Equipo: Chilaquiles

Practica 01.

Integrantes:

- Sebastián Salmerón Gómez - 319060661
- Jaime Octavio Delfin Lopez - 318315308
- Alan Ignacio Lopez Carrillo - 420014760

Parte teórica:

Menciona los principios de diseño esenciales del patrón Strategy y Observer. Menciona una desventaja de cada patrón

Strategy:

Strategy es un patrón de diseño de comportamiento que te permite definir una familia de algoritmos, colocar cada uno de ellos en una clase separada y hacer sus objetos intercambiables.

El patrón Strategy sugiere que se tome esa clase que hace algo específico de muchas formas diferentes y se extraigan todos esos algoritmos para colocarlos en clases separadas llamadas *estrategias*.

La clase original se llamará *contexto* y debe tener un campo para almacenar una referencia a una de las estrategias. El contexto delega el trabajo a un objeto de estrategia vinculado en lugar de ejecutarlo por su cuenta.

La clase contexto no es responsable de seleccionar un algoritmo adecuado para la tarea, en lugar de eso, se pasa la estrategia deseada a la clase contexto y funciona con todas las estrategias a través de la misma interfaz genérica, que sólo expone un único método para disparar el algoritmo encapsulado dentro de la estrategia seleccionada.

De esta forma, el contexto se vuelve independiente de las estrategias concretas, así que puedes añadir nuevos algoritmos o modificar los existentes sin cambiar el código de la clase contexto o de otras estrategias.

¿Cómo implementarlo?:

1. En la clase contexto, identifica un algoritmo que tienda a sufrir cambios frecuentes. También puede ser un enorme condicional que seleccione y ejecute una variante del mismo algoritmo durante el tiempo de ejecución.

2. Declara la interfaz estrategia común a todas las variantes del algoritmo.

3. Uno a uno, se extraen todos los algoritmos y ponlos en sus propias clases. Todas deben implementar la misma interfaz estrategia.

4. En la clase contexto, se añade un campo para almacenar una referencia a un objeto de estrategia. Proporciona un modificador *set* para sustituir valores de ese campo. La clase contexto debe trabajar con el objeto de estrategia únicamente a través de la interfaz estrategia. La clase contexto puede definir una interfaz que permita a la estrategia acceder a sus datos.

5. Los clientes de la clase contexto deben asociarla con una estrategia adecuada que coincida con la forma en la que esperan que la clase contexto realice su trabajo principal.

Desventaja:

Si sólo tienes un par de algoritmos que raramente cambian, no hay una razón real para complicar el programa en exceso con nuevas clases e interfaces que vengan con el patrón; igualmente se deben conocer las diferencias entre estrategias para poder seleccionar la adecuada.

Ejemplos de cuándo utilizar Strategy:

- ➔ Cuando quieras utilizar distintas variantes de un algoritmo dentro de un objeto y poder cambiar de un algoritmo a otro durante el tiempo de ejecución.

Este patrón permite alterar indirectamente el comportamiento del objeto durante el tiempo de ejecución asociándolo con distintos subobjetos que pueden realizar subtareas específicas de distintas maneras.

- ➔ Cuando tengas muchas clases similares que sólo se diferencien en la forma en que ejecutan cierto comportamiento.

Este patrón permite extraer el comportamiento variante para ponerlo en una jerarquía de clases separada y combinar las clases originales en una, reduciendo con ello el código duplicado.

- ➔ Cuando tu clase tenga un enorme operador condicional que cambie entre distintas variantes del mismo algoritmo.

Este patrón permite suprimir dicho condicional extrayendo todos los algoritmos para ponerlos en clases separadas, las cuales implementan la misma interfaz. El objeto original delega la ejecución a uno de esos objetos, en lugar de implementar todas las variantes del algoritmo.

Observer:

Observer es un patrón de diseño de comportamiento que te permite definir un mecanismo de suscripción para notificar a varios objetos sobre cualquier evento que le suceda al objeto que están observando.

El objeto que tiene un estado interesante suele denominarse *sujeto* o *notificador* ya que

igualmente va a notificar a otros objetos los cambios en su estado, en otras ocasiones también se le conoce como *publicador*. El resto de los objetos que quieren conocer los cambios en el estado del notificador, se denominan *suscriptores*.

El patrón Observer sugiere que añadas un mecanismo de suscripción a la clase notificadora para que los objetos individuales puedan suscribirse o cancelar su suscripción a un flujo de eventos que proviene de esa notificadora. Este mecanismo consiste en:

1) un campo matriz para almacenar una lista de referencias a objetos suscriptores y
2) varios métodos públicos que permiten añadir suscriptores y eliminarlos de esa lista.

Cuando le sucede un evento importante al notificador, recorre sus suscriptores y llama al método de notificación específico de sus objetos.

Es fundamental que todos los suscriptores implementen la misma interfaz y que el notificador únicamente se comunique con ellos a través de esa interfaz. Esta interfaz debe declarar el método de notificación junto con un grupo de parámetros que el notificador puede utilizar para pasar cierta información contextual con la notificación.

Si la aplicación tiene varios tipos diferentes de notificadores y se quiere hacer a los suscriptores compatibles con todos ellos, se puede ir más allá y hacer que todos los notificadores sigan la misma interfaz. Esta interfaz sólo tendrá que describir algunos métodos de suscripción. La interfaz permitirá a los suscriptores observar los estados de los notificadores sin acoplarse a sus clases concretas.

La lista de suscriptores se compila dinámicamente: los objetos pueden empezar o parar de escuchar notificaciones durante el tiempo de ejecución, dependiendo del comportamiento que se desee para la aplicación.

En esta implementación, la clase editora no mantiene la lista de suscripción por sí misma. Delega este trabajo al objeto ayudante especial dedicado justo a eso. Se puede actualizar ese objeto para que sirva como despachador centralizado de eventos, dejando que cualquier objeto actúe como notificador.

Añadir nuevos suscriptores al programa no requiere cambios en clases notificadoras existentes, siempre y cuando trabajen con todos los suscriptores a través de la misma interfaz.

¿Cómo implementarlo?:

1. Intenta dividir la lógica de negocio en dos partes: la funcionalidad central, independiente del resto de código, actuará como notificador; el resto se convertirá en un grupo de clases suscriptoras.

2. Declara la interfaz suscriptora. Como mínimo, deberá declarar un único método *actualizar*.

3. Declara la interfaz notificadora y describe un par de métodos para añadir y eliminar de la lista un objeto suscriptor. Los notificadores deben trabajar con suscriptores únicamente a través de la interfaz suscriptora.

4. Decide dónde colocar la lista de suscripción y la implementación de métodos de suscripción. Normalmente, este código tiene el mismo aspecto para todos los tipos de notificadores, por lo que el lugar obvio para colocarlo es en una clase abstracta derivada directamente de la interfaz notificadora. Los notificadores concretos extienden esa clase, heredando el comportamiento de suscripción; sin embargo, si estás aplicando el patrón a una jerarquía de clases existentes, considera una solución basada en la composición: coloca la lógica de la suscripción en un objeto separado y haz que todos los notificadores reales la utilicen.

5. Crea clases notificadoras concretas. Cada vez que suceda algo importante dentro de una notificadora, deberá notificar a todos sus suscriptores.

6. Implementa los métodos de notificación de actualizaciones en clases suscriptoras concretas. La mayoría de las suscriptoras necesitarán cierta información de contexto sobre el evento, que puede pasarse como argumento del método de notificación; otra opción es que al recibir una notificación, el suscriptor puede extraer la información directamente de ella. En este caso, el notificador debe pasarse a sí mismo a través del método de actualización. La opción menos flexible es vincular un notificador con el suscriptor de forma permanente a través del constructor.

7. El cliente debe crear todos los suscriptores necesarios y registrarlos con los notificadores adecuados.

Desventaja:

Los suscriptores son notificados en un orden aleatorio.

Ejemplos de cuándo usar Observer:

- ➔ Cuando los cambios en el estado de un objeto puedan necesitar cambiar otros objetos y el grupo de objetos sea desconocido de antemano o cambie dinámicamente.

Se puede experimentar este problema a menudo al trabajar con clases de la interfaz gráfica de usuario. Este patrón permite que cualquier objeto que implemente la interfaz suscriptora pueda suscribirse a notificaciones de eventos en objetos notificadores. Se puede añadir el mecanismo de suscripción a tus botones, permitiendo a los clientes acoplar su código personalizado a través de clases suscriptoras personalizadas.

- ➔ Cuando algunos objetos de tu aplicación deban observar a otros, pero sólo durante un tiempo limitado o en casos específicos.

La lista de suscripción es dinámica, por lo que los suscriptores pueden unirse o abandonar la lista cuando lo deseen.

Comentarios: Es una práctica algo pesada para entregar en una semana, en especial porque vas conociendo con quien estás trabajando, esto puede generar varios contratiempos en la organización y distribución de trabajo. La única “ventaja” es que te obliga a repasar e

investigar sobre las dudas que tienes para poder entregar la practica lo mejor posible.