Patrones de Diseño: Patrones de Creación. Tema 3-6: Singleton

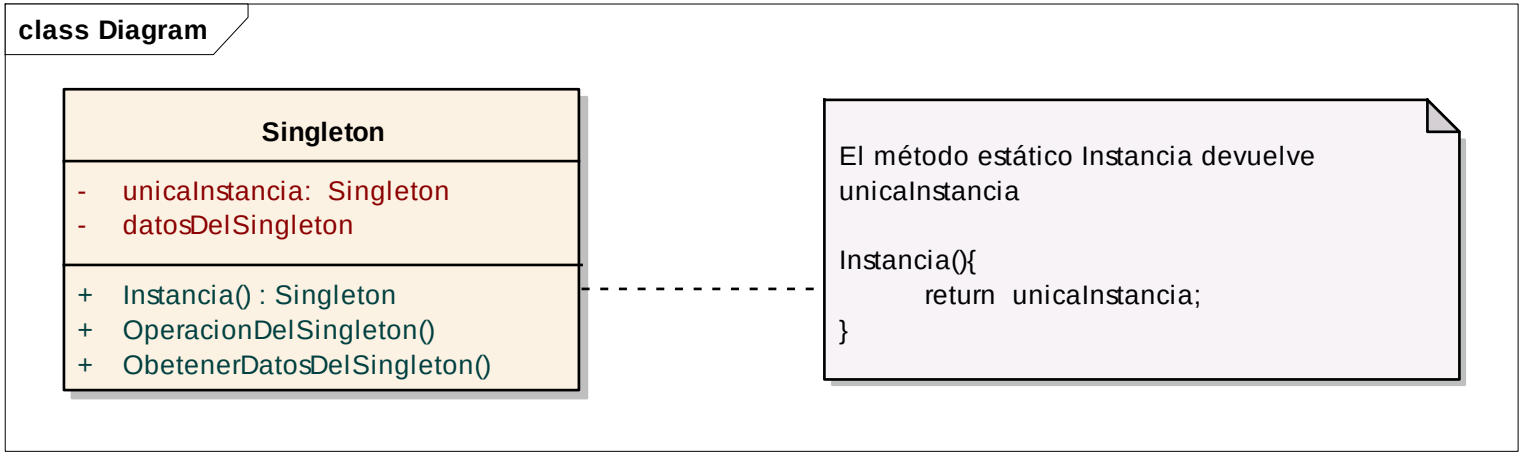
Descripción del patrón

- **Nombre:**
  - Único
- **Propiedades:**
  - Tipo: creación
  - Nivel: objeto
- **Objetivo o Propósito:**
  - Garantiza que una clase solo tenga una instancia, y proporciona un punto de acceso global a ella. Todos los objetos que utilizan una instancia de esa clase usan la misma instancia.

Aplicabilidad

- Use el patrón Singleton cuando:
  - Sólo puede haber una instancia de una clase, y debe ser accesible a los clientes desde un punto de acceso bien conocido.
  - La única instancia debería ser extensible por herencia, y los clientes deberían poder usar una instancia extendida sin modificar su código.

Estructura


class Diagram
Singleton
- unicaInstancia: Singleton
- datosDelSingleton
+ Instancia() : Singleton
+ OperacionDelSingleton()
+ ObetenerDatosDelSingleton()
El método estático Instancia devuelve unicaInstancia
Instancia(){
return unicaInstancia;
}

Estructura


Singleton
-instance : Singleton
-Singleton()
+Instance() : Singleton

Estructura. Participantes

- **Singleton**: Define una operación Instancia que permite a los clientes acceder a su única instancia. Instancia es una operación estática (de clase).

- Puede ser responsable de crear su única instancia.

Estructura. Variaciones

- **Variaciones del patrón:**
- Se puede tener más de una instancia dentro de la clase. Aquellos objetos que conocen la existencia de múltiples instancias pueden utilizar algún método para obtener una de ellas. Ventaja: el resto de la aplicación permanece inalterada.
- El método de acceso al Singleton puede ser el punto de acceso al conjunto total de instancias, aunque todas tengan un tipo distinto. El método de acceso puede determinar, en tiempo de ejecución, qué tipo de instancia devolver.

Estructura. Código

```
public class Singleton
{
        private static Singleton instancia;

        private Singleton() {}

        public static Singleton getInstancia()
        {
                if (instancia == null) {
                  instancia = new Singleton();
                }
                return instancia;
        }
}
```


Estructura. Código

```
public class SingletonOpt
{
        private static SingletonOpt instancia = new SingletonOpt();

        private SingletonOpt() {}

        public static SingletonOpt getInstancia()
        {
                return instancia;
        }
}
```


Consecuencias

- Existe exactamente una instancia de una clase Singleton.
- **Acceso controlado a la única instancia.** Otras clases que quieran una referencia a la única instancia de la clase Singleton conseguirán esa instancia llamando al método estático *getInstancia* de la clase.
- **Espacio de nombres reducido.** Este patrón es una mejora sobre las variables globales. Evita contaminar el espacio de nombres con variables globales que almacenen las instancias.

Consecuencias

- **Permite un número variable de instancias.** El patrón permite la creación de más de una instancia de la clase Singleton, facilitando el control del número de instancias usadas en la aplicación. Solo hay que cambiar la operación de acceso a la instancia del Singleton.
- Tener **subclases** de una clase Singleton es complicado y resultan clases imperfectamente encapsuladas. Una clase derivada de un Singleton no es un Singleton. Para hacer subclases de una clase Singleton, se debería tener un constructor que no sea privado.

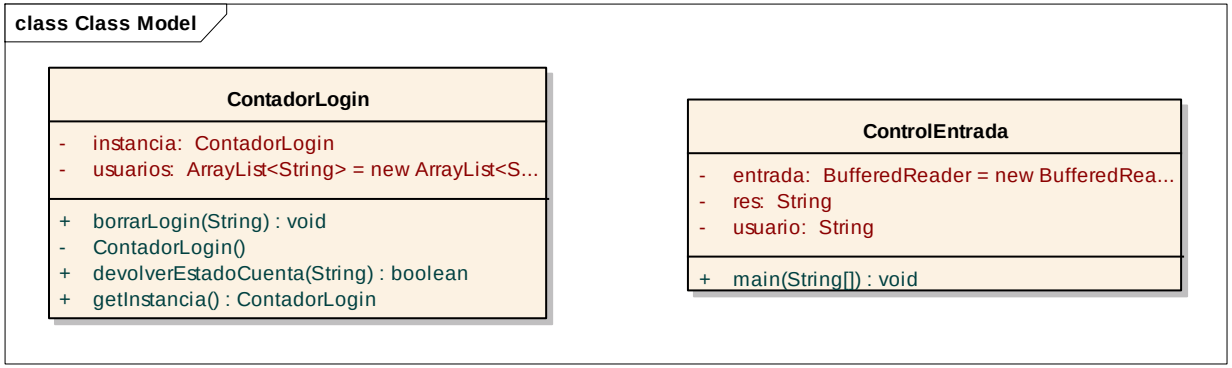
Patrones relacionados

- **Abstract Factory**. Las clases Factorías Concretas (Concrete Factory) son implementadas a menudo como clases Singleton.
- **Factory Method**. Una factoría por aplicación, es un ejemplo perfecto de un Singleton.
- **Builder,** que es usado para construir objetos complejos, mientras que el Singleton es usado para crear un objeto accesible globalmente.
- **Prototipo**, que es usado para copiar un objeto, o crear un objeto a partir de un prototipo, mientras que un Singleton es usado para asegurar que solo se garantiza un prototipo.

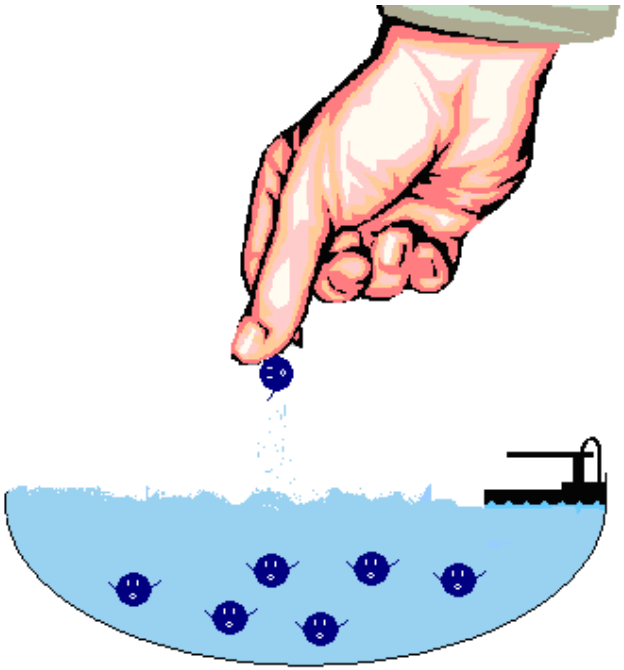
Código de ejemplo

Control de Entrada

Código de ejemplo



Anexo: Object Pool



Descripción del patrón

- El Patrón Object Pool gestiona la reutilización de objetos para una clase cuyas instancias son caras de crear o de la que solo podemos crear un número limitado de objetos.

- Una vez visto el patrón Singleton y sabiendo lo que es una clase Singleton nos será muy sencillo entender este patrón. El patrón Object Pool lleva como base el patrón Singleton.

Aplicabilidad

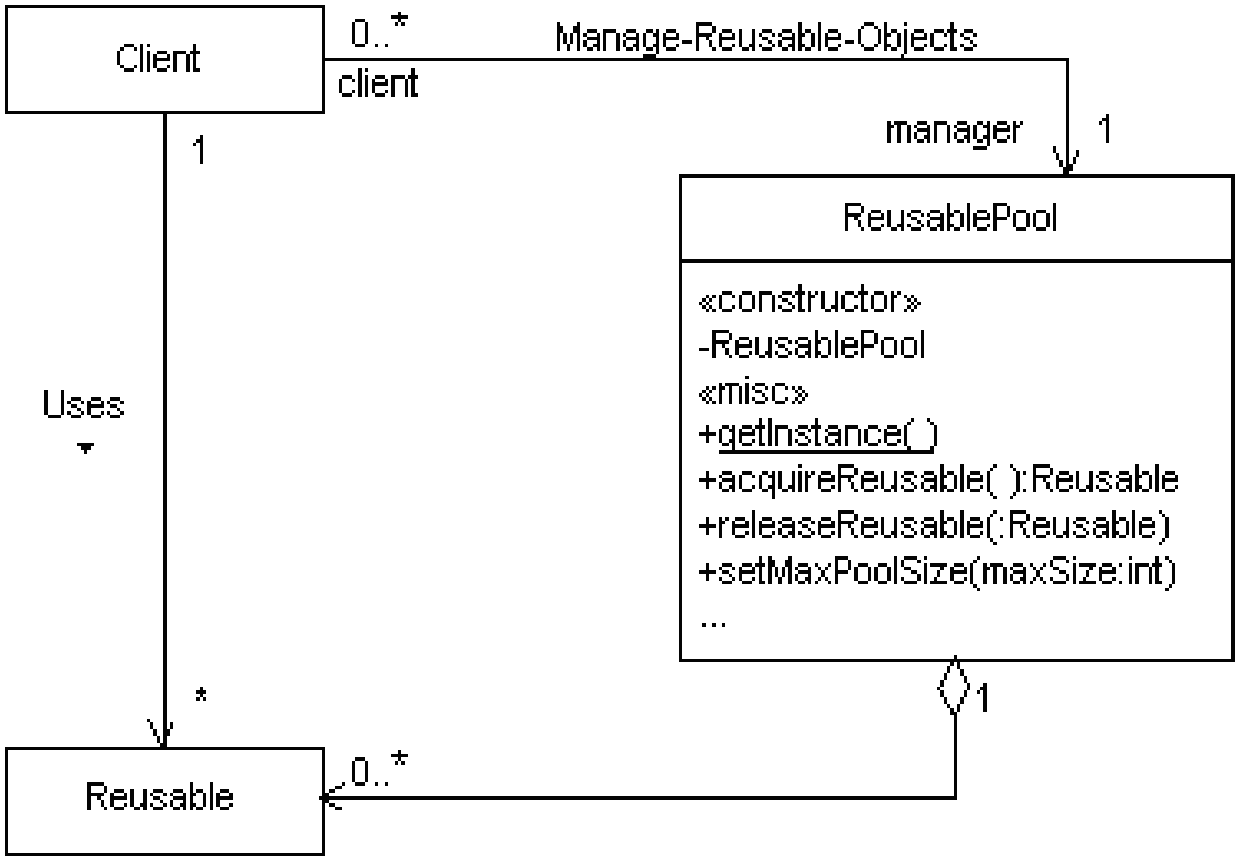
- Tengamos que programar una aplicación para proporcionar acceso a una base de datos dada. Los clientes mandarán peticiones a la base de datos a través de una conexión de red. El servidor de la base de datos recibirá las peticiones a través de la conexión de red y devolverá los resultados a través de la misma conexión.
- Crear conexiones a bases de datos que no son necesarias es malo por las siguientes razones:
  - Crear cada conexión puede llevar unos cuantos segundos.
  - Cuantas más conexiones tengamos establecidas, más tiempo lleva el crear nuevas conexiones.
  - Cada conexión usa una conexión de red. Algunas plataformas limitan el número de conexiones de red que permiten.

Aplicabilidad

- Por lo tanto el patrón evita el gasto de crear objetos de conexión y pone un límite en su número.

- La estrategia que usa el patrón para manejar las conexiones a la base de datos se basará en la premisa de que las conexiones son intercambiables. Tan pronto como necesitemos hacer una petición a una base de datos, no importa cuál de las conexiones de un programa se use.

Estructura


Client
0..*
client
Manage-Reusable-Objects
manager
1
1
ReusablePool
«constructor»
-ReusablePool
«misc»
+getInstance( )
+acquireReusable( ):Reusable
+releaseReusable(:Reusable)
+setMaxPoolSize(maxSize:int)
...
Uses
*
1
Reusable
0..*

Estructura. Participantes

- **Reusable.** Sus instancias colaboran con otros objetos durante un tiempo limitado, después ya no van a ser necesitados para esa colaboración.
- **Client.** Sus instancias utilizan los objetos Reusables.
- **ReusablePool.** Sus instancias manejan los objetos Reusables para ser usados por objetos Client. Se mantienen objetos Reusable que no están siendo utilizados en la misma piscina de objetos de forma que puedan ser utilizados. Está diseñada como una clase Singleton. Su constructor(es) es privado, lo que fuerza a que las otras clases tengan que llamar a su método getInstance para conseguir una instancia de la misma.

Estructura. Participantes

- Un objeto ReusablePool mantiene una colección de objetos Reusable.
- Un Cliente llama al método acquireReusable cuando necesita un objeto Reusable. Situaciones:
  - Si hay algún objeto Reusable en la piscina, lo saca y lo devuelve.
  - Si la piscina está vacía, se crea un objeto Reusable si se puede. Si no puede crear un nuevo objeto Reusable, entonces espera hasta que alguno sea devuelto.
- Un Cliente llama al método releaseReusable cuando ya ha terminado con el objeto, devolviéndolo a la piscina de objetos.

Consecuencias

- Los Clientes no crean objetos. La creación y destrucción de objetos es manejada por una piscina centralizada.
- Los Clientes piden objetos de la piscina y devuelven objetos a la piscina en lugar de destruirlos. Se evita la sobrecarga de crear y destruir objetos frecuentemente.
- Algunas veces, el sistema solo puede dar soporte a un número limitado de objetos. La piscina de objetos se encarga de este límite limitando el número de objetos que hay dentro de la piscina

Código de ejemplo