The Factory Pattern

An example of using design patterns in Delphi

by Peter Hinrichsen

If you only get one thing from reading this article, then I hope it will be an awareness of the growth of design patterns. Specifically, I hope that you will find that the factory pattern is a technique which helps you to write better, more reusable and more easily maintainable, applications.

I was first introduced to the factory pattern by Mark Richards, who was acting as mentor on a multi-tier project I was involved in. Over the past six months, I have used the factory pattern to get me out of dozens of tight corners and will share with you what I have learnt in this article.

The Factory Pattern

The factory pattern is a creational pattern that enables you to delay choosing the class of an object until runtime. As a result, any situation where there is more than one way an object can be created and used is a candidate for the factory pattern.

I believe that, in programming, there are only three integers you need to worry about: zero, one and *n*. Using the example of a system that lets a user choose a report to run, if there are zero possible reports, then we can very quickly move onto something else to do. If there is only one report to be run, then we can write a single `OnClick` event handler behind a button or menu item to execute the report.

However, if there is more than one report to run, the usual approach is to develop some logic to let the user choose their report, then to write some more logic so our application will execute the chosen report. It is this logic that can often cause us to write the `if... then... else...` statement from hell.

The factory pattern solves this problem by introducing some very flexible and powerful logic that chooses the object to create.

The Procedural Approach

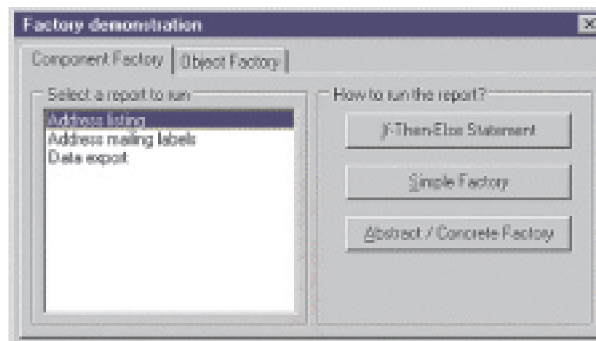
To demonstrate the advantages of using the factory pattern to create objects, we will first look at some of the drawbacks of writing procedural code to choose which object to create.

This shall be done using the application described above, which lets the user select from a list of reports via the user interface. The current requirement is for just three reports (address listing, mailing labels and a data export), so we shall use an `if... then... else...` statement to sort out which report the user has selected. Our application's main form is shown in Figure 1.

To build this rather trivial example we will first create a list of reports to display, so the user can choose which report to run. We define the report names as constants, so that they can be easily referenced in code, then add them to a listbox. This is done in the report selection form's `OnCreate` event, as shown in Listing 1.

➤ *Listing 1*

```
// Fill the listBox with possible report names
procedure TFormReportSelection.FormCreate(Sender: TObject);
begin
  with lbReports.Items do begin
    Clear;
    Add(cgStrReportNameListing);
    Add(cgStrReportNameMailingLabels);
    Add(cgStrReportNameDataExport);
  end;
end;
```



➤ *Figure 1*

The constants are defined in their own unit, constants.pas. It becomes important to break code into smaller units when an application is written for multi-tier.

In the `OnClick` event handler of a button or menu choice, we have the code shown in Listing 2.

We then create a hierarchy of reports descending from the abstract class `TReportAbstract`. It is important to understand the use of abstract and concrete classes when you are using the factory pattern and we shall focus on this in greater detail later. `TReportAbstract` is a descendent of `TForm` and implements the additional method `Execute`. This method has the single command `ShowModal` in its body. The descendent forms add additional functionality to `Execute` in order to to build the report as required. Figure 2 shows the UML diagram for the report class hierarchy.

Once we have created the appropriate instance of `TReportAbstract`, we hit its `Execute` method inside a `try...finally...` block.

Disadvantages Of The Procedural Approach

This code is clean and readable but, as the number of reports grows, it will become harder and harder to maintain. With each report that is added it is necessary to create an additional constant,

```
// Run the selected report, using the If-Then-Else
// programming style
procedure TFormReportSelection.btnIfThenElseClick(
  Sender: TObject);
var lStrReportName : string ; lReport : TFormReportAbstract;
begin
  // If no report is selected, then raise an exception
  if lbReports.ItemIndex = -1 then
    Raise Exception.Create( 'Please select a report',
      mtError, [mbOK], 0) ;
  // Read the selected report name into a string variable
  lStrReportName := lbReports.Items[lbReports.ItemIndex];
  // The If-Then-Else statement
  if lStrReportName = cgStrReportNameListing then
  begin
    lReport := TFormReportNameListing.Create( nil ) ;
  end else if lStrReportName = cgStrReportNameMailingLabels
    then begin
    lReport := TFormReportAddressLabel.Create( nil )  ;
  end else if lStrReportName = cgStrReportNameDataExport
    then begin
    lReport := TFormReportDataExport.Create( nil ) ;
  end else begin
    // more else if lStrReportName = ??? for each report...
  else begin
    raise Exception.Create('Cannot run report. '+
      'Report name <' + lStrReportName + '> not known. '+
      'Called in ' + ClassName + '.btnIfThenElseClick.') ;
  end ;
  // Execute the report
  try
    lReport.Execute ;
  finally
    lReport.Free ;
  end ;
end ;
```

Listing 2

add it to the `TListBox` and synchronise the `if... then... else...` statement. The compiler will not catch a mismatch between the `TListBox` and the `if... then... else...` statement. As an example, I am working on a system with 56 reports: once the number grew beyond about 12, the code became difficult to maintain and expand. I restructured the system to be factory based, which solved most of our maintenance problems.

Factory Pattern To The Rescue

The factory pattern is the solution to the bloated `if... then... else...` statement. We create a factory and register all the possible reports. When we want to run a report, we pass the factory the report name and it returns the appropriate report object. We simply fire the report's `Execute` method then free the object when we are done.

The report factory comprises three objects working together: the `TReportFactory`, `TReportMapping` and `Class-Reference`.

The main object is the `TReportFactory`, which contains a `TList` of `TReportMappings`. There must be one `TReportMapping` object in the list for each report the factory will produce. As well as the usual constructor and destructor, the `TReportFactory` has the procedure `RegisterReport` and the function `GetReport`.

The `TReportMapping` class is simply a container for information about how to create a report object. It has two properties, `ReportName` and `ReportClass`. `ReportName` holds a string which is used to identify the report. These strings are defined as constants in a separate source file so they are easily accessible. `ReportClass` is a `Class-Reference` type and is used to identify the class of the report that the Factory will produce.

`Class-Reference` types are discussed in the Delphi Help under *Class of*. The Help tells us that `Class-Reference` types are useful when you want to invoke a constructor (like `TObject.Create`) on a class or object whose actual type is unknown at compile time. This means that we can execute code like that shown in Listing 3.

Implementing The Report Factory

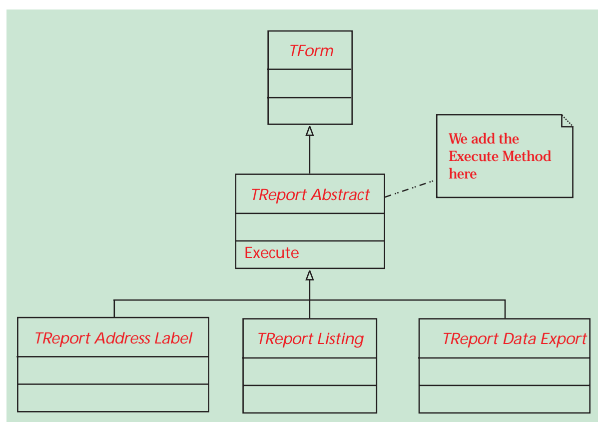


Figure 2

We will examine the `Class-Reference`, then the `TReportMapping`, followed by the `TReportFactory`, which will tie all the pieces together.

`Class-Reference` is the first declaration in the `type` section of our `FactoryReport` unit.

```
MyClassName := TReportListing;
MyReport :=
  MyClassName.Create(nil);
// An instance of TReportListing
MyReport.Execute ;
```

Listing 3

```
unit FactoryReport;
interface
uses
  // For TObject:
  Classes,
  // For TReportAbstract:
  FReportAbstract;
type
  TReportClass =
    class of TFormReportAbstract;
```

Listing 4

Listing 4 shows how this is done in code.

Remember to include `Classes` (for access to `TObject`) and `FReportAbstract` (for access to `TReportAbstract`) in the `interface` section's `uses` clause.

Next, we declare the `TReportMapping` object. The code for this is shown in Listing 5.

The `TReportMapping` object is a simple holder for two pieces of information. The `ReportName`, which was defined in the file constants.pas, and `ReportClass`, which is a `Class-Reference` that identifies the type of object to be created by the factory. The constructor we call is called `CreateExt` and takes two parameters, the `ReportName` and the `Report Class-Reference`. The principle of renaming a constructor that has a different signature (or list of parameters) to the default was introduced to me by Mark Richards. This is a handy technique to overcome the complexities of method overloading (having two

```
TReportMapping = class(TObject)
  private
    FStrReportName : string ;
    FReportClass : TReportClass ;
  public
    Constructor CreateExt(const pStrReportName: string; pClassRef: TReportClass);
    property ReportName  : string read FStrReportName write FStrReportName ;
    property ReportClass : TReportClass read FReportClass write FReportClass ;
  end ;
```

➤ *Above: Listing 5*   ➤ *Below: Listing 6*

```
constructor TReportMapping.CreateExt(const pStrReportName: string;
  pClassRef: TReportClass);
begin
  Create ;
  ReportName  := pStrReportName ;
  ReportClass := pClassRef ;
end;
```

methods with the same name, but different signatures). The implementation (the code inside the method) of CreateExt is shown in Listing 6.

The main trap for inexperienced players here (it had me scratching my head for a while) is that we use Call Create, not Inherited Create, in the body of CreateExt. Although they will both perform the same task in this instance, if we had some custom code in the overridden Create, it would not be executed if we called Inherited Create.

To understand how the pieces are brought together, read through the interface code in Listing 7.

Firstly, we have a private TStringList variable, FReportMappings, which is used to hold a list of ReportNames and TReportMapping objects. FReportMappings is created in the constructor and destroyed in the destructor of TFactoryReport. There is no rocket science here, except we must remember to Free all the objects we added to the Objects property of FReportMappings. The constructor and destructor of TFactoryReport are shown in Listing 8.

The next method we need to know about is RegisterReport which is called in the initialization section of the units that contain the report definitions. RegisterReport is called once for each report to be registered with the Factory. The constant, ReportName and report's Class-Reference are passed as parameters. The source code of RegisterReport is shown in Listing 9. RegisterReport scans the ReportMappings string list looking for the report name. If the report name is found, then the report is already registered and an error is reported. We do not use Assert, or raise an exception here as RegisterReport will most likely have been called from within a unit's initialization section. Delphi's exception handling mechanism may not have been created yet and if an exception is raised here, the fairly meaningless 216 error (*Access Violation*) is reported.

Once we have done our preemptive error checking, we are able to create the ReportMapping object and add it to the list.

The method GetReport is used to create the instance of TFormReportAbstract we have requested. The first few lines of code in GetReport check that the report has been registered with the factory and raise an exception if it can not be found. (An exception is all right here, as we will be out of the initialization sections of the application's code and the application will be executing normally). If the report name is found in the list, we are able to call the constructor against the associated Class-Reference variable and create an instance of the report we have requested.

Finally, we must create an instance of the report factory. As we only want a single instance of the factory, we could implement it as a singleton using one of the techniques described in either of the articles in Issues 41 or 44. Personally, I prefer to use the 'poor man's singleton' (illustrated in Listing 11), by creating a globally visible function gFactoryReport, which surfaces the unit-wide variable uFactoryReport.

The function gFactoryReport first checks to see if uFactoryReport is unassigned and, if so, it creates an instance. The function gFactoryReport is called once in the initialization section of the factory's unit, so an instance is always available. The variable uFactoryReport is freed in the finalization section of the unit. This technique is so rough it hardly deserves to be called a singleton; however, it does work.

Now that we have created our ReportFactory object, we can register reports and call the factory to

```
TFactoryReport = class(TObject)
  private
    FReportMappings : TStringList;
  public
    Constructor Create;
    Destructor Destroy; override;
    Procedure RegisterReport(const pStrReportName: string;
      pClassRef: TReportClass);
    Function GetReport( const pStrReportName : string ) : TFormReportAbstract;
  end;
```

➤ *Above: Listing 7*   ➤ *Below: Listing 8*

```
constructor TFactoryReport.Create;
begin
  inherited ;
  // Create a TSTringList to hold the ReportMappings
  FReportMappings := TStringList.Create;
end;
destructor TFactoryReport.Destroy;
var
  i : integer ;
begin
  // Scan through FReportMappings and free any associated objects
  for i := 0 to FReportMappings.Count-1 do
    TObject(FReportMappings.Objects[i]).Free ;
  FReportMappings.Free ;
  inherited ;
end;
```

create report objects. To register a report, call:

```
gFactoryReport.RegisterReport(
  cgStrReportNameListing,
  TFormReportListing);
```

and to call the factory to create an instance of the report use:

```
lReport :=
  gFactoryReport.GetReport(
  lStrReportName);
```

The UML for the factory we have just created is shown in Figure 3.

Abstract And Concrete Factories

Many of you will now be thinking that this is great, but there is little code reuse from this implementation of the factory pattern.

To demonstrate code reuse, I have rewritten this example on the companion disk as an abstract factory, which builds `TObjects` or `TComponents`, and a concrete factory, which creates the `TFormReportAbstract` descendants.

I have also written a `TFactoryConcreteAnimal` which creates `TObject` descendants and caches them inside the factory. This demonstrates how `TObjects` and `TComponents` must be handled differently, as well as the technique of caching objects within the factory.

Caching objects within the factory is a useful technique where the time required to create the object is significant. An example use of the cached factory may be when you are creating `TQuery` components and preparing them to improve database performance. The prepared queries can be built using the factory and cached for later use. The UML diagram for the abstract and concrete factories is shown in Figure 4.

Other Uses

Factories may be used in any situation where you want to delay the choice of object you shall create until runtime. For example, Delphi itself also uses factories internally to produce remote data modules and other COM related objects.

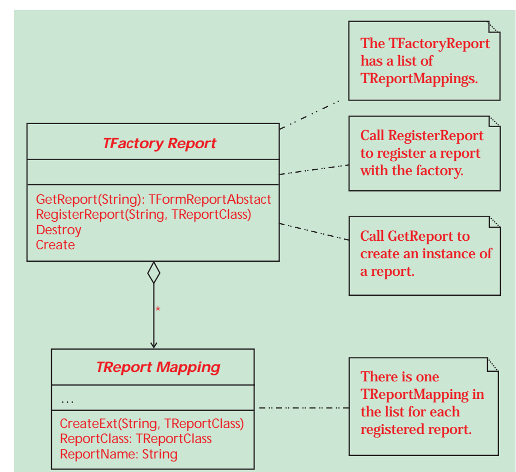
Some other uses of factories include: object streaming in COM, DCOM or multi-tier, MIDAS applications, file based applications where a user can choose to save a data file in a variety of formats, and applications with many user configuration options. I leave the rest to your imagination!

```
procedure TFactoryReport.RegisterReport(const pStrReportName: string;
  pClassRef: TReportClass);
var
  i : integer ;
  lReportMapping : TReportMapping ;
  lStrReportName : string ;
begin
  lStrReportName := upperCase(pStrReportName) ;
  // Does the reportName already exist?
  i := FReportMappings.IndexOf( lStrReportName );
  // If yes, then raise an exception.
  if i <> - 1 then begin
    messageDlg( 'Registering a duplicate report name <' + pStrReportName + '>',
      mtInformation, [mbOK], 0);
  // The report exists in the list
  end else begin
    // Create a reportMapping object
    lReportMapping := TReportMapping.CreateExt(lStrReportName, pClassRef ) ;
    // Add the reportName, and reportMapping object to the list
    FReportMappings.AddObject( upperCase( pStrReportName ), lReportMapping ) ;
  end ;
end;
```

➤ Above: Listing 9

➤ Below: Listing 10

```
function TFactoryReport.GetReport(const pStrReportName: string):
  TFormReportAbstract;
var
  i : integer ;
begin
  // Does the report exist in the list?
  i := FReportMappings.IndexOf( upperCase(pStrReportName));
  // If not, report an error
  if i = -1 then begin
    Raise Exception.Create('Request for invalid report name <' +
      pStrReportName + '>' + #13 + 'Called in ' + ClassName + '.GetReport') ;
  end;
  // Create an instance of the report, and return. Note that the module that
  // called GetReport is responsible for freeing the report.
  result := TReportMapping(FReportMappings.Objects[i]).ReportClass.Create(nil);
end;
```



➤ Figure 3

➤ Listing 11

```
// The poor man's singleton, a function to return
// a reference to a variable with unit wide visibility.
function gFactoryReport : TFactoryReport;
implementation
uses
  SysUtils;  // for UpperCase
var
  // A variable to hold our single instance of TFactoryReport. This variable has
  // unit wide scope, hence the u prefix.
  uFactoryReport : TFactoryReport;
// Our poor man's singleton. This function has global (or application) wide
// scope, hence the prefix g
function gFactoryReport : TFactoryReport ;
begin
  // If uFactoryReport has not been created, then create one.
  if uFactoryReport = nil then
    uFactoryReport := TFactoryReport.Create ;
  // Return a reference to uFactoryReport
  result := uFactoryReport ;
end ;
{...Code...}
initialization
  gFactoryReport;
finalization
  uFactoryReport.Free;
```

Conclusion

I have found the factory pattern to be a powerful technique for solving a number of difficult programming situations. I hope you are able to put it to good use.

The starting point for more information is the book *Design Patterns*, by Gamma, Helm, Honhson and Vlissides. Another good reference is *UML Distilled* by Martin Fowler. The useful Delphi Pattern website is at www.burn-rubber.demon.co.uk/patterns.htm.

Acknowlegements

Mark Richards is a Delphi System Architect and introduced me to the factory pattern while providing a mentoring service during the design and construction of a multi-tier application in Delphi. Mark is at mr@richdata.com.au.

Peter Hinrichsen is an analyst/programmer from Melbourne, Australia, specialising in applications for telecoms, banking and finance. Email him at peter_hinrichsen@techinsite.com.au

➤ *Figure 4*