Patrones de Diseño de Comportamiento

**2014**

Alvaro Javier Martinez Cruz

Personal

30/09/2014



Tabla de contenido

No se encontraron elementos de tabla de contenido.**1**

Escribir el título del capítulo (nivel 2)2

Escribir el título del capítulo (nivel 3)3

**Escribir el título del capítulo (nivel 1)4**

Escribir el título del capítulo (nivel 2)5

Escribir el título del capítulo (nivel 3)6

INTRODUCCIÓN

Los Patrones de Diseño Creacionales, encapsulan la lógica para la creación de instancias de objetos.

Estos pueden ser divididos en patrones de creación de clases y patrones de creación de objetos. Mientras que los primeros usan la herencia de manera efectiva en el proceso de la instanciación, los segundos usan la delegación para hacer su trabajo.

Las diferentes opciones de patrones que tenemos son:

* Memento

**MEMENTO**

**Intención**

Sin violar el encapsulamiento, captura y externaliza el estado interno de un objeto para que el objeto pueda ser devuelto a dicho estado posteriormente.

Una *cookie* mágica que encapsula un ”punto de control”.

Proporciona la capacidad de deshacer el estado completo de un objeto.

**Ejemplo del problema**

Se necesita restaurar un objeto a su estado previo (por ejemplo: operaciones de tipo “deshacer” o “rollback”).

**Discusión**

Un cliente solicita un *Memento* de un objeto cuando se necesita un punto de control del estado de dicho objeto. El objeto inicializa al *Memento* con una caracterización de su estado. El cliente es el “guardián” del *Memento*, pero sólo el objeto original puede almacenar y recuperar información del *Memento (* el *Memento* es “opaco” para el cliente y todos los demás objetos). Si el cliente más adelante necesita realizar un rollback sobre el estado del objeto origen, devuelve *Memento* al objeto origen para que éste restaure su estado.

Una capacidad ilimitada de “deshacer” y “rehacer” puede ser fácilmente implementada con una pila de objetos *Command* y una pila de objetos *Memento*.

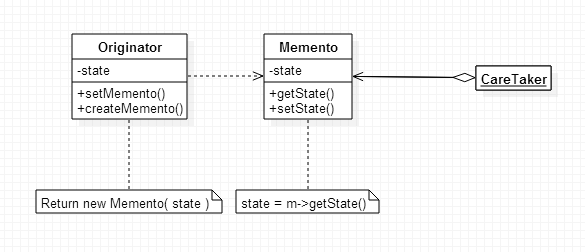
El patrón de diseño *Memento* define tres roles diferentes:

1.- **Originador**: Es el objeto que sabe cómo resguardarse a sí mismo.

2.- **Guardián:** Es el objeto que conoce por qué y cuándo el *Originador* necesita almacenar y restaurarse a sí mismo.

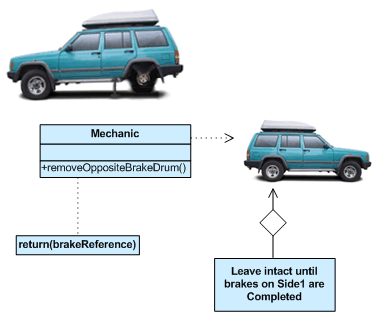
3.- **Memento:** La “caja” de almacenamiento que es utilizado como almacenamiento por el *Originador* y cuidado por el *Guardián*.

**Estructura**

****

**Ejemplo**

El patrón de diseño *Memento* captura y exterioriza el estado de un objeto para que el mismo pueda ser restaurado a dicho estado. Este patrón es común entre los mecanismos de reparación “hágalo usted mismo” de frenos de tambor de autos. Los tambores son removidos de ambos lados, exponiendo ambos frenos, el derecho y el izquierdo. Sólo un lado es desarmado y el otro sirve como *Memento* de cómo las partes del freno encajan juntas. Sólo después de haber finalizado el trabajo de un lado, se puede desarmar el otro. Cuando el segundo lado es desarmado, el primero actúa de *Memento*.



**Check list**

1.- Identificar los roles de “guardián” y “originador”

2.- Crear una clase *Memento* y declarar al *Originador* como *friend*

3.- El guardián conoce cuando almacenar un “punto de control” del *Originador*

4.- El *Originador* crea un *Memento* y copia su estado a dicho *Memento*

5.- El guardián conserva al *Memento* pero sin conocer su contenido

6.- El guardián sabe cuando debe volver al *Originador* a su estado anterior

7.- El *Originador* se restablece a sí mismo utilizando el estado almacenado en el *Memento*

**Reglas de oro**

*Command* y *Memento* actúan como *tokens* mágicos que son pasados e invocados posteriormente. En *Command*, el token representa una solicitud; en *Memento* representa el estado interno de un objeto en un momento determinado. El polimorfismo es importante para *Command*, pero no para *Memento*, porque su interfaz es tan reducida que sólo puede ser pasado como valor.

*Command* puede usar a *Memento* para mantener un estado requerido para implementar una operación deshacer/rollback.

*Memento* es usado a menudo en conjunto con *Iterator*. Un objetopuede usar a un *Memento* para capturar el estado de una iteración. El *Iterator* almacena el *Memento* internamente.

**Intención**

Utilizar un *pool* (o agrupamiento) de objetos puede ofrecer un significativo aumento de rendimiento. Es más eficaz en situaciones donde el costo de la inicialización de instanciar una clase es alta, el ritmo de instanciación también es alta, pero el número de instancias en uso en cualquier momento es baja.

**Ejemplo del problema**

Los agrupamientos de objetos (también conocidos como agrupamientos de recursos) se utilizan para gestionar el caché de objetos. Un cliente con acceso a un agrupamiento de objetos puede evitar crear nuevos objetos simplemente pidiendo al agrupamiento uno que haya sido instanciado con anterioridad. En general, la agrupación será cada vez mayor, es decir, el agrupamiento va a crear por sí mismo nuevas instancias si está vacío, o bien, podría tratarse de un pool que restringe la cantidad de objetos creados.

**Discusión**

El Object Pool permite a otros “obtener” objetos de su agrupamiento. Cuando dichos objetos ya no son necesitados por los procesos que los solicitaron con anterioridad, vuelven al agrupamiento para ser reutilizados.

Sin embargo, no es deseable tener que esperar que un objeto en particular sea liberado, por lo que el Object Pool también crea nuevas instancias a medida que son requeridos, pero debe implementar un mecanismo para “limpiar” periódicamente los objetos que no son utilizados.

**Estructura**

La idea general para el patrón ConnectionPool es que si las instancias de una clase pueden ser reutilizadas, se evite crear nuevas instancias de la clase reutilizándolas.

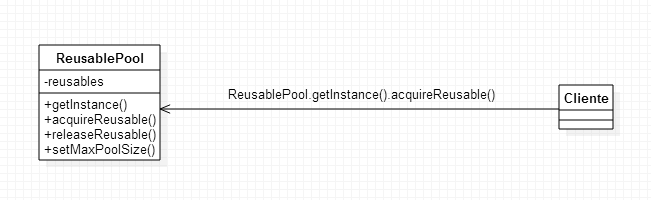
Usualmente es conveniente mantener todos los objetos Reusables, que actualmente no están en uso, en la misma agrupación de objetos para que de esta manera sean administrados por una política coherente. Para lograr esto, la clase es diseñada para ser un Singleton. Su constructor debe ser privado, lo que obliga a otras clases a utilizar su método para obtener una instancia de la clase ReusablePool.

Un objeto Client llama al método acquireReusable del objeto ReusablePool cuando necesita un objeto Reusable.

Un objeto ReusablePool mantiene una colección de objetos Reusable. Este usa la colección para contener un *pool* de objetos Reusable que no están en uso actualmente.

Si existe algún objeto Reusable en la colección cuando el método acquireReusable es llamado, éste quita uno de ellos de la colección y lo devuelve. Si la colección está vacía, entonces dicho método crea un nuevo objeto Reusable si puede. Si no puede crearlo, entonces espera a que uno de los objetos Reusable vuelva a la colección.

En muchas aplicaciones del patrón *Object Pool*, existen razones para limitar el número total de objetos Reusable que pueden existir. En tales casos, el objeto *ReusablePool* que crea objetos Reusable es responsable de no crear más objetos Reusable que el número máximo especificado. Si *ReusablePool* es responsable de limitar el número de objetos que se creará, entocnes la calse *ReusablePool* debe tener un método que permita especificar el número máximo de objetos que pueden ser creados. Dicho método es indicado en la imagen siguiente con el nombre *setMaxPoolSize.*



**Check list**

1.- Crear la clase *ObjectPool* con una colección de objetos privada.

2.- Crear los métodos Obtener(acquire) y Liberar(release) en la clase *ObjectPool*.

3.- Asegurarse de que la clase *ObjectPool* es *singleton*.

**Reglas de Oro**

* El patrón *Factory Method* puede ser usado para encapsular la lógica de creación de objetos. Sin embargo, no los administra después de su creación, el patrón ObjectPool realiza un seguimiento de los objetos que crea.
* Este patrón suele implementarse como *Singleton*.

**FACTORY METHOD**

**Intención**

* Definir una interfaz para crear un objeto, pero dejar que las subclases decidan cuál clase debe instanciar. *Factory Method* permite a una clase diferir la instanciación de subclases
* La definición de un constructor “virtual”
* Que el operador new sea considerado dañino

**Ejemplo del problema**

Un *framework* necesita estandarizar el modelo de arquitectura para una variedad de aplicaciones, pero permitirle a las aplicaciones definir de manera individual sus propios objetos de dominio y asegurar su instanciación

**Discusión**

*Factory Method* sirve para crear objetos así como *Template Method* se utiliza para implementar un algoritmo