Practica 1- Patrones de diseño

Q1. Consideremos los siguientes patrones de diseño: Adaptador, Decorador y Representante. Identifique las principales semejanzas y diferencias entre cada dos ellos (no es suficiente con definirlos, sino describir explícitamente similitudes y semejanzas concretas).

El primer punto común a tener en cuenta es el hecho de que los tres se encargan de **añadir funcionalidades a alguna clase**, para ello emplean una clase u **objeto mediador**, que maneja el uso de los distintos casos.

Adaptador y Decorador

Puntos en común

Ambas clases son capaces de reciclar código o hacerlo compatible con otra clase, en el caso del adaptador, se descarta la opción de crear una nueva implementación especifica para que sea compatible, creando un objeto que sirva de "traductor" entre dos clases. El decorador es capar de añadir nuevas funcionalidades con comportamientos similares, pero manteniendo todas las anteriores e incluso de encapsular ciertas acciones parecidas en un mismo lugar.

De esta forma (muy parecida a una refactorización) se puede reciclar código con el patrón decorador, introduciendo las funcionalidades en clases que hereden del decorador y posteriormente añadiendo más.

Diferencias

El patrón adaptador es únicamente capaz de adaptar una clase por clase auxiliar, esta se especializa en manejar el objeto para que sea compatible con un componente del sistema, mientras que la clase decorador es capaz de, con una misma clase decorador, implementar mas de un comportamiento distinto.

La cascada de ejecución comienza en sentidos distintos, el patrón adaptador empieza en la clase auxiliar y por llamadas alude a las funcionalidades deseadas(hacia abajo en jerarquía).

En la ejecución de un ejemplo de decorador, se comienza llamando a la funcionalidad específicamente, y ésta recurre a métodos de la clase decoradora (que está por encima en jerarquía), que a su vez llama a ciertos métodos de la clase por encima de ella.

Decorador y Representante

Puntos en común

Usan una clase Padre de la que puede depender más de una clase hija, que guarda la implementación de una funcionalidad alternativa.

En el patrón Representante, se consigue con interfaces, que estandarizan el uso de varios tipos y en el decorador, añadiendo distintos escenarios de ejecución alternativos.

Diferencias

El Decorador se basa en añadir “adornos” a un objeto único e invariable mientras que el Representante puede llegar a implementar comportamientos comunes manejar distintos objetos de clases distintas, esta es la finalidad del patrón Representante

El patrón Representante maneja de forma mas inteligente su estado, restringe el acceso a las clases para asegurar.

Representante y Adaptador

Puntos en común

El patrón adaptador de delegación y el de representante son los mas parecidos, por el hecho de lograr una llamada a la clase adaptada/representada desde el objeto intermediario. Esto crea un flujo de ejecución común, pasar por el objeto auxiliar para realizar una operación que es propia de el objeto que hemos adaptado.

Diferencias

Sin embargo, el patrón de representación puede tener más de una clase representada, mientras que el adaptador, dependiendo de si se toma el camino de delegación o el de herencia, puede adaptar también las subclases del objeto o no, respectivamente.

Además, el patrón representante suele recurrir a las interfaces, así se asegura de que en todas las clases que van a ser manejadas por el proxy entienden ciertos mensajes.

Q2. Consideremos los patrones de diseño de comportamiento Estrategia y Estado. Identifique las principales semejanzas y diferencias entre ellos.

Puntos en común

Misma estructura, usan una clase padre (interfaz o clase abstracta), a partir de la cual surgen las alternativas, encapsuladas en clases.

Las llamadas a esta clase padre (que en realidad albergan objetos del tipo dinámico de las clases hijas) se realizan desde la clase a la que se le quieren dar diferentes comportamientos.

Ambas acaban manejando su comportamiento acorde con las necesidades, creando un estado interno que depende del parámetro que se le quieran pasar al método en la clase a principal.

Diferencias

Estado emplea la **herencia de una clase abstracta**, permitiendo reciclar código de estados anteriores para implementar nuevos estados hijos ligeramente (o no) distintos.

Estrategia opta por el uso de una interfaz, muy parecido a una clase abstracta, pero sin la posibilidad de que nuestra clase maestra pueda heredar comportamientos o ser compatible con subclases como podría hacer la clase abstracta del patrón Estado.

Q3. Consideremos los patrones de diseño de comportamiento Mediador y Observador. Identifique las principales semejanzas y diferencias entre ellos.

Puntos en común

Una clase toma el control, conteniendo los comportamientos de interacción entre las clases, así éstas no deben guardar implementaciones sobre otras clases, lo que brida al código la oportunidad de reusar esas clases para otros tipos de comportamiento si el programador lo deseara.

Diferencias

El patrón observador es más bien un algoritmo pasivo, que se fija en ciertos atributos de la una clase observada y estos cambios detonarán acciones entre clases; pero al final, son las clases las que acaban provocando ese cambio.

En el Mediador, el componente llama a la clase mediadora para adquirir el comportamiento especifico que le da la clase con otras.

Cliente de correo e-look

Una de las funciones de *e-look* es la de permitir visualizar (método `show()`) los e-mails de un *`mailbox`* ordenados por diferentes criterios (from, subject, date, priority, ...). Para ello, hemos implementado en la clase `Mailbox` una operación `sort()`, utilizando el llamado método de la burbuja que, como vemos, se basa a su vez en una operación `before()`. Como es lógico, el resultado que devuelva `before` dependerá del criterio de ordenación elegido para visualizar el *`mailbox`* (from, subject, date, priority, ...), por lo que sería necesario parametrizar de alguna manera `sort()` de acuerdo a dicho criterio

Identifique un patrón de diseño que nos permita resolver la situación presentada, permitiendo incluso cambiar de un criterio de ordenación a otro mientras el usuario utiliza e-look, y de forma que sea fácilmente extensible (por ejemplo, para ordenar por otros criterios aparte de los indicados). Mostrar de forma esquemática la implementación en Java del patrón propuesto al problema descrito.

Se han tenido en consideración dos patrones similares para solucionar este problema: el patrón Decorador y el patrón Estado.

Por simpleza, usaremos el patrón Estado. Este patrón puede contener comportamientos para estados distintos; si tomamos cada uno de los criterios de ordenación como un estado por el que pasa mailbox, podemos implementarlo de forma que añadir un nuevo criterio de orden es tan simple como implementar una sola función en una clase aislada.

Para que before() admita más de una alternativa de implementación, se le ha transferido su declaración a la clase abstracta (y a sus hijas) en lugar de encontrarse en MailBox.

De manera que, parametrizando el método sort con una de las hijas de la clase abstracta, podemos pasarle el método before y que este ejecute su algoritmo (este código se muestra en MailBox del diagrama mas abajo)

A continuación, se muestra el diagrama de clases con la distribución escogida para resolver el problema, nótese que Nuevo_criterio alude a todos los demás criterios de ordenación que se quieran agregar.