**Cuestiones tema 6**

1. **Consideremos los siguientes patrones de diseño: Adaptador, Decorador, Compuesto y Representante. Identificar las principales semejanzas y diferencias entre cada dos ellos.**

**Adaptador - Decorador: ​**Ambos patrones tienen propósito de diseño estructural, además de que ambos trabajan con interfaces. Sin embargo, el patrón decorador únicamente lo podemos utilizar cuando hablamos de objetos, mientras que el adaptador sirve tanto para objetos como para clases. Además el patrón Adaptador no se añaden funcionalidades mientras que en el Decorador si. Adaptador, adapta la interfaz de una clase a la interfaz que necesita su cliente, mientras que el decorador dota dinámicamente de funcionales a los objetos mediante composición.

**Adaptador - Compuesto:** ​Ambos patrones tienen propósito estructural, y ambos trabajan con interfaces para realizar su función. El patrón Compuesto solamente sirve para objetos, y el patrón Adaptador tanto para objetos como para clases. Son dos patrones completamente diferentes, la finalidad del patrón Adaptador “permitir la comunicación” entre dos clases cuyas interfaces no son compatibles y la finalidad del patrón Compuesto es combinar y representar múltiples objetos como si fuese un único objeto y por tanto su finalidad es su diferencia.

**Adaptador - Representante**​: Ambos son patrones estructurales. El patrón Adaptador y el patrón Representante se asimilan en que ambos proporcionan una interfaz para redirigir la petición del cliente al verdadero sujeto que la ejecuta. En cambio, se distinguen en que la interfaz del patrón Adaptador es diferente al objeto que adapta, mientras que el Representante tiene la misma interfaz.

**Decorador - Compuesto:** Ambos patrones no tienen una jerarquía de clases muy compleja ya que se relacionan mediante composición, también utilizan la recursividad ambos a la hora de instanciar objetos de manera dinámica en tiempo de ejecución. Sin embargo, el objetivo principal del patrón decorador es aumentar la funcionalidad de un componente y el patrón Compuesto intenta tratar los objetos y composiciones de objetos de manera uniforme. Ambos patrones pueden trabajar juntos y es frecuente encontrarse dicha combinación.

**Decorador - Representante:** El patrón decorador añade responsabilidades a un objeto, mientras que el Proxy se encarga de controlar el acceso, o la carga de un fichero “on demand” o incluso hacer de “interfaz” en la comunicación entre el usuario y el objeto real. Por otro lado, el decorador, el componente proporciona parte de la funcionalidad y las clases del decorador de añadir nuevas funcionalidades. Ambos patrones se asemejan en que utilizan la composición y proporcionan una interfaz idéntica para el usuario.

**Compuesto - Representante:** Ambos patrones tratan de igual manera los objetos tratándolos como iguales, sin embargo, el patrón compuesto utiliza la recursividad, mientras que el representante no, además de que su objetivo principal es totalmente distinto, uno trata de igual manera a los objetos y conjuntos de objetos mientras que el otro pretende controlar los accesos directos a un objeto en concreto.

1. **Consideremos los siguientes patrones de diseño de comportamiento: Estrategia, Estado, Iterador, Mediador, Observador, Visitante. Identificar las principales semejanzas y diferencias entre cada dos ellos.**

**Estrategia-Estado:** Se asemejan en que ambas permiten realizar una acción de maneras distintas. La gran diferencia entre ambos es que mientras el patrón Estado permite que un objeto modifique su comportamiento según el estado en el que se encuentra; el patrón Estrategia permite a una aplicación elegir, en tiempo de ejecución, el algoritmo que debe seguir y es independiente del cliente que los utiliza.

**Estrategia-Iterador:** Aunque ambos son patrones de comportamiento en el ámbito de los objetos, sus finalidades son muy distintas. Estrategia agrupa varios tipos de comportamientos mediante una interfaz, mientras que Iterador se basa en recorrer una colección de objetos para poder acceder a cada uno de ellos.

**Estrategia-Mediador:** Son dos patrones que tienen el propósito de comportamiento en el ámbito de los objetos. En el Mediador, que está centrado en la comunicación entre objetos, es este quien decide con quién se establece la conexión, mientras que en Estrategia, es el cliente quién decide qué tipo de comportamiento seguir.

**Estrategia-Observador:** Son dos patrones que tienen el propósito de comportamiento en el ámbito de los objetos. El patrón Estrategia se centra en la implementación de métodos de un objeto, mientras que el patrón Observador no se preocupa de la implementación y solo se encarga de comunicar la actualización del estado del objeto.

**Estrategia-Visitante:** Ambos patrones pueden redefinir sus funciones sin afectar a relaciones externas puesto que son independientes a las clases con las que se relacionan. Sin embargo, el patrón Estrategia define una clase compuesta por algoritmos usados en otras clases, y Visitante accede a otras clases para alterar los objetos de una estructura.

**Estado-Iterador:** Aunque ambos son patrones de comportamiento en el ámbito de los objetos, realmente son bastante diferentes. El patrón Iterador trabaja recorriendo objetos en una estructura de datos, mientras que el patrón Estado trabaja a nivel de implementación permitiendo que un objeto modifique su comportamiento según el estado en el que se encuentra.

**Estado-Mediador:** Ambos patrones se comportan en el ámbito de los objetos. Además, los dos patrones definen interfaces principales. Por otro lado, en el patrón Estado se desarrollan los métodos de un objeto mientras que en el Mediador, lo que se desarrolla es la comunicación entre objetos.

**Estado-Observador:** Ambos giran en torno al concepto de estado de un objeto pero son muy distintos. El patrón Estado trabaja con la implementación de los métodos de un objeto concreto según su estado y el patrón Observador, por otro lado, solo notifica a los objetos de los cambios realizados en el estado de un objeto.

**Estado-Visitante:** A nivel estructural, estos patrones no tienen ninguna similitud. Ambos pueden definir nuevas operaciones sin necesidad de modificar las clases sobre las que operan. El patrón Estado depende del estado en el que se encuentre mientras que el Visitante solo depende de las estructuras que modifica.

**Iterador-Mediador:** Ambos trabajan a nivel de objeto y también tienen el propósito de comportamiento. Son patrones muy diferentes, el patrón Iterador proporciona un método de recorrer todos los elementos de una colección mientras que el patrón Mediador comunica objetos. En el uso del patrón Mediador puede ser útil el uso del patrón Iterador a la hora de buscar un objeto.

**Iterador-Observador:** Ninguno de los dos patrones realiza ningún tipo de modificación en la clase con las que se relaciona. Sin embargo, sus funcionalidades son bien distintas; ya que el Iterador se encarga de recorrer una estructura de datos, mientras que el Observador notifica a las clases dependientes sobre los cambios realizados y estas actúan.

**Iterador-Visitante:** El patrón Iterador accede a un objeto secuencialmente sin exponer su representación. Podría ocultar una lista, matriz o algo similar detrás de un iterador. El patrón Visitante se utiliza para realizar una acción en una estructura de elementos sin cambiar la implementación de estos.

**Mediador-Observador:** El patrón del Mediador se usa cuando se tiene dos subsistemas con alguna dependencia debiéndose uno de ellos a un cambio. Dado que es posible que no desee cambiar el sistema que depende del otro, es posible que se desee introducir un mediador que lo haga. El patrón observador, por otro lado, se usa cuando una clase quiere permitir que otras clases se registren y reciban notificaciones sobre eventos.

**Mediador-Visitante:** Ambos patrones son similares al tratarse de interfaces que comunican clases. Sin embargo, el patrón Mediador comunica objetos mientras que el patrón Visitante ejecuta operaciones sin modificar las clases de los elementos sobre los que opera.

**Observador-Visitante:** Respecto a la estructura son similares puesto que ambos son interfaces que comunican clases. Sin embargo, el patrón Visitante ejecuta operaciones para realizar cambios mientras que el patrón Observador notifica a las clases los cambios que han sucedido.

1. **Describir y poner un ejemplo de uso de los siguientes patrones de diseño:**
2. **Composite View**

Vistas compuestas de múltiples subvistas atómicas. Cada componente debe ser añadido dinámicamente y la distribución de la página debe ser gestionado independientemente del contenido.

Se utiliza en el diseño de páginas web y, por ejemplo, los frameworks Angular y React están basados en este patrón, ya que se utiliza para desarrollar SPAs o *Single Page Applications* cambiando los componentes que se muestran en cada momento en la pantalla.

1. **Front Controller**

El patrón Front Controller (Controlador Frontal) se usa para proporcionar un mecanismo centralizado de manejo de solicitudes, de modo que todas las solicitudes sean manejadas por un solo controlador. Este controlador puede autenticar/autorizar/registrar o rastrear la solicitud y luego pasar las solicitudes a los controladores correspondientes.

* **Front Controller**: Controlador único para todo tipo de solicitudes entrantes a la aplicación (basadas en web/escritorio).
* **Dispatcher**: el Front Controller puede utilizar un transportista que envía la solicitud al controlador específico.
* **View (Vista)**: las Vistas son el objeto por el cual se realizan las solicitudes.

Es utilizado en la compra online a través de páginas web, es decir, en el diseño interno de la página.

1. **Facade**

El patrón Facade oculta las complejidades del sistema y provee una interfaz para el cliente mediante la cual el cliente puede acceder al sistema. Este tipo de patrón de diseño se encuentra bajo un patrón estructural, ya que agrega una interfaz al sistema existente para ocultar las complejidades.

Este patrón implica una clase única que proporciona métodos simplificados requeridos por el cliente y delega las llamadas a los métodos de las clases existentes del sistema.

Por ejemplo, el seguimiento de un pedido comprado por internet cuando el comprador se pone en contacto con el personal de la empresa.

1. **Data Mapper**

Un Data Mapper es una capa de acceso a datos que realiza la transferencia bidireccional de datos entre un almacén de datos persistente (a menudo una base de datos relacional) y una representación de datos en memoria (la capa de dominio). El objetivo del patrón es mantener la representación en memoria y el almacenamiento de datos persistentes independientes entre sí y con el correlacionador de datos.

Una clase que implementa los métodos de insertar, borrar y actualizar los valores de una tabla específica de la base de datos.

1. **Service to Worker**

Patrón asociado al controlador, es responsable del acceso al modelo y maneja la navegación de la capa de presentación. El patrón de Service to work describe una combinación de otros patrones antes descritos. Este patrón describe la combinación de un controlador y un despachador con vistas y ayudantes para gestionar las solicitudes de los clientes y preparar una presentación dinámica como respuesta. Los controladores delegan la recuperación de contenido a los ayudantes, que administran la población del modelo intermedio para la vista. Un despachador es responsable de la administración de vistas y la navegación, y se puede encapsular en un controlador o en un componente por separado.

Un ejemplo son los Service workers que actúan como proxy servers entre las aplicaciones web, el navegador y la red.

1. **Discutir las semejanzas y diferencias entre los patrones de diseño Data Transfer Object y Data Access Object.**

La principal diferencia entre ambos objetos es que un objeto de transferencia de datos (Data Transfer Object, DTO) sólo tiene el comportamiento de almacenar y entregar sus propios datos. El DTO transfiere datos entre clases de un sistema y solo debe tener atributos privados (junto getters y setters) y constructores. Sin embargo, un objeto de acceso a datos (Data Access Object, DAO) encapsula los métodos para guardar y actualizar el determinado almacenamiento de datos que use. Un ejemplo sería el siguiente:

interface BookDTO {

String getISBN();

void setISBN(String ISBN);

}

interface BookDAO {

BookDAO searchByISBN(String ISBN);

void save(BookDAO book);

}