Singleton Pattern

Introducción y nombre

Singleton. Creacional. La idea del patrón Singleton es proveer un mecanismo para limitar el número de instancias de una clase. Por lo tanto el mismo objeto es siempre compartido por distintas partes del código. Puede ser visto como una solución más elegante para una variable global porque los datos son abstraídos por detrás de la interfaz que publica la clase singleton.

Intención

Garantizar que una clase sólo tenga una instancia y proporcionar un punto de acceso global a ella.

También conocido como

Algunas clases sólo pueden tener una instancia. Una variable global no garantiza que sólo se instancia una vez. Se utiliza cuando tiene que haber exactamente una instancia de una clase y deba ser accesible a los clientes desde un punto de acceso conocido.

Motivación

Algunas clases sólo pueden tener una instancia. Una variable global no garantiza que sólo se instancia una vez. Se utiliza cuando tiene que haber exactamente una instancia de una clase y deba ser accesible a los clientes desde un punto de acceso conocido.

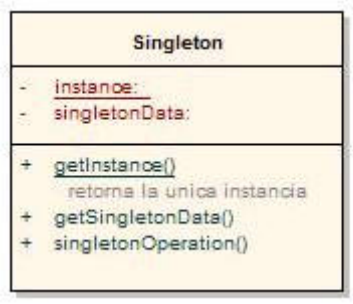
Solución

Usaremos este patrón cuando:

Debe haber exactamente una instancia de una clase y deba ser accesible a los clientes desde un punto de acceso conocido.

Se requiere de un acceso estandarizado y conocido públicamente.

Diagrama UML



Participantes

Singleton: la clase que es Singleton define una instancia para que los clientes puedan accederla. Esta instancia es accedida mediante un método de clase.

Colaboraciones

Clientes: acceden a la única instancia mediante un método llamado getInstance().

Consecuencias

El patrón Singleton tiene muchas ventajas:

Acceso global y controlado a una única instancia: dado que hay una única instancia, es posible mantener un estricto control sobre ella.

Es una mejora a las variables globales: los nombres de las variables globlales no siguen un estándar para su acceso. Esto obliga al desarrollador a conocer detalles de su implementación.

Permite varias instancias de ser necesario: el patrón es lo suficientemente configurable como para configurar más de una instancia y controlar el acceso que los clientes necesitan a dichas instancias.

Implementación

Privatizar el constructor

Definir un método llamado getInstance() como estático

Definir un atributo privado como estático.

Pueden ser necesarias operaciones de terminación (depende de la gestión de memoria del lenguaje)

En ambientes concurrentes es necesario usar mecanismos que garanticen la atomicidad del método getInstance(). En Java esto se puede lograr mediante la sincronización de un método.

Según el autor que se lea el método getInstance() también puede llamarse instance() o Instance()

Código de muestra

[code]
```
public class Singleton {
	private static Singleton instance;

	private Singleton(){
	}

	public synchronized static Singleton getInstance(){
		if (instance == null) {
			instance = new Singleton();
		}
		return instance;
	}
}
```

La forma de invocar a la clase Singleton sería:

```
public static void main(String[] args) {
		Singleton s2 = new Singleton();
		// lanza un error ya que el constructor esta privatizado.

		Singleton s1 = Singleton.getInstance();
		// forma correcta de convocar al Singleton
```
[/code]

Cuándo utilizarlo

Sus usos más comunes son clases que representan objetos unívocos. Por ejemplo, si hay un servidor que necesita ser representado mediante un objeto, este debería ser único, es decir, debería existir una sola instancia y el resto de las clases deberían de comunicarse con el mismo servidor. Un Calendario, por ejemplo, también es

único para todos.

No debe utilizarse cuando una clase esta representando a un objeto que no es único, por ejemplo, la clase Persona no debería ser Singleton, ya que representa a una persona real y cada persona tiene su propio nombre, edad, domicilio, DNI, etc.