Patrones de Diseño Estructurales

**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 Estructurales, 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:

* Flyweight
* Proxy
* Private Class Data
* Facade
* Decorator
* Composite
* Bridge

**FLYWEIGHT**

**Intención**

* Comparte para soportar un gran número de objetos de grano fino de manera eficiente
* La estrategia de GUI Motif de reemplazo de *widgets* pesados por *widgets* livianos

**Ejemplo del problema**

El diseño de los objetos hasta los niveles más bajos de granularidad del sistema proporciona una óptima flexibilidad, pero puede resultar inapropiadamente costosa en términos de rendimiento y uso de memoria.

**Discusión**

El patrón *Flyweight* describe cómo compartir objetos para permitir su uso en granularidad fina pero sin costes excesivos. Cada objeto *Flyweight* es dividido en dos partes: el estado dependiente(extrínseco) y el estado independiente(intrínseco). El estado intrínseco es almacenado (compartido) en el objeto *Flyweight*. El estado extrínseco es almacenado por el objeto cliente y pasado al objeto *Flyweight*, cuando sus operaciones son invocadas.

Un ejemplo de este enfoque serían los widgets *Motif* que han sido rediseñados como gadgets livianos. Mientras que los widgets son “inteligentes” como para valerse por sí mismos; los gadgets existen en una relación de dependencia con su widget controlador padre. Cada controlador proporciona un contexto dependiente de la gestión de eventos, estados y servicios de recursos para sus gadgets *Flyweight* y cada gadget es sólo responsable por el estado de contexto independiente y comportamiento.

**Estructura**

Usar cuando:

* Se requiera exactamente una instancia de una clase y ésta deba ser accesible a los clientes desde un punto de acceso conocido.
* La única instancia debería ser extensible mediante herencia y los clientes deberían ser capaces de utilizar una instancia extendida sin modificar su código.

**Consecuencias**

Algunos de los beneficios de este patrón son:

* Acceso controlado a la única instancia. Puede tener un control estricto sobre cómo y cuando acceden los clientes a la instancia.
* Espacio de nombres reducido. El patrón Singleton es una mejora sobre las variables globales.
* Permite el refinamiento de operaciones y la representación. Se puede crear una subclase Singleton.
* Permite un número de variable de instancias. El patrón hace que sea más fácil cambiar de opinión y permitir más de una instancia de la clase Singleton.
* Más flexible que las operaciones de clase.
* Se crea una única instancia del objeto
* Ahorra el gasto de memoria que se realiza cada vez que se crea una nueva instancia.

Características con las que debe cumplir un objeto para que sea considerado un SINGLETON:

* Constructor privado
* Instancia estática y privada de sí mismo
* Método público getInstance( )

**Posibles problemas**

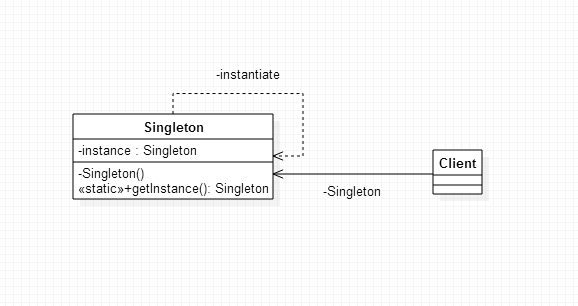
**Singletonitis**

En Refactoring to Patterns[Kerievsky04] se presenta el término **Singletonitis**, rifiréndose a “la adicción al patrón Singleton”. Este término aparece en la motivación del refactoring “Inline Singleton”, cuyo objetivo es remover los Singletons innecesarios en una aplicación. Dado que el Singleton es quizá el patrón más sencillo del GoF a veces es sobreutilizado y muchas veces en forma incorrecta.

¿Esto quiere decir que los Singletons son malos y no hay que usarlos?

Definitivamente no. Pero como todo, debe utilizarse en su justo medida y en el contexto adecuado.

**Estructura**



**OBJECT POOL**

**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.

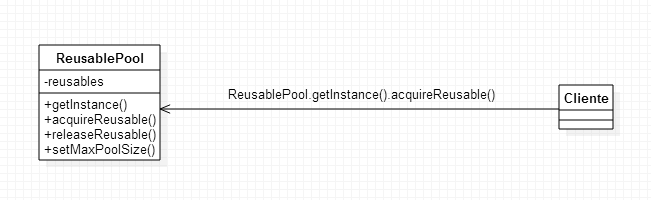
Usualmente es conveniente mantener todos los objetos Reusables, que actualmente no están en uso, en la misma agrupación de objetos para que de esta manera sean administrados por una política coherente. Para lograr esto, la clase es diseñada para ser un Singleton. Su constructor debe ser privado, lo que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