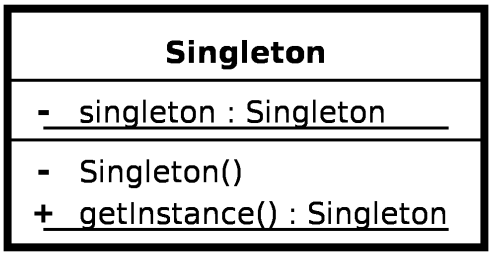
Singleton



El patrón de diseño **singleton** (instancia única) está diseñado para restringir la creación de objetos pertenecientes a una clase o el valor de un tipo a un único objeto.

Su intención consiste en garantizar que una clase sólo tenga una instancia y proporcionar un punto de acceso global a ella.

El patrón *singleton* se implementa creando en nuestra clase un método que crea una instancia del objeto sólo si todavía no existe alguna. Para asegurar que la clase no puede ser instanciada nuevamente se regula el alcance del constructor (con atributos como protegido o privado).

La instrumentación del patrón puede ser delicada en programas con múltiples hilos de ejecución. Si dos hilos de ejecución intentan crear la instancia al mismo tiempo y esta no existe todavía, sólo uno de ellos debe lograr crear el objeto. La solución clásica para este problema es utilizar exclusión mutua en el método de creación de la clase que implementa el patrón.

Las situaciones más habituales de aplicación de este patrón son aquellas en las que dicha clase controla el acceso a un recurso físico único (como puede ser el ratón o un archivo abierto en modo exclusivo) o cuando cierto tipo de datos debe estar disponible para todos los demás objetos de la aplicación.

El patrón *singleton* provee una única instancia global gracias a que:

- La propia clase es responsable de crear la única instancia.
- Permite el acceso global a dicha instancia mediante un método de clase.
- Declara el constructor de clase como privado para que no sea instanciable directamente.

Ejemplo de implementación

Delphi

Ésta implementación ha sido sacada de [1] y está basada en la sobreescritura de los métodos **NewInstance** y **FreeInstance** que se hereda de la clase TObject, la madre de todos los objetos en Delphi.

```
<font size="12">
type
  TSingleton = class
  public
    class function NewInstance: TObject; override;
    procedure FreeInstance; override;
    class function RefCount: Integer;
  end;
var
  Instance  : TSingleton  = nil;
  Ref_Count : Integer     = 0;
</font>
```

Y su implementación sería así:

```
<font size="12">
procedure TSingleton.FreeInstance;
begin
  Dec( Ref_Count );
  if ( Ref_Count = 0 ) then
  begin
    Instance := nil;
    // Destroy private variables here
    inherited FreeInstance;
  end;
end;

class function TSingleton.NewInstance: TObject;
begin
  if ( not Assigned( Instance ) ) then
  begin
    Instance := inherited NewInstance;
    // Initialize private variables here, like this:
    //   TSingleton(Result).Variable := Value;
  end;
  Result := Instance
  Inc( Ref_Count );
end;

class function TSingleton.RefCount: Integer;
begin
  Result := Ref_Count;
end;
</font>
```

Java

Una implementación **correcta** en el lenguaje de programación Java para programas multi-hilo es la solución conocida como "inicialización bajo demanda" sugerida por Bill Pugh:

```
<font size="12">
public class Singleton {
    private static Singleton INSTANCE = new Singleton();

    // El constructor privado no permite que se genere un constructor
por defecto
    // (con mismo modificador de acceso que la definicion de la clase)
    private Singleton() {}

    public static Singleton getInstance() {
        return INSTANCE;
    }
}
</font>
```

Un ejemplo **correcto** de inicialización diferida. Se deja para comentar un error común en Java al no tener en cuenta la sincronización de métodos.

```
<font size="12">
public class Singleton {
    private static Singleton INSTANCE = null;

    // Private constructor suppresses
    private Singleton() {}

    // creador sincronizado para protegerse de posibles problemas
multi-hilo
    // otra prueba para evitar instanciación múltiple
    private synchronized static void createInstance() {
        if (INSTANCE == null) {
            INSTANCE = new Singleton();
        }
    }

    public static Singleton getInstance() {
        if (INSTANCE == null) createInstance();
        return INSTANCE;
    }
}
</font>
```

Para asegurar que se cumpla el requerimiento de "única instancia" del singleton; la clase deberia producir un objeto no clonable:

```
<font size="12">
//Así se podria clonar el objeto y no tendria unicidad.
SingletonObjectDemo clonedObject = (SingletonObjectDemo) obj.clone();
</font>
```

Entonces, se deberia impedir la clonacion sobreescribiendo el metodo "clone" de la siguiente manera:

```
<font size="12">
//El metodo "clone" es sobreescrito por el siguiente que arroja una
excepción:
public Object clone() throws CloneNotSupportedException {
          throw new CloneNotSupportedException();
}
</font>
```

Ref: http://forums.sun.com/thread.jspa?threadID=785736, 28/07/2010.

Otra cuestión a tener en cuenta es que los métodos (o la clase) deberían ser declarados como: ***final*** para que no puedan ser sobreescritos.

C#

Un ejemplo **correcto** de inicialización diferida y segura en entornos multi-hilo en C# sería:

```
<font size="12">
public class Singleton
{
    // Variable estática para la instancia, se necesita utilizar una
función lambda ya que el constructor es privado
    private static readonly Lazy<Singleton> instance = new Lazy<Singleton>(() => new
Singleton());

    // Constructor privado para evitar la instanciación directa
    private Singleton()
    {
    }

    // Propiedad para acceder a la instancia
    public static Singleton Instance
    {
        get
        {
            return instance.Value;
        }
    }
}

// Clase de prueba
public class Prueba
{
   private static void Main(string[] args)
   {
     //Singleton s0 = new Singleton();  //Error
     Singleton s1 = Singleton.Instance;
     Singleton s2 = Singleton.Instance;
     if(s1==s2)
     {
       // Misma instancia
     }
   }
}
</font>
```

C++

Una solución posible en C++ (conocida como el singleton de Meyers) en la cual el singleton es un objeto local estático (notar que esta solución no es segura en programas multi-hilo):

```
<font size="12">
template<typename T> class Singleton
{
    public:
        static T& Instance()
        {
            static T laInstanciaSingleton; //asumir T posee un
constructor por defecto
            return laInstanciaSingleton;
        }
};

class SoloUno : public Singleton<SoloUno>
{
     friend class Singleton<SoloUno>; //para dar acceso al constructor privado
de SoloUno
     //..definir aquí el resto de la interfaz
};
</font>
```

Python

El siguiente es un ejemplo de implementación de Singleton en Python (tampoco es segura en la programación multi-hilo)

```
<font size="12">
class Singleton (object):
    instance = None
    def __new__(cls, *args, **kargs):
        if cls.instance is None:
            cls.instance = object.__new__(cls, *args, **kargs)
        return cls.instance

#Usage
mySingleton1 = Singleton()
mySingleton2 = Singleton()

#mySingleton1 y mySingleton2 son la misma instancia
assert mySingleton1 is mySingleton2
</font>
```

Y otra posibilidad interesante es implementarlo como una metaclase:

```
<font size="12">
class Singleton(type):
```

```
    def __init__(cls, name, bases, dct):
        cls.__instance = None
        type.__init__(cls, name, bases, dct)

    def __call__(cls, *args, **kw):
        if cls.__instance is None:
            cls.__instance = type.__call__(cls, *args,**kw)
        return cls.__instance

class A:
    __metaclass__ = Singleton
    # Definir aquí el resto de la interfaz

a1 = A()
a2 = A()

assert a1 is a2
</font>
```

Visual Basic. NET

Una implementación del patrón singleton en Visual Basic. NET es la siguiente:

```
<font size="12">

Public Class Singleton

    Private Sub New()
    End Sub

    Private Shared _instancia As Singleton

    Public Shared ReadOnly Property Instancia() As Singleton
        Get
            If _instancia Is Nothing Then
                _instancia = New Singleton
            End If

            Return _instancia
        End Get
    End Property

End Class

</font>
```

PHP5

Una implementación del patrón singleton en PHP5 es la siguiente:

```
<font size="12">
<?php
class Ejemplo
{
    // Contenedor Instancia de la Clase
    private static $instance;

    // A private constructor; previene creacion de objetos via new
    private function __construct()
    {
        echo 'Soy el constructor';
    }

    // EL metodo singleton
    public static function singleton()
    {
        if (!isset(self::$instance)) {
            $c = __CLASS__;
            self::$instance = new $c;
        }

        return self::$instance;
    }

    // Clone no permitido
    public function __clone()
    {
        trigger_error('Clone no se permite.', E_USER_ERROR);
    }

}

?>
</font>
```

Action Script 3

Una implementación del patrón singleton en Action Script 3 es la siguiente:

```
<font size="12">
public class Singleton{
   private static var instance:Singleton;
   private static var allowInstance:Boolean;
   public function Singleton(){
      if(!allowInstance){
         throw new Error("Debes usar getInstance()");
```

```
      }else{
         trace("Se inicializó una instancia de Singleton");
      }
   }

   public static function getInstance():Singleton{
      if(instance==null){
         allowInstance=true;
         instance= new Singleton();
         allowInstance=false;
      }else{
         trace("Se regresa la instancia existente");
      }
      return instance;
   }
}
</font>
```

Javascript

Una implementación del patrón singleton en Javascript es la siguiente:

```
<font size="12">
Singleton = function Singleton$constructor() {
    return {
        getInstance : function Singleton$getInstance() {
            return this;
        }
    };
}();
</font>
```

Patrones relacionados

- Abstract Factory: muchas veces son implementados mediante Singleton, ya que normalmente deben ser accesibles públicamente y debe haber una única instancia que controle la creación de objetos.
- Monostate: es similar al singleton, pero en lugar de controlar el instanciado de una clase, asegura que todas las instancias tengan un estado común, haciendo que todos sus miembros sean de clase.f

Enlaces externos

- Patrón Singleton en PHP [2]
- A Q&A page [3] (java.sun.com)
- PEC(TM) [4], compilador que garantiza la correcta instrumentación del patrón singleton y de otros patrones.
- Data&Object Factory [5], explicación e implementación en C# del patrón singleton.
- Patrón Singleton explicado en video [6], artículo de Lucas Ontivero donde explica mediante tres videos de 8 minutos 3 implementaciones del patrón Singleton en CSharp (nivel básico)

Referencias

[1] http://edn.embarcadero.com/article/22576
[2] http://desarrolladorsenior.blogspot.com/2009/09/el-patron-de-diseno-singleton-esta.html
[3] http://developer.java.sun.com/developer/qow/archive/111/
[4] http://pec.dev.java.net/
[5] http://www.dofactory.com/Patterns/PatternSingleton.aspx
[6] http://geeks.ms/blogs/lontivero/archive/2008/07/27/patterns-patr-243-n-singleton-explicado.aspx

Fuentes y contribuyentes del artículo

Singleton *Fuente*: http://es.wikipedia.org/w/index.php?oldid=46116687 *Contribuyentes*: Adryitan, Arleytriana, Ascánder, Bnl, Dem, Denistorres, Dodo, Dusan, Farisori, GermanX, Gothmog, Guishogt, Isha, Jlprwp, Jugones55, Marimillo, Marmas, Nebulha, Niqueco, Oscar ., PayoMalayo, Penyaskito, Pleira, Porao, Taichi, 101 ediciones anónimas

Fuentes de imagen, Licencias y contribuyentes

Archivo:Singleton UML class diagram.svg *Fuente*: http://es.wikipedia.org/w/index.php?title=Archivo:Singleton_UML_class_diagram.svg *Licencia*: Public Domain *Contribuyentes*: User:Trashtoy

Licencia