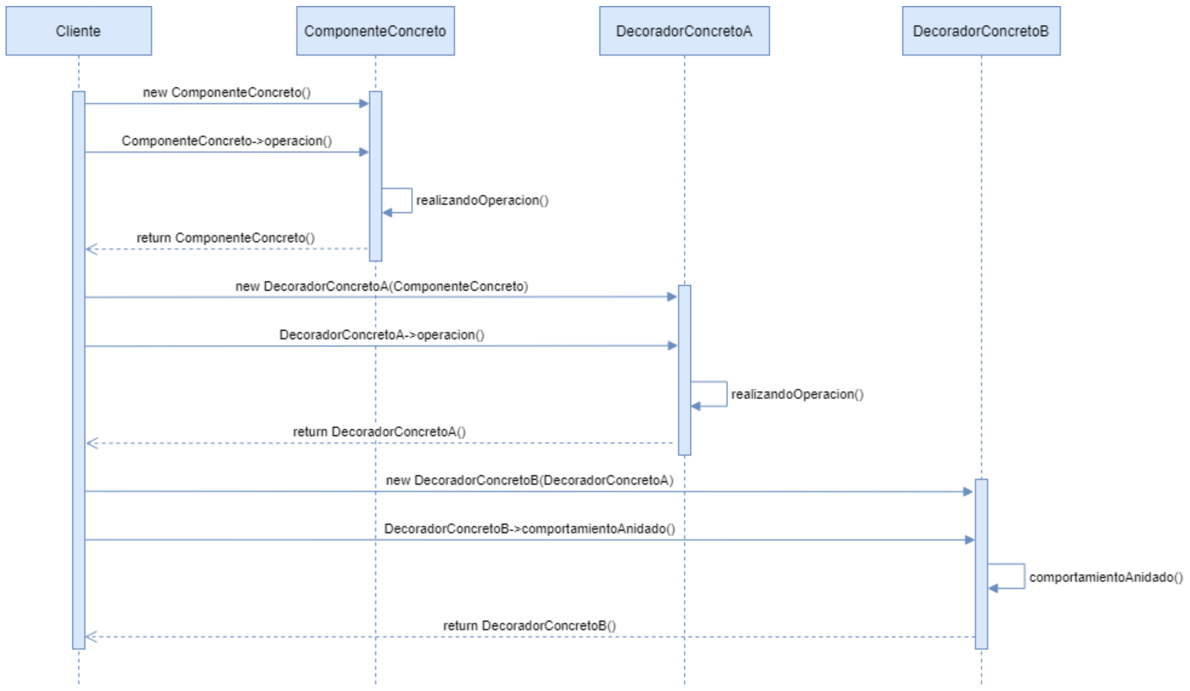

Cliente
ComponenteConcreto
DecoradorConcretoA
DecoradorConcretoB
new ComponenteConcreto()
ComponenteConcreto->operacion()
realizandoOperacion()
return ComponenteConcreto()
new DecoradorConcretoA(ComponenteConcreto)
DecoradorConcretoA->operacion()
realizandoOperacion()
return DecoradorConcretoA()
new DecoradorConcretoB(DecoradorConcretoA)
DecoradorConcretoB->comportamientoAnidado()
comportamientoAnidado()
return DecoradorConcretoB()

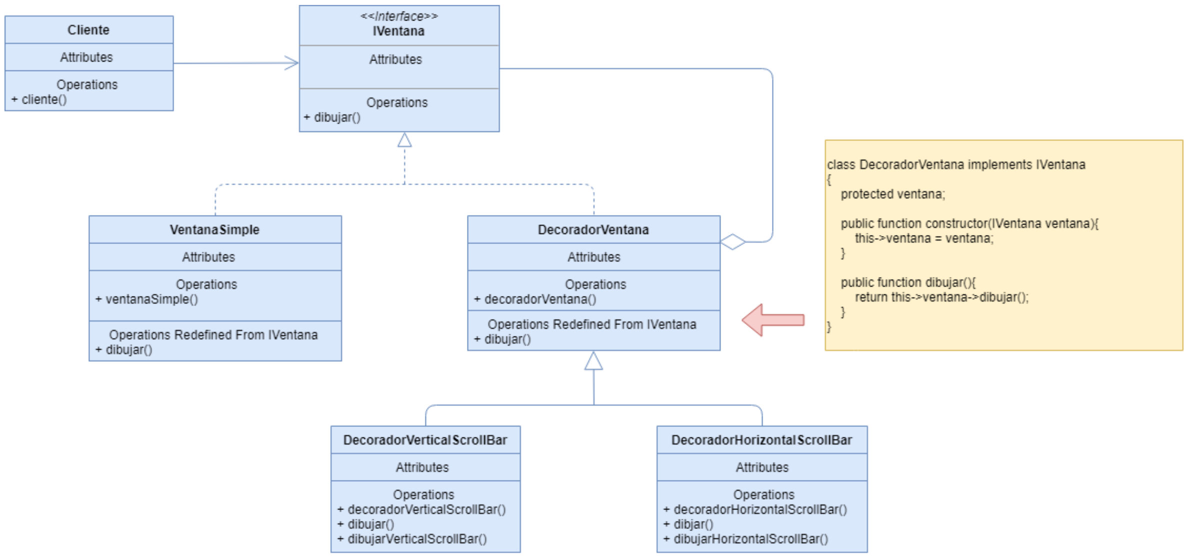
Figure 2: UML Diagram Decorator Pattern

Figure 2 explains the behaviour of the Decorator pattern by means of a sequence diagram.

- Client Class performs an operation on the DecoratorA component.
- The component DecoratorA performs the same operation on component DecoratorB.
- The DecoradorB component class performs an action on ConcreteComponent.
- The component type DecoratorB executes a decoration operation.
- The component type DecoratorA executes a decoration operation.
- The Client class receives as a result an object decorated by all the Decorators, which encapsulated the Component in several layers.

Study Cases

Interface System



```
class DecoradorVentana implements IVentana
{
   protected ventana;

   public function constructor(IVentana ventana){
      this->ventana = ventana;
   }

   public function dibujar(){
      return this->ventana->dibujar();
   }
}
```

Figure 3: UML Diagram Interface System

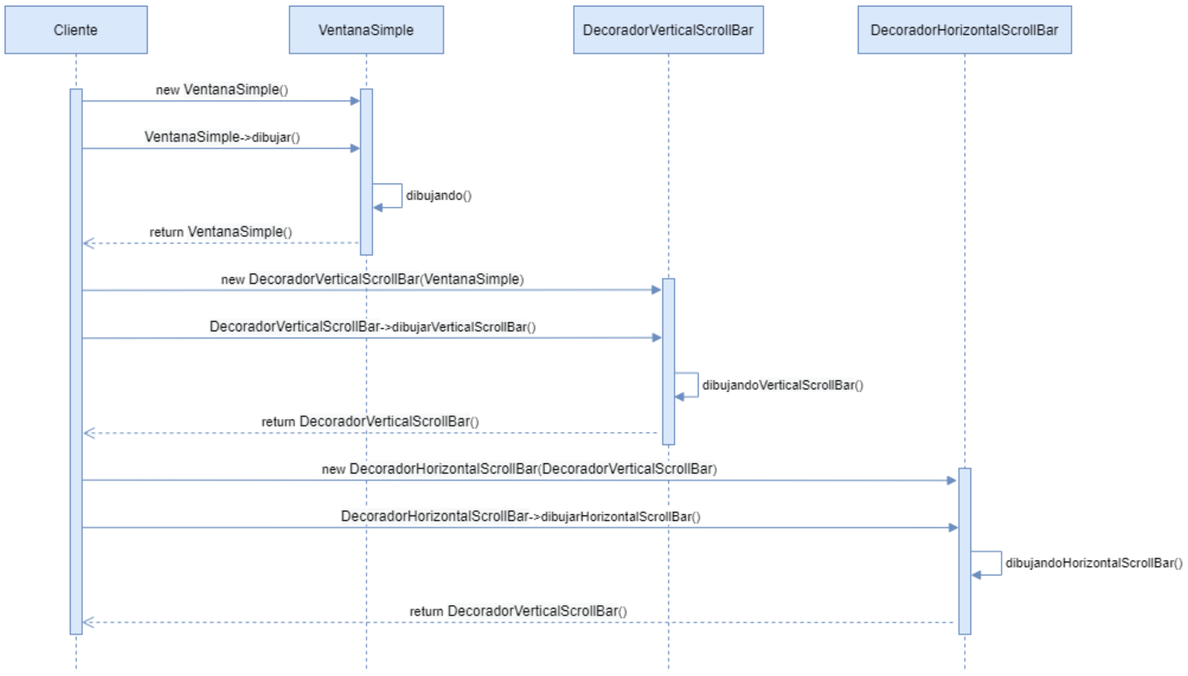
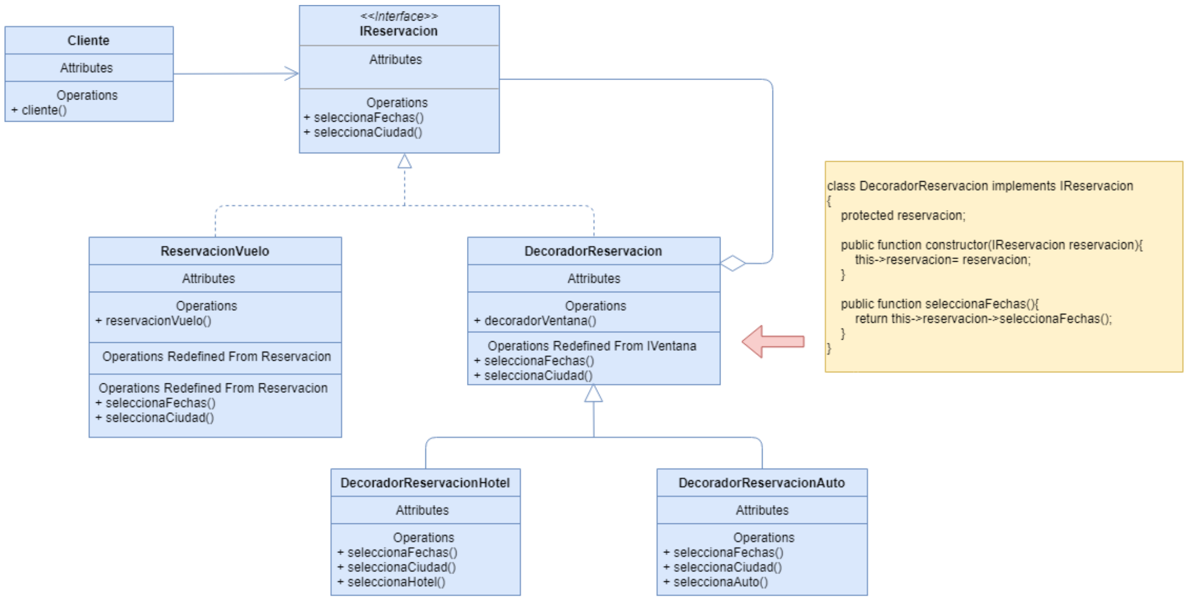

Cliente
VentanaSimple
DecoradorVerticalScrollBar
DecoradorHorizontalScrollBar
new VentanaSimple()
VentanaSimple->dibujar()
dibujando()
return VentanaSimple()
new DecoradorVerticalScrollBar(VentanaSimple)
DecoradorVerticalScrollBar->dibujarVerticalScrollBar()
dibujandoVerticalScrollBar()
return DecoradorVerticalScrollBar()
new DecoradorHorizontalScrollBar(DecoradorVerticalScrollBar)
DecoradorHorizontalScrollBar->dibujarHorizontalScrollBar()
dibujandoHorizontalScrollBar()
return DecoradorVerticalScrollBar()

Figure 4: UML Diagram Interface System

Tourist Reservation System



```
class DecoradorReservacion implements IReservacion
{
   protected reservacion;

   public function constructor(IReservacion reservacion){
      this->reservacion= reservacion;
   }

   public function seleccionaFechas(){
      return this->reservacion->seleccionaFechas();
   }
}
```

Figure 5: UML Diagram Tourist Reservation System

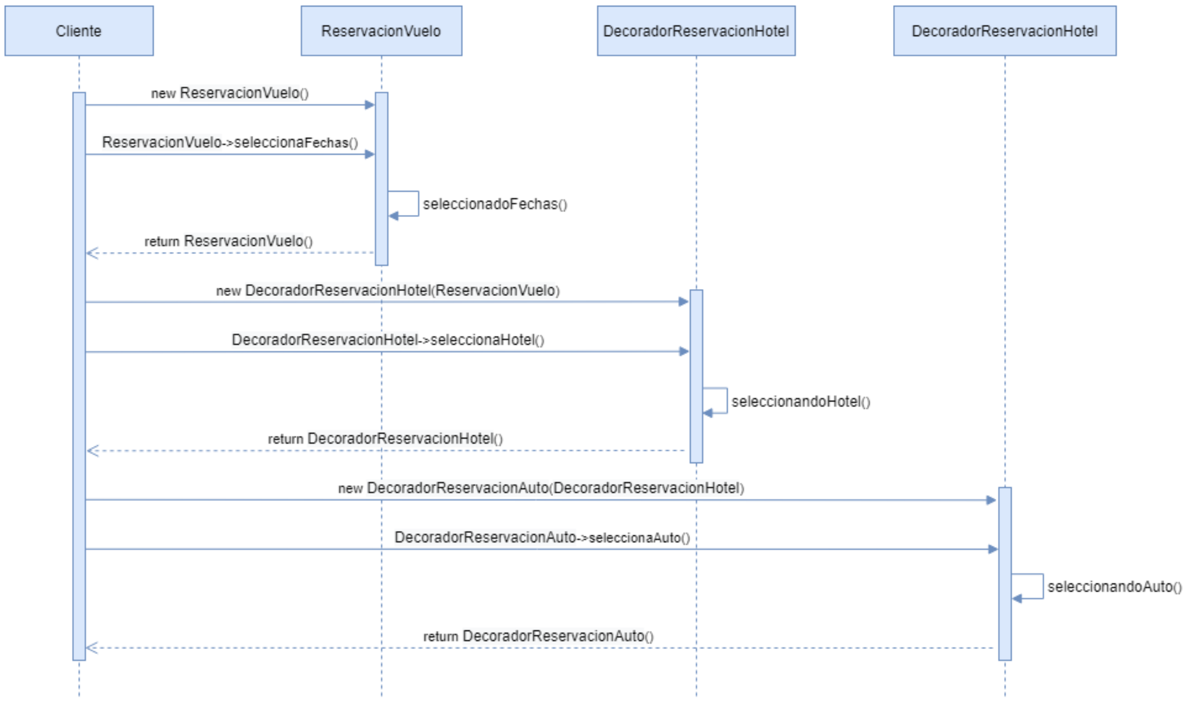

Cliente
ReservacionVuelo
DecoradorReservacionHotel
DecoradorReservacionHotel
new ReservacionVuelo()
ReservacionVuelo->seleccionaFechas()
seleccionadoFechas()
return ReservacionVuelo()
new DecoradorReservacionHotel(ReservacionVuelo)
DecoradorReservacionHotel->seleccionaHotel()
seleccionandoHotel()
return DecoradorReservacionHotel()
new DecoradorReservacionAuto(DecoradorReservacionHotel)
DecoradorReservacionAuto->seleccionaAuto()
seleccionandoAuto()
return DecoradorReservacionAuto()

Figure 6: UML Diagram Tourist Reservation System