|  |  |  |  |
| --- | --- | --- | --- |
| NOMBRE | Diseño de sistemas | | |
| AREA | Tecnologías Aplicadas | REGIMEN | Semestral |
| MODALIDAD | Por examen final | CARGA HORARIA | |  | | --- | | 6 (seis) semanales,  90 totales | |

***Alumno:***

***Curso:***

***Fecha de entrega:***

**Tema: Patrones de diseño**

**Método Decorator**

El patrón decorator permite añadir responsabilidades a objetos concretos de forma dinámica. Los decoradores ofrecen una alternativa más flexible que la herencia para extender las funcionalidades.  
Es conocido como Wrapper (igual que el patrón Adapter). A veces se desea adicionar responsabilidades a un objeto pero no a toda la clase. Las responsabilidades se pueden adicionar por medio de los mecanismos de Herencia, pero este mecanismo no es flexible porque la responsabilidad es adicionada estáticamente. La solución flexible es la de rodear el objeto con otro objeto que es el que adiciona la nueva responsabilidad. Este nuevo objeto es el Decorator.  
  
Este patrón se debe utilizar cuando:

* Hay una necesidad de extender la funcionalidad de una clase, pero no hay razones para extenderlo a través de la herencia.
* Se quiere agregar o quitar dinámicamente la funcionalidad de un objeto.

Dado que este patrón decora un objeto y le agrega funcionalidad, suele ser muy utilizado para adicionar opciones de "embellecimiento" en las interfaces al usuario. Este patrón debe ser utilizado cuando la herencia de clases no es viable o no es útil para agregar funcionalidad. Imaginemos que vamos a comprar una PC de escritorio. Una estándar tiene un precio determinado. Pero si le agregamos otros componentes, por ejemplo, un lector de CD, el precio varía. Si le agregamos un monitor LCD, seguramente también varía el precio. Y con cada componente adicional que le agreguemos al estándar, seguramente el precio cambiará. Este caso, es un caso típico para utilizar el Decorator.  
 **Diagrama UML**

A diagram of a component

Description automatically generated

Component: define la interface de los objetos a los que se les pueden adicionar responsabilidades dinámicamente.  
ComponenteConcreteo: define el objeto al que se le puede adicionar una responsabilidad.  
Decorator: mantiene una referencia al objeto Component y define una interface de acuerdo con la interface de Component.  
DecoratorConcreto: adiciona la responsabilidad al Component.  
  
Decorator propaga los mensajes a su objeto Component. Opcionalmente puede realizar operaciones antes y después de enviar el mensaje.  
  
Ejemplo  
Imaginemos que vendemos automóviles y el cliente puede opcionalmente adicionar ciertos componentes (aire acondicionado, mp3 player, etc). Por cada componente que se adiciona, el precio varía.

A white background with black text

Description automatically generated

A close up of text

Description automatically generated

A screenshot of a computer code

Description automatically generatedA screenshot of a computer code

Description automatically generated

Bien, hasta aquí clases comunes de negoci: una interface que implementa la clase Auto y dos tipos de Auto (Ford y Fiat). Ahora veremos en que consiste el Decorator y los decoradores:

A screenshot of a computer program

Description automatically generated

A screenshot of a computer code

Description automatically generated

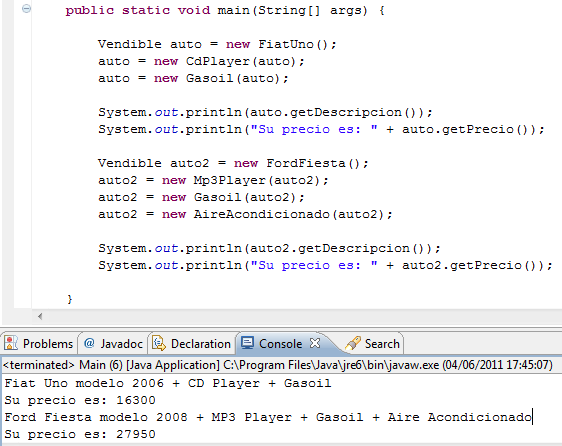
A screen shot of a computer program

Description automatically generated

A screen shot of a computer code

Description automatically generated

Veremos como funciona desde el punto de vista del cliente:



Consecuencias

* Es más flexible que la herencia: utilizando diferentes combinaciones de unos pocos tipos distintos de objetos decorator, se puede crear muchas combinaciones distintas de comportamientos. Para crear esos diferentes tipos de comportamiento con la herencia se requiere que definas muchas clases distintas.
* Evita que las clases altas de la jerarquía estén demasiado cargadas de funcionalidad.
* Un componente y su decorador no son el mismo objeto.
* Provoca la creación de muchos objetos pequeños encadenados, lo que puede llegar a complicar la depuración.
* La flexibilidad de los objetos decorator los hace más propenso a errores que la herencia. Por ejemplo, es posible combinar objetos decorator de diferentes formas que no funcionen, o crear referencias circulares entre los objetos decorator.