



**UNIVERSIDAD DEL CAUCA**  
**FACULTAD DE INGENIERIA ELECTRÓNICA Y TELECOMUNICACIONES**  
**PROGRAMA DE INGENIERÍA DE SISTEMAS**

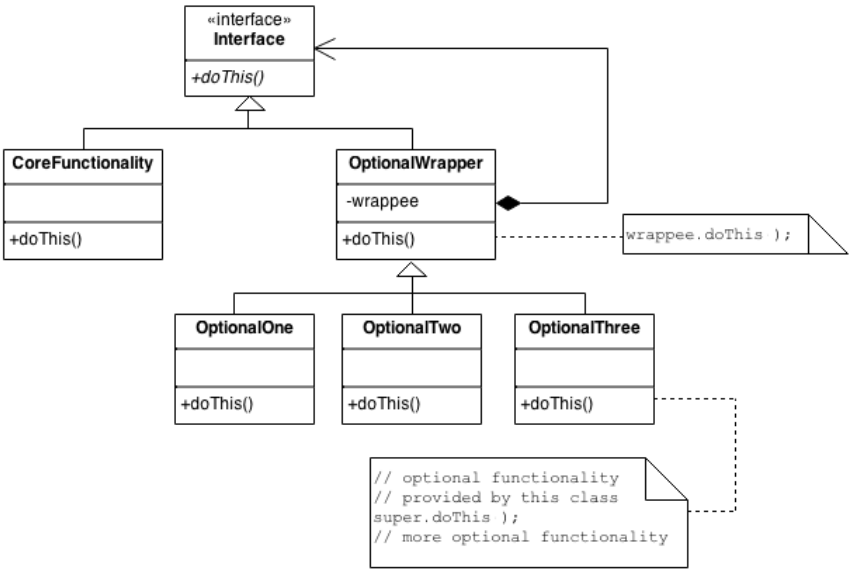
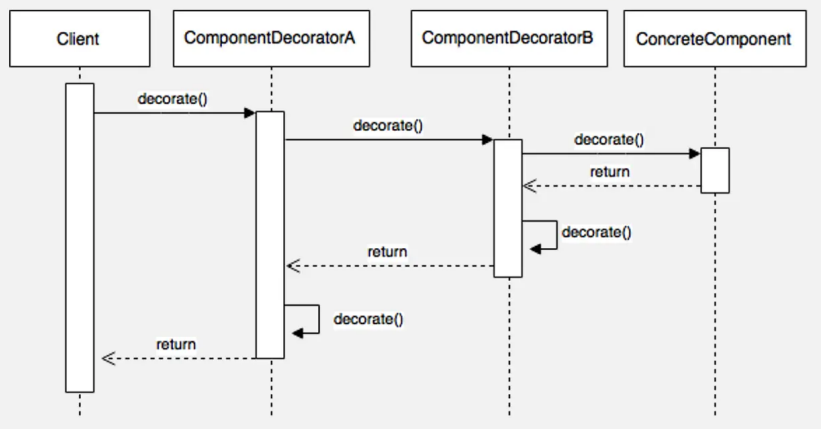
**PRACTICA DE LABORATORIO No. 7**  
**Definición del proyecto de trabajo**

**OBJETIVOS:** Comprender e implementar el patrón estructural Decorator (Decorator).

**ACTIVIDADES:**

1. Desarrolle la siguiente plantilla para el patrón Decorator:

Patrón creacional: <b>Decorator</b>	
Intención	Permite adjuntar responsabilidades adicionales a un objeto dinámicamente. Los decoradores ofrecen una alternativa flexible a la subclasificación para extender la funcionalidad.
Problema que soluciona	Cuando se necesita agregar comportamiento o estado a objetos individuales en tiempo de ejecución, y la herencia no es factible porque es estática y se aplica a toda una clase.
Solución propuesta	La solución consiste en encapsular el objeto original dentro de una interfaz de envoltura abstracta. Tanto los objetos decoradores como el objeto central heredan de esta interfaz abstracta. La composición recursiva se utiliza para permitir un número ilimitado de "capas" de decoradores que se pueden agregar a cada objeto central. Esto permite que las responsabilidades se agreguen a un objeto sin modificar la interfaz del objeto.

<p>Diagrama de clases</p>	 <pre> classDiagram     class Interface {         &lt;&lt;interface&gt;&gt;         +doThis()     }     class CoreFunctionality {         +doThis()     }     class OptionalWrapper {         -wrappee         +doThis()     }     class OptionalOne {         +doThis()     }     class OptionalTwo {         +doThis()     }     class OptionalThree {         +doThis()     }     Interface &lt; -- CoreFunctionality     Interface &lt; -- OptionalWrapper     OptionalWrapper &lt; -- OptionalOne     OptionalWrapper &lt; -- OptionalTwo     OptionalWrapper &lt; -- OptionalThree     OptionalWrapper o--&gt; Interface : wrappee.doThis();     OptionalWrapper o--&gt; OptionalWrapper : wrappee.doThis();     OptionalOne ..&gt; OptionalWrapper : super.doThis();     OptionalTwo ..&gt; OptionalWrapper : super.doThis();     OptionalThree ..&gt; OptionalWrapper : super.doThis(); </pre> <p>The diagram illustrates the Decorator Pattern structure. It features an <b>Interface</b> with a <code>+doThis()</code> method. <b>CoreFunctionality</b> implements the interface. <b>OptionalWrapper</b> also implements the interface and holds a reference to an <b>Interface</b> object (labeled <code>wrappee.doThis();</code>) and another <b>OptionalWrapper</b> object (labeled <code>wrappee.doThis();</code>). Three concrete classes, <b>OptionalOne</b>, <b>OptionalTwo</b>, and <b>OptionalThree</b>, inherit from <b>OptionalWrapper</b>. Each concrete class has a <code>+doThis()</code> method that delegates the call to <code>super.doThis();</code>. A note indicates that the concrete classes provide optional functionality by calling <code>super.doThis();</code> and then adding more optional functionality.</p>
<p>Diagrama de secuencia</p>	 <pre> sequenceDiagram     participant Client     participant ComponentDecoratorA     participant ComponentDecoratorB     participant ConcreteComponent     Client-&gt;&gt;ComponentDecoratorA: decorate()     activate ComponentDecoratorA     ComponentDecoratorA-&gt;&gt;ComponentDecoratorB: decorate()     activate ComponentDecoratorB     ComponentDecoratorB-&gt;&gt;ConcreteComponent: decorate()     activate ConcreteComponent     ConcreteComponent--&gt;&gt;ComponentDecoratorB: return     deactivate ConcreteComponent     ComponentDecoratorB--&gt;&gt;ComponentDecoratorA: return     deactivate ComponentDecoratorB     ComponentDecoratorA--&gt;&gt;Client: return     deactivate ComponentDecoratorA </pre> <p>The sequence diagram shows the runtime interaction. The <b>Client</b> calls <code>decorate()</code> on <b>ComponentDecoratorA</b>. <b>ComponentDecoratorA</b> then calls <code>decorate()</code> on <b>ComponentDecoratorB</b>, which in turn calls <code>decorate()</code> on <b>ConcreteComponent</b>. The return values flow back up the chain: <b>ConcreteComponent</b> returns to <b>ComponentDecoratorB</b>, which returns to <b>ComponentDecoratorA</b>, which finally returns to the <b>Client</b>.</p>
<p>Participantes</p>	<ul style="list-style-type: none"> <li>● Cliente (Client): Quien usa el objeto central y los decoradores.</li> <li>● Componente (Component): La interfaz común para el objeto central y los decoradores.</li> <li>● Componente Concreto (ConcreteComponent): El objeto central al que se pueden agregar decoradores.</li> <li>● Decorador (Decorator): La clase base para los decoradores, que también implementa la interfaz Component.</li> <li>● Decorador Concreto (ConcreteDecorator): Implementaciones concretas de decoradores que añaden funcionalidad adicional.</li> </ul>
<p>Aplicabilidad</p>	<ul style="list-style-type: none"> <li>● Cuando se necesita agregar funcionalidad adicional a objetos individuales de forma dinámica y flexible.</li> <li>● Cuando se quiere evitar la subclasificación excesiva de clases para agregar nuevas funcionalidades.</li> </ul>

	<ul style="list-style-type: none"> <li>● Cuando se desea mantener la interfaz del objeto constante para el cliente mientras se agrega responsabilidad adicional.</li> </ul>
Consecuencias	<ul style="list-style-type: none"> <li>● Permite una extensión flexible de la funcionalidad de objetos.</li> <li>● Permite un diseño modular y de fácil mantenimiento.</li> <li>● Permite agregar o quitar decoradores sin afectar a otros componentes del sistema.</li> <li>● Sin embargo, puede resultar en una cadena de decoradores larga y compleja si se abusa de él.</li> </ul>