Design Patterns

Implementing The Singleton Pattern

by Hallvard Vassbotn

In their book *Design Patterns*[1], Gamma *et al* (the 'gang of four') lay the foundation for a new way of approaching software design. The book provides us with recipes for solving typical problems encountered in software construction. This lets us easily re-use knowledge, even in the design stages of development. The book also formalises documentation and understanding of a design pattern.

What Is A Design Pattern?

Let us first establish what we mean by a design pattern. The idea originally came from Christopher Alexander, who proposed using a pattern language to architect buildings and cities. He said: 'Each pattern describes a problem which occurs over and over again... and then describes the core of the solution to that problem, in such a way that you can use the solution a million times over, without ever doing it the same way twice'[2].

The same thing applies to design patterns as used in software construction. Generally, a design pattern has a specific name that makes it easy to reference when discussing possible solutions to a problem. For instance, later in this article we will look at a design pattern with the name Singleton. The pattern includes a description of the problem it solves, to guide us when it is appropriate to use it. The main part is a description of the solution with the different elements, how they relate to each other, and so on. It is important that this description is general, so that it can easily be applied to different situations and implementation languages. Finally, there will usually be a set of consequences and trade-offs of using the pattern.

Implementation Language

As Gamma *et al* admit in their introductory chapter, the choice of implementation language will significantly influence the design problems you will meet and decisions you have to make. If a specific feature is missing from a language, you may have to implement it yourself. Unfortunately for us, Gamma *et al* use only Smalltalk and C++ as their example languages.

In this article we will first look at the language elements that are unique to Object Pascal when compared to C++ and how this makes many of the problems the design patterns try to solve, non-existent, or at least much easier to solve. Then we will look at one example of a very simple design pattern, the Singleton pattern (pp 127-134 in the book), and how this can best be implemented in Delphi.

Object Pascal As A Better Language

Many people (not to mention the popular computer press) tend to believe that Object Pascal (OP from now on) has only a subset of the language features of C++. While it is true that OP lacks features like multiple inheritance, operator overloading (except for the array subscript operator `[..]`) and templates, it still has an array of language features not found in C++. Over the years, OP has borrowed many features from C++, for better or for worse. I sincerely hope they will not copy all of them: C++ is such a complex language with many pitfalls for both the novice and experienced programmer.

If we look at the language elements which are unique to OP, we find an amazing array of useful features: units, sets, sub-ranges, native DLL support, class types, class reference variables, virtual class methods, virtual constructors, extensive runtime type information (RTTI), message methods, dynamic methods, the `override` keyword, properties, the `try..finally` clause, initialization and finalization sections, variants, `threadvars`, native COM support, interfaces, packages, dynamic arrays, interface delegation, the automatic reference counting mechanism and method pointers. I could go on. This is not intended as an exercise in C++ bashing, but rather an attempt to heighten our awareness of the goodies of OP that we are enjoying daily.

Simplifying Design Patterns

These goodies also make it easier to solve many of the design problems a typical programmer will face again and again. The purpose of having a set of design patterns to follow is that old re-use slogan. Re-use existing knowledge to reduce the time spent and to avoid the many possible traps you can step into.

Of the many OP features, the one that probably stands out from the rest when it comes to structuring code is the method pointer. Method pointers allow us, in a way, to override methods at runtime and even change the override dynamically. The override is not limited to methods of a specific class, the only requirement is that the method follows a specific parameter signature. This single feature alone will eliminate the need for, or greatly simplify, many of the design patterns found in Gamma *et al*, such as Chain of Responsibility, Command, Mediator, Observer, Template Method and Visitor. The existence of method pointers will often let you avoid having to declare a new descandent class just to add new behaviour. As we know from the world of VCL components, we usually only have to write an event handler. After all, an event is nothing more than a method

1. *Design Patterns, Elements of Reusable Object-Oriented Software*, by Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides. ISBN 0-201-63361-2, Addison-Wesley Publishing Company, 1996. This book is highly recommended reading for anyone doing object oriented programming in any language.

2. *A Pattern Language*, by Christopher Alexander and others, Oxford University Press, 1977.

pointer exposed as a property of an object.

The other vital language feature when it comes to implementing design patterns is the combination of class reference variables and virtual constructors. By using these features, it is possible to create classes whose type is not known until runtime. Many of the creational patterns in Gamma *et al* exist only because C++ lacks this nice feature. So there will often be no need to use these patterns at all in OP. Examples of this are Abstract Factory, Factory Method and Prototype.

The Singleton Pattern

As an example of a design pattern and how to implement it in Delphi, we will look at the Singleton pattern. This is one of the simplest in Gamma *et al* and it is used when we want to ensure that there will be only one instance of a particular class in the application. This can be very useful at times.

If we look at the VCL, many classes are Singletons by intent (for instance `TApplication`, `TScreen` and `TPrinter`), but this is not enforced in the design of the classes.

For instance, there is nothing stopping me from creating another instance of the `Tscreen` class:

```
MyScreen :=
  TScreen.Create(Application);
```

or, even worse, from freeing the instance or clearing the instance pointer, like this:

```
Screen.Free;
Screen := nil;
```

We will see how we can apply the Singleton design pattern to create classes that cannot be easily broken like this.

Even if the goal of designing a Singleton class might seem very simple, there are a number of points we must consider:

- When and by whom should the single instance be created?
- When and by whom should the single instance be destroyed?
- How should external clients get access to the single instance?
- How can we avoid that the value of the instance reference is corrupted?
- How can we avoid illegal creation of additional instances?
- How can we avoid illegal destruction of the single instance?
- How can we keep the design of the Singleton class, but still allow extension by inheritance?

For instance, the situation you are facing might require that you keep some kind of reference count on the Singleton instance, so that it is created the first time it is needed and destroyed the last time it is released.

Keeper Of The Time

For our example, we will implement a very simple Singleton class, `TTimeKeeper`, that will return the current date and time. The class defines three read-only properties that return the values of the `SysUtils` functions `Time`, `Date` and `Now`.

We will assume a model where the instance is created the first time it is needed, but destroyed only when the application closes, in the `finalization` section of the unit.

We could give clients access to the Singleton instance by providing them with a global instance variable in the interface section, but as we have seen with the `Screen` variable, this is not ideal as we have no protection of clients overwriting the instance variable.

The approach I have found to be both robust and readable is to use a simple function that returns the required instance. Then we can move the actual instance variable into the `implementation` section to protect it from external modifications. This is analogous to moving a field from the public to the private part of a class.

To create an even more object oriented interface, we keep the actual implementation in a class function called `Instance`, and let the global function be a friendly wrapper: see Listing 1.

Lifetime Protection

My first attempt at prohibiting illegal creation and destruction of our Singleton instance was too naive. I simply moved the `Create` constructor and `Destroy` destructor into the `private` part of the class declaration. My thinking was that they would then not be available for use outside the unit.

Unfortunately, this does not work. There is no way to reduce the visibility of already existing methods. In fact, in Delphi 4 they added a new hint to the compiler. As the help file states:

> *Overriding virtual method <class>.<method> has a lower visibility than the base class.*

Even if you try this, the help file goes on to say:

> *...the method will maintain the original (higher) visibility. In practice this is never harmful, but it can be confusing to someone reading the source code.*

My next idea was better. We add new class methods that obscure the visibility of the `TObject` methods of the same name:

➤ *Listing 1*

```
var
  TimeKeeperInstance: TTimeKeeper = nil;
class function TTimeKeeper.Instance: TTimeKeeper;
begin
  if not Assigned(TimeKeeperInstance) then
    TimeKeeperInstance := TTimeKeeper.Create;
  Result := TimeKeeperInstance;
end;
class procedure TTimeKeeper.Shutdown;
begin
  if Assigned(TimeKeeperInstance) then begin
    TimeKeeperInstance.Destroy;
    TimeKeeperInstance := nil;
  end;
end;
function TimeKeeper: TTimeKeeper;
begin
  Result := TTimeKeeper.Instance;
end;
finalization
  TTimeKeeper.Shutdown;
```

```
public
  class procedure Create;
  class procedure Free(
    Dummy: integer);
  class procedure Destroy(
    Dummy: integer);
```

To ensure that we get a compile time error if anyone tries to call the `Free` method or `Destroy` destructor, we add a dummy parameter to the corresponding class methods. In addition, we obscure the `Create` constructor with a class procedure, so that people trying to access the function result, would also get a compile error. Note that the declaration of the `Destroy` class procedure will now generate a compile-time warning: we will fix this later.

The alert reader will object (*sic*) and say that the `Destroy` destructor can still be used by casting our `Singleton` instance into a `TObject`, like this:

```
TObject(Singleton).Destroy;
```

This will still destroy our precious `Singleton` instance. To get around this problem, we must override the `Destroy` destructor and disable it completely by raising an exception if it is ever called, that should teach them! However, we still need a constructor and destructor for private use, so we declare new ones in the protected section, `SingletonCreate` and `SingletonDestroy`.

If we try to override the `Destroy` destructor in the same class as we declare the `Destroy` class procedure, we will generate an *Identifier redeclared* error. To get around this, we inherit from another class where the `Destroy` destructor has already been overridden and disabled (Listing 2).

After doing all this, we have the situation shown in Listing 3 if we try to call `Create`, `Destroy` or `Free` directly or indirectly (this is example code from another unit).

This is exactly the situation we want for a `Singleton` class. We have access to the single instance, but we cannot overwrite it or free it. Neither can we create new instances of the class (and thus break the Singleton design).

```
type
  TInvalidateDestroy = class(TObject)
...
  TTimeKeeper = class(TInvalidateDestroy)
...
class procedure TInvalidateDestroy.SingletonError;
// Raise an exception in case of illegal use
begin
  raise ESingleton.CreateFmt('Illegal use of %s singleton instance!',
    [ClassName]);
end;
destructor TInvalidateDestroy.Destroy;
// Protected against use of default destructor
begin
  SingletonError;
end;
```

➤ *Above: Listing 2*

➤ *Below: Listing 3*

```
var
  MyTimeKeeper: TTimeKeeper;
begin
  TimeKeeper := nil;                  // Will not compile
  MyTimeKeeper := TTimeKeeper.Create; // Will not compile
  TimeKeeper.Free;                    // Will not compile
  TimeKeeper.Destroy;                 // Will not compile
  TObject(TimeKeeper).Destroy;     // Compiles, but raises exception at runtime
  TObject(TimeKeeper).Free;        // Compiles, but raises exception at runtime
```

```
var
  TimeKeeperClass: TTimeKeeperClass = TTimeKeeper;
class procedure TTimeKeeper.SetTimeKeeperClass(
  aTimeKeeperClass: TTimeKeeperClass);
// Allow change of instance class
begin
  Assert(Assigned(aTimeKeeperClass));
  if Assigned(TimeKeeperInstance) then
    SingletonError;
  TimeKeeperClass := aTimeKeeperClass;
end;
```

➤ *Above: Listing 4*

➤ *Below: Listing 5*

```
class function TTimeKeeper.Instance: TTimeKeeper;
// Single Instance function - create when first needed
begin
  Assert(Assigned(TimeKeeperClass));
  if not Assigned(TimeKeeperInstance) then
    TimeKeeperInstance := TimeKeeperClass.SingletonCreate;
  Result := TimeKeeperInstance;
end;
```

Inheriting Singleton Classes

There might be cases where you want to ensure that you only have one instance of a class, but you still need the flexibility of declaring a descendant class and creating an instance of that class instead. To support this, we provide a class procedure to set a class reference variable of the class we want to create. This would typically be called from within the `initialization` section of the unit containing the descendant class (Listing 4).

As we can see from the code, we only allow a change of the class type if the `Singleton` instance has not yet been created. Since we have no proper class fields we keep the class reference in another global variable in the `implementation` section of the unit. We must also update the `Instance` class function to use this class reference instead of the hard-coded `TTimeKeeper` value (Listing 5).

The logic of the code stays the same: we have just added an `Assert` statement to make sure the class reference has a valid value and then use it to create the `Singleton` instance. Note that the `SingletonCreate` constructor must be declared `virtual` for this to work properly.

The Complete Picture

Now we have found an implementation that follows the initial design goals well. The complete `TTimeKeeper Singleton` class can be found in Listing 6.

Generalising The Solution

We have now found one satisfactory implementation of the

Singleton design pattern. However, the OOP purist in me is bothered by all that code. What if I need to implement a number of different Singleton classes? Do I really have to write all that code for each one?

The answer is, of course, no. If there are potentially many uses for the general `Singleton` class, we should make it re-usable. Let us try to factor out the Singleton specific parts of `TTimeKeeper` and move them to a separate class, say `TSingleton`. The goal is to keep as little code as possible in the `HVTimeKeeper` unit, while still keeping all the benefits from the Singleton design.

Implementing Class Fields

The only major obstacle I faced when doing this was the recurring fact that there are no class fields in Object Pascal. The problem is that we have simulated class fields by using simple global variables in the `implementation` section. This works fine for a single class, but the model breaks down once you introduce inheritance into the picture. All the classes will share the single global variable with the base class, potentially stepping on each other's feet.

The concept of class fields is such an unusual one for most Pascal programmers that many find it hard to grasp how they should work if they were indeed a part of the language. In OP we do have hard-coded, read-only, class fields, such as `ClassName`, `InstanceSize`, RTTI pointers and so on. These are accessed through class functions, but the actual information is stored as part of the extended VMT (virtual method table). This is how class fields should be implemented as well. A class field follows the VMT, so each new derived class has a separate copy of the class field slot.

➤ *Listing 6*

```
unit HVTimeKeeper;
interface
uses SysUtils;
type
  ESingleton = class(Exception);
  TInvalidateDestroy = class(TObject)
  protected
    class procedure SingletonError;
  public
    destructor Destroy; override;
  end;
  TTimeKeeper = class;
  TTimeKeeperClass = class of TTimeKeeper;
  TTimeKeeper = class(TInvalidateDestroy)
  private
    class procedure Shutdown;
    function GetTime: TDateTime;
    function GetDate: TDateTime;
    function GetNow: TDateTime;
  protected
    // Allow descendants to set new class for the instance
    class procedure SetTimeKeeperClass(aTimeKeeperClass:
      TTimeKeeperClass);
    // Actual constructor and destructor that will be used
    constructor SingletonCreate; virtual;
    destructor SingletonDestroy; virtual;
  public
    // Not for use - for obstruction only
    class procedure Create;
    class procedure Free(Dummy: integer);
{$IFNDEF VER120} {$WARNINGS OFF} {$ENDIF}
    // This generates a warning in Delphi 3. Delphi 4 has
    // the reintroduce keyword to solve this
    class procedure Destroy(Dummy: integer);
     {$IFDEF VER120} reintroduce; {$ENDIF}
    // Simple interface:
    class function Instance: TTimeKeeper;
    property Time: TDateTime read GetTime;
    property Date: TDateTime read GetDate;
    property Now: TDateTime read GetNow;
  end;
{$IFNDEF VER120} {$WARNINGS ON} {$ENDIF}
function TimeKeeper: TTimeKeeper;
implementation
class procedure TInvalidateDestroy.SingletonError;
// Raise an exception in case of illegal use
begin
  raise ESingleton.CreateFmt(
    'Illegal use of %s singleton instance!', [ClassName]);
end;
destructor TInvalidateDestroy.Destroy;
// Protected against use of default destructor
begin
  SingletonError;
end;
var
  TimeKeeperInstance: TTimeKeeper = nil;
  TimeKeeperClass: TTimeKeeperClass = TTimeKeeper;
class procedure TTimeKeeper.SetTimeKeeperClass(
  aTimeKeeperClass:TTimeKeeperClass);
// Allow change of instance class
begin
  Assert(Assigned(aTimeKeeperClass));
  if Assigned(TimeKeeperInstance) then
    SingletonError;
  TimeKeeperClass := aTimeKeeperClass;
end;
class function TTimeKeeper.Instance: TTimeKeeper;
// Single Instance function - create when first needed
begin
  Assert(Assigned(TimeKeeperClass));
  if not Assigned(TimeKeeperInstance) then
    TimeKeeperInstance := TimeKeeperClass.SingletonCreate;
  Result := TimeKeeperInstance;
end;
class procedure TTimeKeeper.Shutdown;
// Time to close down the show
begin
  if Assigned(TimeKeeperInstance) then begin
    TimeKeeperInstance.SingletonDestroy;
    TimeKeeperInstance := nil;
  end;
end;
constructor TTimeKeeper.SingletonCreate;
// Protected constructor
begin
  inherited Create;
end;
destructor TTimeKeeper.SingletonDestroy;
// Protected destructor
begin
  // We cannot call inherited Destroy; here!
  // It would raise an ESingleton exception
end;
// Protected against use of default constructor
class procedure TTimeKeeper.Create;
begin
  SingletonError;
end;
// Protected against use of Free
class procedure TTimeKeeper.Free(Dummy: integer);
begin
  SingletonError;
end;
// Protected against use of default destructor
class procedure TTimeKeeper.Destroy(Dummy: integer);
begin
  SingletonError;
end;
// Property access methods
function TTimeKeeper.GetDate: TDateTime;
begin
  Result := SysUtils.Date;
end;
function TTimeKeeper.GetNow: TDateTime;
begin
  Result := SysUtils.Now;
end;
function TTimeKeeper.GetTime: TDateTime;
begin
  Result := SysUtils.Time;
end;
// Simplified functional interface
function TimeKeeper: TTimeKeeper;
begin
  Result := TTimeKeeper.Instance;
end;
initialization
finalization
// Destroy when application closes
  TTimeKeeper.Shutdown;
end.
```

As we have no user-defined class fields and because of the obvious shortcomings of the simple global variables we used in our first attempt, we must find a better solution. One possible approach is to add a virtual class function that will return the address of the global variable in the descendant class. This will work, but it is fairly involved, as the descendant must always remember to override the class function. Also, there must be one class function for each separate variable.

The solution which I eventually decided on is to use a registration scheme. The base class keeps a global `TList` that keeps all the global variables for all descendants. As new descendants are declared, they must register themselves (typically in the `initialization` section of the unit). As each class registers itself, a new slot in the `TList` is allocated and a handle is returned to the client class. The handle is actually the index of the used slot in the `TList`, but this is invisible to the client.

In our case we need two `TLists`: one to keep the instance pointers of the Singleton objects (`SingletonInstances`) and one to contain the corresponding class references to decide what class will be instantiated at runtime (`SingletonClasses`). The two lists grow in parallel and are always in synchronisation with each other. If we needed more than two fields per registration then I would recommend keeping them in a separate object and having a pointer to the object in a single list instead.

In addition to the registration mechanism, we also allow descendants to override the class that will be used for a specific instance. This is done by sending in the original base class reference and the new class reference that should replace it. The `SingletonClasses` list is searched until the base class is found. Assuming that the corresponding instance in the `SingletonInstances` list has not already been created, the class reference is then overwritten with the new value. This ensures that the new class will be used when the `Singleton` instance is first needed.

The final implementation of the general `TSingleton` class can be seen in Listing 7.

Generalising the implementation of the Singleton pattern has made it more complex, but it has the benefit of simplifying writing new Singleton classes. Simply inherit from the `TSingleton` class and call the `RegisterSingletonClass` function. Our `TTimeKeeper` class can now be written in far fewer lines: see Listing 8.

There are only three major changes here, besides removing all of that support code. In the `initialization` section of the unit, we call the `RegisterSingletonClass` function to register ourselves as a `Singleton` class. This function returns a handle that we store in the global variable `TimeKeeperHandle`. In the `Instance` class function we call `InstanceOf` with our newly acquired handle as a parameter. This will take care of creating the correct class and returning the correct instance to us. It's all pretty simple, isn't it?

➤ *Facing page: Listing 7*

Extending The Singleton

We have talked about the need to extend a given Singleton class by inheriting from it, but we have not shown how to do this in practice. Let us make a simple extension to the time keeper class.

The new class, `TExtTimeKeeper`, will add a couple of new properties and register itself so that it will be used instead of `TTimeKeeper`. The new properties will firstly return a string representation of the current date and time and secondly return a string representing the day of the week: see Listing 9 for the code.

Because the public interface of the class has been extended, we must also create a new function to return the `Singleton` instance to clients. This is to avoid the need for typecasts when accessing the new properties. Other than that, the only requirement is that we must call the `Override SingletonClass` procedure. We do this in the `initialization` section and specify that we want to replace the `TTimeKeeper` Singleton with our own version. This assumes that `TTimeKeeper` has already been registered. In this case it will, because we are using the `HVTimeKeeper2` unit and

➤ *Listing 8*

```
unit HVTimeKeeper2;
interface
uses HVSingleton;
type
  TTimeKeeper = class(TSingleton)
  private
    function GetTime: TDateTime;
    function GetDate: TDateTime;
    function GetNow: TDateTime;
  public
    class function Instance: TTimeKeeper;
    property Time: TDateTime read GetTime;
    property Date: TDateTime read GetDate;
    property Now: TDateTime read GetNow;
  end;
function TimeKeeper: TTimeKeeper;
implementation
uses SysUtils;
var TimeKeeperHandle: TSingletonHandle;
class function TTimeKeeper.Instance: TTimeKeeper;
// Single Instance function - create when first needed
begin
  Result := TTimeKeeper(InstanceOf(TimeKeeperHandle));
end;
// Property access methods
function TTimeKeeper.GetDate: TDateTime;
begin
  Result := SysUtils.Date;
end;
function TTimeKeeper.GetNow: TDateTime;
begin
  Result := SysUtils.Now;
end;
function TTimeKeeper.GetTime: TDateTime;
begin
  Result := SysUtils.Time;
end;
// Simplified functional interface
function TimeKeeper: TTimeKeeper;
begin
  Result := TTimeKeeper.Instance;
end;
initialization
  TimeKeeperHandle :=
    TTimeKeeper.RegisterSingletonClass(TTimeKeeper);
end.
```

```
unit HVSingleton;
interface
uses
  SysUtils;
type
  ESingleton = class(Exception);
  TInvalidateDestroy = class(TObject)
  protected
    class procedure SingletonError;
  public
    destructor Destroy; override;
  end;
  TSingletonOpaqueInfo = record end;
  TSingletonHandle = ^TSingletonOpaqueInfo;
  TSingleton = class;
  TSingletonClass = class of TSingleton;
  TSingleton = class(TInvalidateDestroy)
  private
    class procedure Startup;
    class procedure Shutdown;
  protected
    // Allow descendants to register themselves
    class function RegisterSingletonClass(aSingletonClass:
      TSingletonClass): TSingletonHandle;
    // Allow descendants to set new class for the instance:
    class procedure OverrideSingletonClass(
      BaseSingletonClass, NewSingletonClass:
      TSingletonClass);
    // Interface for descendants to get instance pointer
    class function InstanceOf(Handle: TSingletonHandle):
      TSingleton;
    // Actual constructor and destructor that will be used:
    constructor SingletonCreate; virtual;
    destructor SingletonDestroy; virtual;
  public
    // Not for use - for obstruction only:
    class procedure Create;
    class procedure Free(Dummy: integer);
{$IFNDEF VER120} {$WARNINGS OFF} {$ENDIF}
    // This generates a warning in D3. D4 has the
    // reintroduce keyword to solve this
    class procedure Destroy(Dummy: integer);
      {$IFDEF VER120} reintroduce; {$ENDIF}
  end;
{$IFNDEF VER120} {$WARNINGS ON} {$ENDIF}
implementation
uses
  Classes;
class procedure TInvalidateDestroy.SingletonError;
// Raise an exception in case of illegal use
begin
  raise ESingleton.CreateFmt(
    'Illegal use of %s singleton instance!', [ClassName]);
end;
destructor TInvalidateDestroy.Destroy;
// Protected against use of default destructor
begin
  SingletonError;
end;
var
  SingletonInstances : TList; { of TSingletons       }
  SingletonClasses   : TList; { of TSingletonClasses }
class procedure TSingleton.Startup;
begin
  SingletonInstances := TList.Create;
  SingletonClasses   := TList.Create;
end;
class procedure TSingleton.Shutdown;
var
  SingletonInstance: TSingleton;
  i : integer;
begin
  // Free any singleton instances
  for i := SingletonInstances.Count-1 downto 0 do begin
    SingletonInstance :=
      TSingleton(SingletonInstances.List^[i]);
    if Assigned(SingletonInstance) then
      SingletonInstance.SingletonDestroy;
  end;
  // Free the lists
  SingletonInstances.Free;
  SingletonInstances := nil;
  SingletonClasses.Free;
  SingletonClasses := nil;
end;
class function TSingleton.RegisterSingletonClass(
  aSingletonClass: TSingletonClass): TSingletonHandle;
// Register a new Singleton class and allocate space for the
// instance pointer
var Index: integer;
begin
  Assert(Assigned(aSingletonClass));
  Assert(
    SingletonClasses.IndexOf(Pointer(aSingletonClass)) < 0);
  SingletonClasses.Add(Pointer(aSingletonClass));
  // Return the index of the instance pointer as a handle
  Index := SingletonInstances.Add(nil);
  Result := TSingletonHandle(Index);
  Assert(SingletonClasses.Count = SingletonInstances.Count);
end;
class procedure TSingleton.OverrideSingletonClass(
  BaseSingletonClass, NewSingletonClass: TSingletonClass);
// Allow change of instance class
var
  ThisClass: TSingletonClass;
  i : integer;
begin
  Assert(Assigned(BaseSingletonClass));
  Assert(Assigned(NewSingletonClass));
  Assert(BaseSingletonClass <> TSingleton);
  Assert(
    NewSingletonClass.InheritsFrom(BaseSingletonClass));
  for i := 0 to SingletonClasses.Count-1 do begin
    ThisClass := TSingletonClass(SingletonClasses.List^[i]);
    if ThisClass.InheritsFrom(BaseSingletonClass) and
      (SingletonInstances.List^[i] = nil) then begin
      SingletonClasses.List^[i] :=
        Pointer(NewSingletonClass);
      Exit;
    end;
  end;
  // If we get here, the base class was not found or
  // an instance had already been created
  SingletonError;
end;
class function TSingleton.InstanceOf(
  Handle: TSingletonHandle): TSingleton;
// Single Instance function - create when first needed
var Index: Integer;
begin
  // Convert the handle back to an index
  Index := Integer(Handle);
  Assert((Index >= 0) and
    (Index <= SingletonInstances.Count-1));
  Assert(Assigned(SingletonClasses.List^[Index]));
  if not Assigned(SingletonInstances.List^[Index]) then
    SingletonInstances.List^[Index] :=
      TSingletonClass(
        SingletonClasses.List^[Index]).SingletonCreate;
  Result := SingletonInstances.List^[Index];
end;
constructor TSingleton.SingletonCreate;
// Protected constructor
begin
  inherited Create;
end;
destructor TSingleton.SingletonDestroy;
// Protected destructor
begin
  // We cannot call inherited Destroy; here!
  // It would raise an ESingleton exception
end;
// Protected against use of default constructor
class procedure TSingleton.Create;
begin
  SingletonError;
end;
// Protected against use of Free
class procedure TSingleton.Free(Dummy: integer);
begin
  SingletonError;
end;
// Protected against use of default destructor
class procedure TSingleton.Destroy(Dummy: integer);
begin
  SingletonError;
end;
initialization
  TSingleton.Startup;
finalization
  TSingleton.Shutdown;
end.
```

`TTimeKeeper` is registered in that unit's `initialization` section.

This process could theoretically be repeated as we wish, by adding descendants to `TExtTimeKeeper` and so on. Note that if more than one descendant of `TTimeKeeper` calls `OverrideSingletonClass`, the last one will actually be used. There is no check to guard against this situation, but it would be very easy to add one.

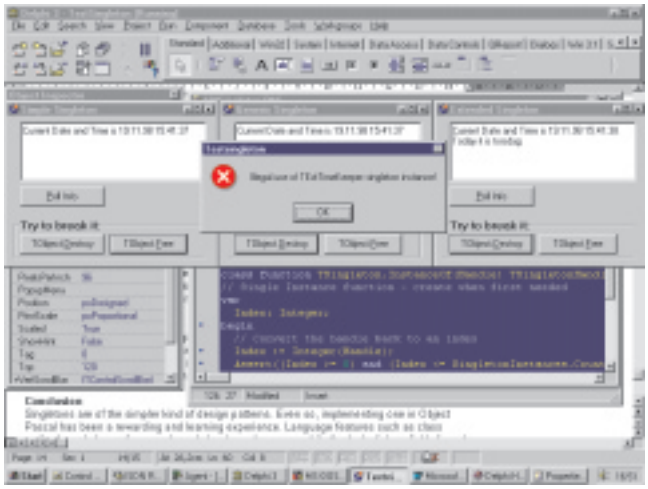
Demonstration Project

On the disk you will find a simple demo project that utilises the three different time keeper classes we have presented: see the screenshot on the next page.

```
unit HVExtTimeKeeper;
interface
uses HVTimeKeeper2;
type
  TExtTimeKeeper = class(TTimeKeeper)
  private
    function GetNowStr: string;
    function GetTodayName: string;
  public
    property NowStr: string read GetNowStr;
    property TodayName: string read GetTodayName;
  end;
// New access function is only needed if the public
// interface has been extended
function TimeKeeper: TExtTimeKeeper;
implementation
uses SysUtils;
function TExtTimeKeeper.GetNowStr: string;
begin
  Result := SysUtils.DateTimeToStr(Self.Now);
end;
function TExtTimeKeeper.GetTodayName: string;
begin
  Result :=   SysUtils.LongDayNames[
    SysUtils.DayOfWeek(Self.Date)]
end;
// Simplified functional interface
function TimeKeeper: TExtTimeKeeper;
begin
  Result := TExtTimeKeeper(TExtTimeKeeper.Instance);
end;
initialization
  // Register ourselves as the new TTimeKeeper class
  TExtTimeKeeper.OverrideSingletonClass(
    TTimeKeeper, TExtTimeKeeper);
end.
```

➤ *Listing 9*

It is not very exciting in itself, but it illustrates that the classes work as intended and that the security measures we added to avoid illegal use of the Singleton classes, serve their purpose.



Conclusion

Singletons are of the simplest kind of design patterns. Even so, implementing one in Object Pascal has been a rewarding and learning experience. Language features such as class methods and class references have helped us a long way, while the lack of class fields forced us to work a little harder. In the future we might look at other design patterns and see how they can be implemented using Object Pascal.

Hallvard Vassbotn is a Senior Software Developer at Reuters Norge AS, Falcon R&D. You can reach him at hallvard@falcon.no