***Patrones de Diseño***

**Presentados por:** Mauricio Correa García cc. 8.161.927

Yhors Brayan Bohórquez Rave CC. 1.216.725.076

**Materia:** Taller de Programación

**Patrones de Diseño Estructurales**

**Decorator**

Este patrón de diseño está diseñado para solucionar los problemas más comunes en el momento en que se tiene una jerarquía con subclasificación, pero no puede ser aplicada, en donde se espera un gran impacto en todas las clases de la jerarquía con el fin de poder lograr el comportamiento esperado. El patrón de diseño Decorator permite al usuario añadir nuevas funcionalidades a un objeto existente sin alterar su estructura, mediante la adición de nuevas clases que envuelven a la anterior dándole funcionamiento extra, es decir, se crean varias clases que envolverán las funcionalidades de una previa estructura definida.

Este patrón de diseño se aplica cuando:

* Es necesario añadir responsabilidades a objetos individuales de forma dinámica y transparente, además se pueden revocar responsabilidades antes asignadas a estos objetos.
* Necesitamos extender la funcionalidad de una clase, pero la herencia no es una solución viable.
* Necesitamos extender la funcionalidad de un objeto en tiempo de ejecución e incluso eliminarla si fuera necesario.

Algunas consideraciones positivas y negativas de este patrón:

*POSITIVAS:*

* Es más flexible que la herencia.
* Permite añadir y eliminar responsabilidades en tiempo de ejecución.
* Evita la herencia con muchas clases y la herencia múltiple.
* Limita la responsabilidad de los componentes para evitar clases con excesivas responsabilidades en los niveles superiores de la jerarquía.

*NEGATIVAS:*

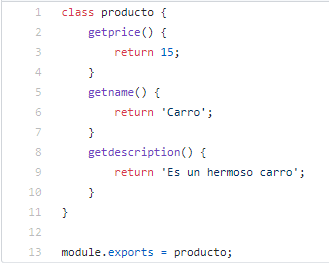
* Genera gran cantidad de objetos pequeños.
* Puede haber problemas con la identidad de los objetos, ya que los decoradores son 'envoltorios' que se comportan como el objeto original, pero la referencia no es la misma; no deberíamos apoyarnos en la identidad a la hora de trabajar con este patrón.

*Ejemplo:*

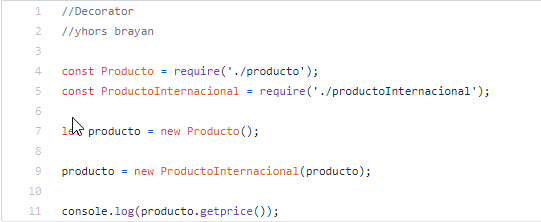
Se tiene una API mediante dos clases que me van a mostrar el precio y la descripción de un automóvil. Mediante la clase **productoInternacional.js** al estamos obteniendo las características principales de la clase **producto.js**



En la clase **producto**.js se detallan el precio del producto, que en este caso es un automóvil.



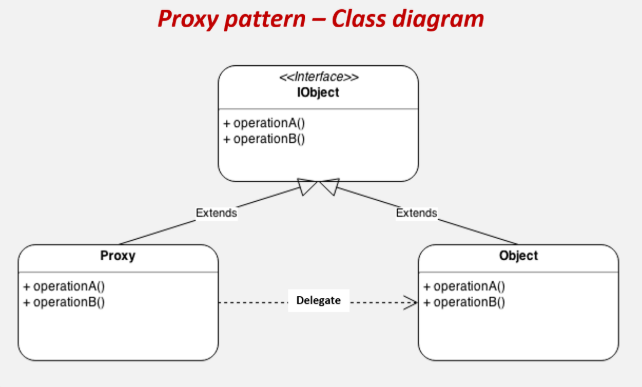
Ahora en la mediante **app.js** que funcionará como Decorator se hará el llamado de las anteriores clases en una sola, aprovechando cada una de sus funcionalidades y mostrará el detalle y el precio del automóvil.



**Proxy**

El patrón Proxy es un patrón estructural que tiene como propósito proporcionar un subrogado o intermediario de un objeto para controlar su acceso, es decir, se centra en la mediación entre un objeto y otro, permitiendo realizar ciertas acciones antes y después de realizar la acción deseada por el usuario.

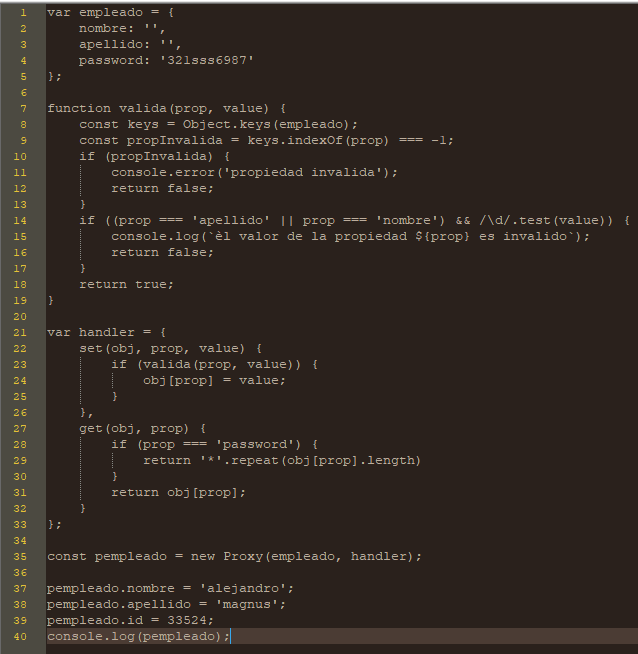
El Proxy se caracteriza por que el cliente ignora totalmente que una mediación se está llevando acabo debido a que el cliente recibe un objeto (**Object**:) idéntico en estructura al esperado, y no es consciente de la implementación tras la interface (**IObject**) ejecutada, de esta manera el cliente interactúa con el Proxy sin saberlo.



El patrón proxy se usa cuando se necesita una referencia a un objeto más flexible o sofisticada que un puntero. Dependiendo de la función que se desea realizar con dicha referencia podemos distinguir diferentes tipos de proxies:

* **proxy remoto:** representante local de un objeto remoto.
* **proxy virtual:** crea objetos costosos bajo demanda (como la clase ImagenProxy en el ejemplo de motivación).
* **proxy de protección:** controla el acceso al objeto original (ejemplo de proxy de protección: [1])
* **proxy de referencia inteligente:** sustituto de un puntero que lleva a cabo operaciones adicionales cuando se accede a un objeto (ej. contar número de referencias al objeto real, cargar un objeto persistente bajo demanda en memoria, control de concurrencia de acceso tal como bloquear el objeto para impedir acceso concurrente, …).

*Ejemplo:*



**Patrones de Diseño Creacionales**

**AbstractFactory**

AbstractFactory (Fábrica Abstracta) es un patrón de diseño para el desarrollo de software. Este provee una interfaz para crear familias de objetos relacionados o dependientes entre ellos sin especificar una clase en concreto. El contexto de este patrón es crear diferentes objetos, todos pertenecientes a la misma familia, y de esta manera poder dar solución al problema de crear diferentes familias de objetos.

AbstractFactory brinda flexibilidad al aislar a las clases concretas, además facilita cambiar las familias de productos, pero también se debe ser cuidadoso en su implementación, pues para agregar nuevos productos se deben modificar tanto las fabricas abstractas como las concretas.

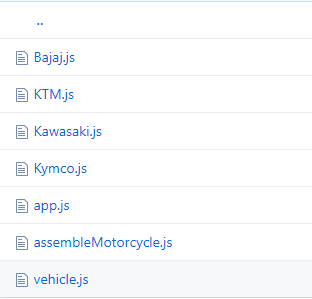
La estructura típica de este patrón está dada por:

* **Cliente:** La clase que llamará a la factoría adecuada ya que necesita crear uno de los objetos que provee la factoría, es decir, Cliente lo que quiere es obtener una instancia de alguno objeto.
* **AbstractFactory:** Es la definición de las interfaces de las factorías, generando un método para la obtención de cada objeto que pueda crear.
* **Factorías Concretas:** Estas son las diferentes familias de productos. Provee de la instancia concreta de la que se encarga de crear.
* **Producto abstracto:** Definición de las interfaces para la familia de productos genéricos. El cliente trabajará directamente sobre esta interfaz, que será implementada por los diferentes productos concretos.
* **Producto concreto:** Implementación de los diferentes objetos.

*Ejemplo:*

En este ejemplo, se presenta un tipo de objeto vehicle.js que detallara las posibles características que tienen las motocicletas, adicionalmente a ello se tienen cada una de las clases (Bajaj.js, Kawasaky.js, KTM.js, Kymco.js) que contienen las información de cada tipo de motocicleta y que conforman el inventario del objeto vehicle.js, por otra parte tendremos la clase assembleMotorcycle.js que se encargara de hacer un llamado al objeto vehicle.js para que le facilite las características de las clases con los tipos de motocicletas, y para finalizar la clase app.js que hará el correspondiente llamado a la factoria creada para que trabaje con el objeto vehicle.js.

El código del ejemplo se mostrará en la dirección del github. https://github.com/Ybrayan/Actividad4



Familia 1

Familia 2

Familia 3

Cliente

***Patrón de Diseño (Creacionales) Singleton***

El patrón singleton consiste en crear una instancia de un objeto y solo una, para toda nuestra aplicación. Sería como una especie de variable global que almacena nuestro objeto.

Este patrón recibe su nombre debido a que sólo se puede tener una única instancia para toda la aplicación de una determinada clase, esto se logra restringiendo la libre creación de instancias de esta clase mediante el operador new e imponiendo un constructor privado y un método estático para poder obtener la instancia.

Los componentes que conforman el patrón son los siguientes:

* **Client:** Componente que desea obtener una instancia de la clase Singleton.
* **Singleton:** Clase que implementa el patrón Singleton, de la cual únicamente se podrá tener una instancia durante toda la vida de la aplicación.

*Ejemplo:*

En el siguiente ejemplo se tiene la clase office con su respectivo constructor el cual contiene las características de un empleado de oficina (nombre y tipo); se hace la instancia de dos objetos de tipo oficina y haciendo el respectivo llamado a la clase, devolviendo el nombre y tipo de empleado para cada objeto.

