A Delphi Singleton Class: Update

by Marco Cantù and Hallvard Vassbotn

Following Hallvard Vassbotn's article in the January 1999 issue, *Design Patterns*, Marco Cantù and Hallvard have been discussing the implementation of the Singleton pattern in Delphi between them. The discussion is illuminating, not least because it points up the fact that there will always be differing opinions on the best implementation technique for some aspects of our craft, even when the protagonists are as skilled as Marco and Hallvard!
Chris Frizelle, Editor

Marco Cantù Writes...

In the January issue there is an interesting article by Hallvard Vassbotn discussing the implementation of the Singleton pattern in Delphi. I'm very happy to see discussion of design patterns in Delphi, because they can really help to define common coding practices and teach programmers to design classes before coding them (something not so common as one might think).

The reason for this letter is that I see that in some articles on patterns the implementation tends to follow the C++ perspective a little too much. The Object Pascal language, luckily, offers us a few extra features we can leverage to improve the implementation of this and other patterns.

In this specific case, Delphi's `TObject` class has a nice class function, `NewInstance`, which is automatically called by the system every time a new object is created by any call to a constructor. Overriding this function we can customise the way an object allocates its memory, but we can also implement objects pooling and other advanced techniques. As a simple example, we can create a Singleton class as shown in Listing 1.

Creating one or more instances of this class with the usual code:

```
S1 := TSingle.Create;
```

the system effectively creates only one instance of the object, returning the same single instance for every new object being created.

The advantage of this code, compared to the solution presented in the article, is that this object can be used like any other Delphi object, with no special functions to access to an instance. Of course the second time `NewInstance` is called it might raise an exception to avoid this behaviour and implement a different type of Singleton.

It is important to notice that this changes the nature of the Singleton a little, as it is not seen any more as a global object instantiated when the program starts, but as a class which disallows multiple instances, returning always the same one on request. So I'm actually implementing something different than what was in the original article, which is appropriate only in some circumstances.

The code shown above doesn't solve the problem of the destruction of the single object. It is certainly possible to disable the destructor, eventually raising an exception, and let the program clear the single instance in the `finalization` part of the unit.

As an alternative, we can also override the `FreeInstance` method, so that calling `Free` over the aliases of the object doesn't destroy its instance. Listing 2 shows an extension of this idea which implements a reference counted object. This class diverges from the original design pattern presented in Hallvard's article, but you should easily change the implementation to suit your needs.

With this code behind the scenes, a user of this class can call

```
type
  TSingle = class (TObject)
  public
    class function NewInstance: TObject; override;
  end;
implementation
var Instance: TObject = nil;
class function TSingle.NewInstance: TObject;
begin
  if not Assigned (Instance) then
    Instance := inherited NewInstance;
  Result := Instance;
end;
```

➤ *Above: Listing 1*

➤ *Below: Listing 2*

```
type
  TSingle = class (TObject)
  public
    class function NewInstance: TObject; override;
    procedure FreeInstance; override;
  end;
implementation
var
  Instance: TObject = nil;
  nCount: Integer = 0;
procedure TSingle.FreeInstance;
begin
  Dec (nCount);
  if nCount = 0 then begin
    inherited FreeInstance;
    Instance := nil;
  end;
end;
class function TSingle.NewInstance: TObject;
begin
  if not Assigned (Instance) then
    Instance := inherited NewInstance;
  Result := Instance;
  Inc (nCount);
end;
```

```
unit singlebase;
interface
type
  TSingleton = class (TObject)
  public
    class function NewInstance: TObject; override;
    procedure FreeInstance; override;
    class procedure FreeAll;
  end;
implementation
uses
  classes;
var
  InstanceList: TStringList;
  Destroying: Boolean = False;
class procedure TSingleton.FreeAll;
var
  I: Integer;
begin
  Destroying := True;
  for I := 0 to InstanceList.Count - 1 do
    InstanceList.Objects [I].Free;
end;

procedure TSingleton.FreeInstance;
begin
  if Destroying then
    inherited FreeInstance;
end;
class function TSingleton.NewInstance: TObject;
var
  idx: Integer;
begin
  idx := InstanceList.IndexOf (ClassName);
  if idx >= 0 then
    Result := InstanceList.Objects [idx]
  else begin
    Result := inherited NewInstance;
    InstanceList.AddObject (ClassName, Result);
  end;
end;
initialization
  InstanceList := TStringList.Create;
finalization
  TSingleton.FreeAll;
  InstanceList.Free;
end.
```

➤ *Listing 3*

Create, Free and Destroy as usual, but the effect of the calls will be re-interpreted by the class itself. I know this can be seen as a negative effect, as the user of the class doesn't realise what's happening, but on the positive side the user of the class doesn't have to bother with the specific role of the class and can use it as any other class. If the class instance is shared then this is a problem of the class, not a problem of the class user!

Again, I don't think this is always a better solution, but I think it is important to realise that there are many different alternatives to implement the idea of a class with a single instance and the idea of a Singleton. It will be up to you to determine which implementation is the best in each context.

As for the generic solution used for building a base class, I have some alternative coding here as well. The code used by Hallvard to register the derived classes seems quite long and poses a burden on the users of the class. I've had little time to write down these notes, but using the solution indicated above and replacing the single global instance with a list of instances, stored in a TStringList and traced by class name, seems to do the trick. The code for the single base class for a Singleton is in Listing 3. Notice that you can inherit from this class without having to do an extra step: each class inherited by TSingleton will simply create one single shared instance in a completely transparent way.

For the destruction of the objects I've decided to perform it automatically when the program terminates. The operation is done by the FreeAll method, which sets the Destroying flag to True so that the FreeInstance method will actually destroy the objects.

I cannot guarantee this code is perfect, but it works in the simple test program I've built and you can find it on this month's disk.

Again, I was really happy to see Hallvard's article published and I'm just offering a few more hints to let programmers broaden the tools they have to implement a common problem.

Marco Cantù
www.marcocantu.com

Hallvard Vassbotn Replies...

To sum up my position, I agree that the Singleton pattern can be implemented in different and better ways. The solution of using NewInstance and FreeInstance is simple, but it has some major drawbacks.

Firstly, it doesn't prevent the constructor from being called or executed. It just prevents additional instances (ie memory) from being allocated. This means that there is no way for the Singleton author to know when to initialize or free resources (unless he or she is using some kind of reference counting).

Secondly, it has an illogical way of obtaining a reference. For instance, would it not look strange if I was forced to call the constructor of TScreen to get a reference to the Screen Singleton?

Thirdly, after calling TSingleton.Create to get the reference, the user might be confused about the need to call Free or not. Also, if he or she is calling Free, the code becomes more complex:

```
MyFormCount :=
  Screen.FormCount;
```

versus:

```
with TScreen.Create do
  try
    MyFormCount := FormCount;
  finally
    Free;
  end;
```

Because of this, users might skip calling Free altogether, so any reference counting scheme will easily break.

Even with a reference counting scheme and users calling Create and Free correctly, the Singleton instance will not be persistent in memory. It will be freed every time the outer scope frees it.

The virtual NewInstance and FreeInstance (in addition to the Delphi 4-specific AfterConstruction and BeforeDestruction) can be very useful at times, but they are not suited to implementing the Singleton pattern, in my opinion.

Hallvard Vassbotn
hallvard@balder.no

And Finally...

Marco agreed that Hallvard had a point in his response, but believes his alternative solution has merit. So, I guess it's down to you, dear reader, to make your choice!