**PATRONES DE DISEÑO**

Solución estándar a un problema común.

Diseñar algo consiste en tomar una decisión de diseño.

No tienen en cuenta la estructura de datos o los diseños específicos de dominio.

En cada patrón tenemos las clases y las instancias que participan, la distribución de responsabilidades

Dos familias de patrones:

- GRASP

- GoF

**-- PATRONES GoF --**

Pueden resolver tres tipos de problemas, los que resuelven el dónde y cómo creamos objetos que necesitamos, los que resuelven problemas respecto a la estructura de la app, y los que definen el comportamiento y la forma en la que determinadas clases y objetos se relacionan.

Se utilizan estos patrones para hacer que nuestra app sea más fácil de modificar.

Para nuestros proyectos los más importantes van a ser:

- Memento

- Estrategia

- Patrón observador, en conjunto con el modelo vista-controlador

**-- PATRON MEMENTO --**

Se utiliza para guardar el estado de un objeto (se parece al patrón comando). Nos sirve "para hacer un deshacer" y sobre todo para guardar nuestra partida.

Si queremos guardar el estado de un objeto o varios, lo que se hace es coger una clase para que guarde ese estado, pedimos la información que queremos guardar al objeto y posteriormente lo guardamos.

Si nosotros hacemos esto, de alguna manera "violamos" el principio de encapsulación.

Para evitar esto utilizamos el patrón Memento.

Creamos un objeto "Originator" y le añadimos un método "createMemento" (como si fuese un guardar), entonces se crea un objeto Memento y ahí es donde guardamos el estado de nuestro objeto con un método setState() del objeto Memento y así no "enseño a nadie" mi estado, solo al "Memento" que es como si fuese "parte de mi".

Una vez creado Memento con el estado que quiero guardar se origina un objeto "careTaker" que es un creador de mementos y que sirve para guardar diferentes estados, para poder guardar varias partidas.

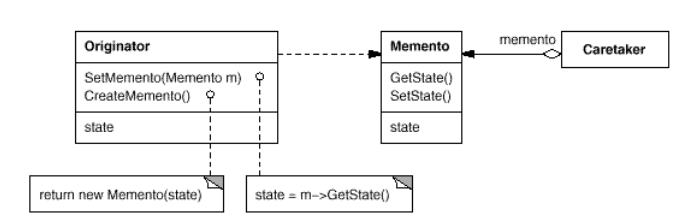
Para recuperar una partida el objeto careTaker llama al originador y le utiliza el método setMemento(), es decir recupera un estado en concreto, entonces el originador coge el Memento, le hace un getState() y los traslada a su propio estado.

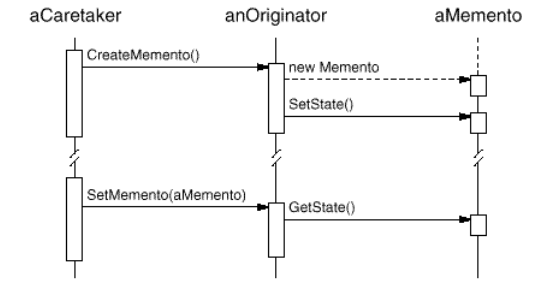
El careTaker suele ser un objeto juego o un objeto partida.

Para guardar varios objetos habría que tener un Memento separado de cada objeto que queremos guardar y un careTaker general parea todos.

- Ventajas: mantenemos la encapsulación.

- Inconvenientes: su uso puede conllevar un pequeño sobrecoste.



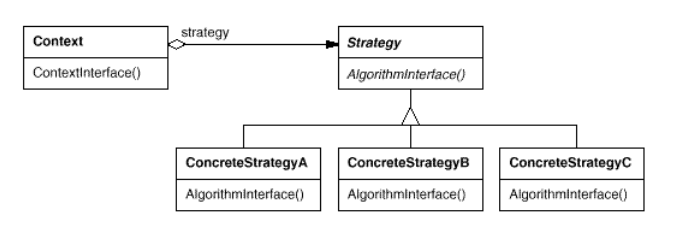


**-- PATRON ESTRATEGIA --**

El código que queremos refactorizar consistiría en un algoritmo con bastantes if ... else.

Los utilizaremos para crear las estrategias de juego de la CPU en nuestra app.

La estructura del patrón estrategia seria así:



La estrategia no es más que una interfaz java.

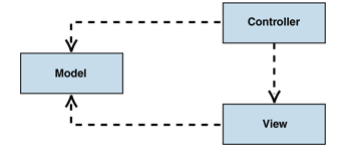
- Ventajas: quitarnos estructuras largas en forma de if... else, es una alternativa a la herencia, permite familia de algoritmos reaccionados (en el caso del ajedrez, se presta muy bien al crear muchos tipos de estrategia para jugar, que nos permite hacer poca exploración).

- Inconvenientes: los clientes (CPU) deben conocer las estrategias, aunque en nuestro caso utilizaremos otros patrones de creación para que esto no suceda, mayor utilización de objetos y aumento del coste.

**-- PATRÓN MODELO-VISTA-CONTROLADOR --**

Sirve para separar la parte de modelo o de lógica de las entradas y salidas de datos.

Tiene una estructura así:

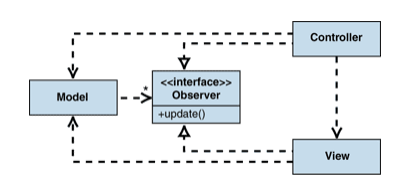


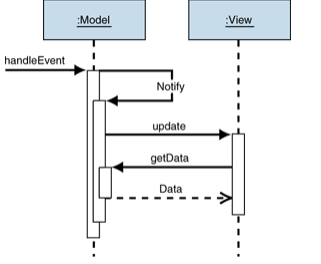
Ejemplo: en hundir la flota si quieres disparar a una casilla, pulsamos en una casilla, el controlador coge esa pulsación de la casilla, informa al modelo del disparo e informa a la vista por si hay que actualizar algo en ella. Luego el modelo procesa el disparo y la vista le pide al modelo la actualización que ha habido con respecto a esto.

El modelo es independiente del controlador y de la vista.

Es posible que tengamos más de un controlador y más de una vista, entonces puede que todas las vistas tengan que actualizarse en función del controlador o los controladores, y a la vez que esto produzca cambios en el modelo que se tenga que actualizar en varias vistas también.

Entonces el funcionamiento básico de este patrón es este:





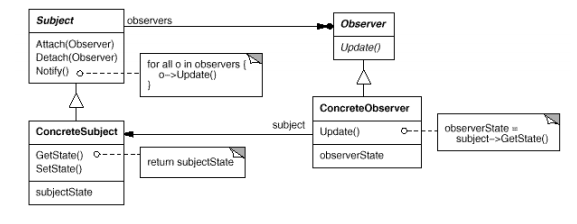
De cara a nuestra app, esto nos sirve en la parte multijugador y el la de multijugador on line.

**-- PATRÓN OBSERVADOR –**

Sirve para notificar el cambo de estado de un objeto determinado a los diferentes objetos que necesitan saber esta información.

Es muy habitual cuando se produce un cambio en el modelo de la app y queremos que se refleje en las vistas.

La estructura seria así:



Cuando cambia el objeto el programa entra en el método notify() y es ahí donde actualiza a todos los observadores ( update() ). Como consecuencia de esto, irán al ConcreteSubjet y harán un getState() para recuperar el nuevo estado y actualizarlo en consecuencia.

Esto se utiliza mucho cuando tenemos interfaces graficas que tienen que cambiar en función de un cambio en el modelo.

Ejemplo: en un periódico on line, pueden colgar una nueva noticia cada poco tiempo, por eso lo que hacen es refrescarse cada cierto tiempo para actualizarse. Para ello hay que ir preguntando todo el rato si hay algún cambio. Esto puede originar una sobrecarga si preguntas cada poco tiempo y si no preguntas con suficiente frecuencia puede suceder que te saltes cambios y que se sobrescriban. Para esto se utiliza este patrón, para que se notifique cada vez que haya un cambio sin sobrecarga ni sobrescritura.