|  |  |  |
| --- | --- | --- |
|  | Decorator | Strategy |
| Facilidad para agregar nuevos criterios de filtro | Mayor  Lo que nos permite esta técnica es ir agregando comportamiento a medida que lo voy necesitando y además sin cambiar ninguna línea del código original. | Menor  Usar esta estrategia implicaría que tenga que modificar el código original a medida que voy agregando comportamiento. Si bien existen muchas alternativas para trabajarlo, cada una presenta una desventaja. Quizás la solución más viable es la ya conocida colección de objetos. |
| Posibilidad de eliminar filtros dinámicamente | Menor  Para eliminar los filtros en este caso es más complicado ya que “desdecorar” el componente requiere conocer al decorator específico (la clase), por eso la estrategia consiste en tener el componente original y agregarle en runtime los decoradores que yo quiero. | Mayor  La implementación de colecciones para guardar los comportamientos mencionada anteriormente haría fácil la supresión de alguno de los filtros. Bastaría el uso del método remove(objeto) y enviarle el filtro que quiero eliminar. |
| posibilidad de intercambiar el orden en que se aplican los filtros o el posterior manejo de resultados | Mayor  Si todos los métodos tendrían que ejecutarse, el orden no sería problema alguno para obtener el resultado esperado ya que yo mismo me encargo de armar la secuencia en el momento. | Menor  Si se requiere de un orden de ejecución para el uso de los filtros, en el uso de las colecciones uno ha de ser muy cauteloso a la hora de agregar los mismos ya que de esto dependerá no sólo su correcto funcionamiento, sino el resultado esperado. |
| simplicidad | Menor  Basta con el ver como queda confeccionado el diagrama de clases, si bien en un principio puede parecer entendible (ya que no presenta mucha complejidad), al momento de escribir el código se torna complicado y abstracto a la mente para acomodar todas las ideas que este concepto encierra. | Mayor  Es bastante más legible que el uso de decorator, ya que la perspectiva del diseño es más simple y se acomoda a la forma en la que el ser humano está acostumbrado a pensar, desde el objeto de la clase hacia adentro y no agregando “pieles” desde el objeto hacia afuera. |
| cohesión o cuántos objetivos resuelve el componente desarrollado | Menor  La desventaja de esta solución es que me obliga a respetar la misma interfaz que el objeto decorado y por ende tener que implementar todos los métodos obligatoriamente aunque algunos no sean usados. Un posible arreglo puede ser delegando la responsabilidad a la clase que si le corresponde. | Mayor  Existe una mayor libertad para diseñar las clases con las interfaces o superclases correspondientes, sin la necesidad de agregar comportamientos o asignar responsabilidades que no les correspondan. |
| mantenibilidad | Mayor  Desde el punto de vista de costos, es mucho más mantenible ya que no requiere que modifique sobre lo ya trabajado y le pueda dar una continuidad sin tener que comprometer el resto del código o hacerlo pero en menor escala. También es extensible por los motivos ya mencionados. | Menor  En base a los ejemplos ya presentados, podemos observar que esta alternativa requiere de un acoplamiento en algunos casos (en menor o mayor medida, pero siempre trataremos de que no exista), es decir que unas clases dependan de otras y por ende tener que modificarlas si así lo requiere nuestro nuevo diseño. |