Simple Factory Pattern

Graeme Geldenhuys

2008-08-02

In this article I am going to discuss one of three Factory design patterns. The Factory patterns are actually subtle variations of each other and all fall under the Creational Patterns category. I will show you a real-world example of the Simply Factory implemented in Object Pascal, that demonstrates how that pattern can save you time and make your code more maintainable and extensible.

The Procedural Approach

To demonstrate the advantage of the factory pattern we will first look at the procedural approach and then discuss the disadvantages of this approach and how the factory pattern comes to the rescue. The example I am going to use is one that appears often in many real-world applications. Reporting is a common feature in today's business applications. For this trivial example our user interface presents a list of available reports in a ListBox component. The user can select a single report to preview. Figure 1 shows how our project's main form looks like

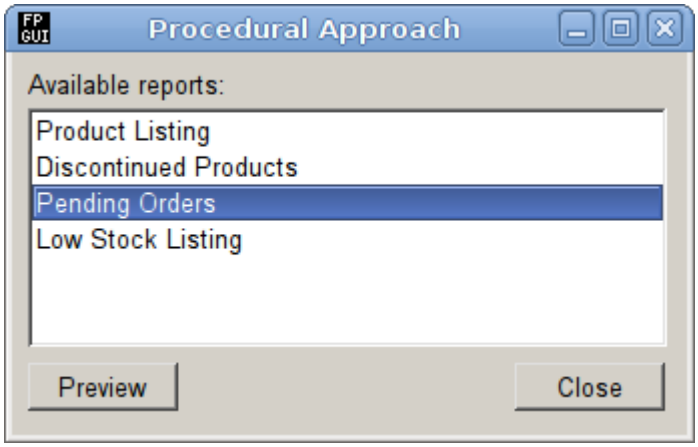

Figure 1: Our example project's Main Form.

The steps to build such a screen would be to first populate the ListBox component with the names of the available reports. The code to accomplish that is shown in Listing 1.

```
Listing 1:
// Fill the listbox with available report names
procedure TMainForm.BuildReportList;
begin
  with lstReports.Items do
  begin
    Clear;
    Add(cProductListing);
    Add(cDiscontinuedProducts);
    Add(cPendingOrders);
    Add(cLowStockListing);
  end;
end;
```

The reports' names are defined as string constants in a separate unit called *constants.pas*. This allows us to easily reference the report names in code. To simplify our example project we don't actually use a reporting engine for our reports, we simply shows a window containing a label component with some text and the correct report name in the window title. Each concrete report class derives from the abstract `TBaseReport` class which in turn derives from a standard Form. The concrete reports implement the specific reporting logic for each report, by overriding the `Execute` method introduced in the `TBaseReport` class. Figure 2 shows the report hierarchy we are going to create in this article.

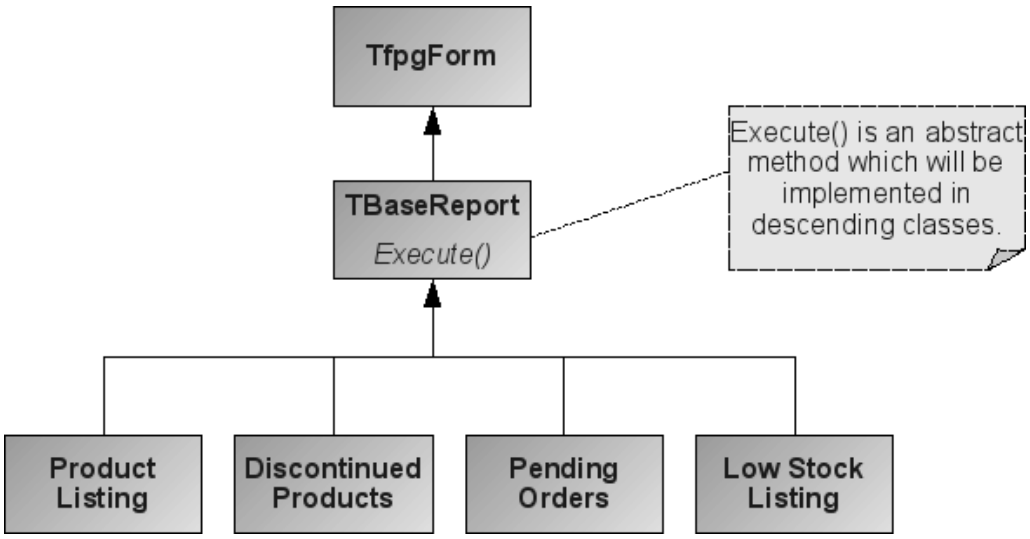

Figure 2: Hierarchy of our report classes

Now we need to implement the Preview button's OnClick event handler. It will contain the `if...then...else` code block that will figure out which report the user has selected and create the correct report instance.

Listing 2:

```
procedure TMainForm.btnPreviewClicked(Sender: TObject);
var
  rptfrm: TBaseReport;
  lReportname: string;
begin
  if lstReports.FocusItem < 0 then
    raise Exception.Create('Please select a report'
      + ' and try again.');
  lReportname := lstReports.Text;
  if lReportname = cProductListing then
    rptfrm := TProductListingReport.Create(nil)
  else if lReportname = cDiscontinuedProducts then
    rptfrm := TDiscontinuedProductsReport.Create(nil)
  else if lReportname = cPendingOrders then
    rptfrm := TPendingOrdersReport.Create(nil)
  else if lReportname = cLowStockListing then
    rptfrm := TLowStockListingReport.Create(nil)
{
  // many more else/if pairs go here
  else if lReportname = ???? then ...
  ...
}
  else
    raise Exception.Create('Unknown report <'
      + lReportname + '>');
  try
    rptfrm.Execute;
  finally
    rptfrm.Free;
  end;
end;
```

Listing 2 shows the bulk of our code to create and display our reports. First we make sure the user did select a report in the ListBox, otherwise we raise an exception. If we passed the first test we can move on by storing the selected report name in a local variable which we use in the `if...then...else` code block. The main logic does a comparison between the report name we got from the ListBox and the existing report name constants. If we find a match, we create an instance of that report and store it in the local `rptfrm` variable. You will notice that the `rptfrm` variable is of the generic type `TReportBase`. This allows us to store any object instance that derives from `TReportBase`. We then call the `Execute` method which internally sets up the report and finally calls `ShowModal` to display the report. Once we are done viewing the report, we free the report instance.

Now lets look at some disadvantages to using the procedural approach to create and display reports. If your applications are anything like in our company, you end up with many reports. As an example, our company's latest project already has around 40 reports and growing. This very quickly becomes a nightmare to maintain, when using the procedural approach. Every time we add a new report, we would have to follow these steps.

- Create a new report name constant.
- Manually update the `BuildReportList` method with our new report name.
- We would also have to manually update the `if...then...else` block in the Preview button's OnClick event handler to create the correct report instance.

- Plus we would have to include the unit containing our new report in the implementations section's uses clause.

As you can see, this is a lot of work and a maintenance nightmare. Plus it's prone to errors. For example, we could very easily remove a report from the OnClick event hander's `if...then...else` block, but forget to remove the report name from the ListBox in the `BuildReportList` method.

Simple Factory Pattern to the rescue

The Simple Factory approach solves most of these maintenance problems for us. Even though this design pattern is not one of the 23 documented patterns in the GoF (Gang of Four) book[1], it is a frequently used pattern. The Simple Factory returns an instance of one of several possible classes, depending on the data supplied to it at runtime. Normally those returned classes derive from the same base class, but act on the data in a different way. This sounds perfect for our reporting example. This pattern is similar to the Factor Method pattern, but with a subtle variation in it's implementation.

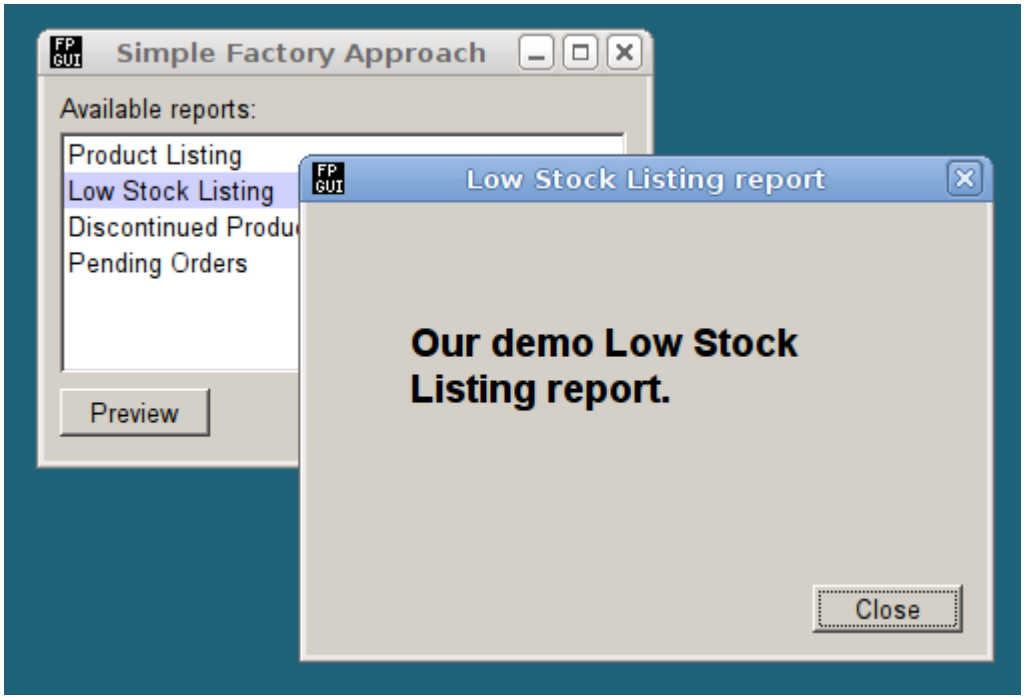

Figure 3: The Factory approach showing one of our sample reports.

To implement our Report Factory, we are going to need three classes that work together. A *class-reference type*, a *mapping class* and the *factory class* itself.

We will start off by declaring a class-reference type, `TReportClass`, as shown in Listing 3. A *class-reference type* is used when you need to perform an operation on a class itself, rather than on instances of a class. One popular example of this is when you want to invoke a constructor (like `TObject.Create`). In our example, we can't refer to the specific class type, because we don't know what report we want to create at compile

1 Design Patterns Elements of Reusable Object-Oriented Software.

time. That information will only be available at runtime, and the class-reference variable will at that point identify the report class we want to produce.

Listing 3:
```
type
  TReportClass = class of TBaseReport;
```

The next class we need, will be the `TReportMapping` class, as shown in Listing 4. The `TReportMapping` class is simply a container class with information that the Factory needs when it wants to create a report. It has two properties, `Name` and `ReportClass`. The `Name` property holds a string value which is used to identify the report. The `ReportClass` property is a class-reference type, and is used to identify the class of the report that the Factory will produce.

Listing 4:
```
TReportMapping = class(TObject)
private
  FName: string;
  FReportClass: TReportClass;
public
  constructor CreateEx(const AName: string;
    const AReportClass: TReportClass);
  property Name: string read FName;
  property ReportClass: TReportClass read FReportClass;
end;
```

The remaining part of this class is the constructor. It's implementation is shown in Listing 5. We use a *custom* constructor named `CreateEx` that takes two parameters. The first parameter, `AName`, is the name of the report. We will assign one of the report name constants, as defined in the *constants.pas* unit, to that parameter. The second parameter, `AReportClass`, is the class-reference that identifies the type of the object the Factory must produce.

The reason I said it is a custom constructor, is because it doesn't have the default constructor name `Create`, so we don't have to worry about overloading methods. A word of caution though. Because we defined a custom constructor, doesn't mean the default `Create` constructor is hidden or doesn't exist any more. It still exists, and that is why we call `Create` inside our constructor, instead of `inherited Create`, so that in case we defined some code in the default constructor, our version will still allow that code to be executed.

Listing 5:
```
constructor TReportMapping.CreateEx(const AName: string;
  const AReportClass: TReportClass);
begin
  Create;  // don't call inherited Create!
  FName := AName;
  FReportClass := AReportClass;
end;
```

The final class that we need, to complete our Factory implementation, is the `TReportFactory`, which contains a list of `TReportMapping`s. The Factory class declaration is shown in Listing 6. The internal list is a private field variable, `FMappings`, of type

`TStringList`. It will contain one instance of `TReportMapping` for each report that the Factory will produce. The Factory class also introduces three methods that we will need. The `RegisterReport`, the `CreateReport` and the `GetRegisteredReports` methods.

Listing 6:
```
TReportFactory = class(TObject)
private
  FMappings: TStringList;
public
  constructor Create;
  destructor Destroy; override;
  procedure RegisterReport(const AName: string;
    const AReportClass: TReportClass);
  function CreateReport(const AName: string): TBaseReport;
  procedure GetRegisteredReports(var AList: TStringList;
end;
```

The `RegisterReport` method, as the name suggests, is used to register a new report with the Factory. Normally this method is called from the *initialization* section of the units containing the report definitions. It is only called once per report. Listing 7 shows how we call this method.

Listing 7:
```
initialization
  gReportFactory.RegisterReport(
    cPendingOrders, TPendingOrdersReport);
```

The implementation of `RegisterReport` is show in Listing 8. `RegisterReport` scans the string list looking for an existing entry using the report name constant passed in via the `AName` parameter. As I mentioned before, a report can only be registered once, so if it finds a entry it raises an exception with an appropriate message. If nothing was found it can continue by creating a `TReportMapping` instance and set the report name and report class. The report class was passed in via the `AReportClass` class-reference parameter. The newly created `TReportMapping` instance is then added to the internal string list.

Listing 8:
```
procedure TreportFactory.RegisterReport(
  const AName: string; const AReportClass: TReportClass);
var
  i: integer;
  lMapping: TReportMapping;
begin
  i := FMappings.IndexOf(UpperCase(AName));
  if i <> -1 then
    raise Exception.Create('Registering a duplicate report'
      + ' name <' + AName + '>')
  else
  begin
    lMapping := TReportMapping.CreateEx(AName, AReportClass);
    FMappings.AddObject(UpperCase(AName), lMapping);
  end;
end;
```

The `CreateReport` method will return an instance of `TBaseReport`. The implementation of `CreateReport` is shown in Listing 9. `CreateReport` first scans the internal list for a registered report by the name `AName`. If nothing is found, it raises an exception with an appropriate error message. On the other hand, if it did find a match, it extracts the `TReportMapping` information from the internal list entry. It uses the `ReportClass` class-reference to create an instance of the appropriate report and returns that instance.

Listing 9:
```
function TReportFactory.CreateReport(const AName: string):
  TBaseReport;
var
  Idx: integer;
  lRptClass: TReportClass;
begin
  Idx := Fmappings.IndexOf(UpperCase(AName));
  if Idx = -1 then
    raise Exception.Create('No report was registered by'
      + ' the name <' + AName + '>')
  else
  begin
    lRptClass :=
      TReportMapping(FMappings.Objects[Idx]).ReportClass;
    Result := lRptClass.Create(nil);
  end;
end;
```

The last method in the Factory we need to cover, is the `GetRegisteredReports` method. This method is not critical to the functionality of the Factory. It is more a convenience method and is used to return a list of registered reports. This method is used to populate the ListBox component with the available registered reports. The source code for `GetRegisteredReports` is shown in Listing 10. We pass in the `Items` property of the ListBox component as the `AList` parameter to `GetRegisteredReports`. `GetRegisteredReports` clears the content of `AList`, and then loops through the internal list of the Factory adding the names of the registered reports to the `AList` string list. Nothing fancy.

Listing 10:
```
procedure TReportFactory.GetRegisteredReports(
  var AList: TStringList);
var
  i: integer;
begin
  AList.Clear;
  for i := 0 to FMappings.Count - 1 do
    Alist.Add(TReportMapping(FMappings.Objects[i]).Name);
end;
```

Now all that remains is to create an instance of the Factory. For that we use the Singleton design pattern. We want to ensure we only have one instance of the Factory in our application. We will use a very basic implementation of the Singleton pattern and it works quite well for our needs. You have already seen me use the singleton in Listing 7. The singleton is defined as the globally visible `gReportFactory` function. The source code for our Singleton is shown in Listing 11. We define the `gReportFactory`

function in the *Interface* section of the Factory unit, making in globally visible to other units. It will return a single instance of the `TReportFactory` class, even if we call it multiple times. The way we accomplish that is as follows. We have a unit visible variable, `uReportFactory`, of type `TReportFactory` defined in the *Implementation* section of the Factory unit. This variable will hold our Factory instance. We initialise it to nil in the initialization section of the Factory unit. We also need to take care of freeing the `uReportFactory` object once our application terminates. We handle that in the `finalization` section of the Factory unit, by calling `uReportFactory.Free`. The `gReportFactory` funcion itself is quite simple by design. When it is called, it first checks to see if the `uReportFactory` variable has already been assigned. If it is unassigned, it creates a instance of `TReportFactory` and stores it in the `uReportFactory` variable. It then simply returns the `uReportFactory` variable. As I said, this is a very basic implementation of the Singleton pattern, but it work quite well.

```
Listing 11:
function gReportFactory: TReportFactory;
begin
  if not Assigned(uReportFactory) then
    uReportFactory := TReportFactory.Create;
  Result := uReportFactory;
end;
```

So to call the Factory and create a instance of a report we do the following.

```
rptfrm := gReportFactory.CreateReport(lReportname);
```

And that's it! Our implementation of the Simply Factory pattern is complete. I have found this pattern very useful in many areas of my code. I also use many other design patterns, but they are for a later article. Please note though that design patterns are not meant to be used as a code template. They are normally documented in such a way to show a design solution to a problem, but how you implement them is based on your understanding of the pattern and the problem you are trying to solve.

** The following section is meant as an inset block in the article, but can also be appended as part of the main article content. **

A small Questions and Answers session

I often get asked questions regarding the various Factory Patterns. So I thought I would take this time to summarise a few of the more frequent ones. Please not that not all the questions are specific to the Simply Factory. Some questions related the the Factory Method which is a slight variation of the Simply Factory, though their end goal is the same - creating an object instance.

How do I decide what must go into the factory class or method?
Identify the aspects that vary. Move those out to a new class we call the factory.

What is a good way of accessing the Factory from the Client code?
You could access the factory using a global Singleton class, but that makes it a bit harder to unit test your class. It's handy to pass the Factory object in as a parameter to the Client. This has the benefit that we can subclass the Factory and still pass it to the Clients. Remember we are programming to an interface (not necessarily a Object Pascal *Interface* type, but simply an interface of a class). In the Simply Factory article the Main Form was the *client* of the Factory, and we used the singleton approach because we had no unit tests to worry about.

Why can't we use a class method to create the products?
It has the disadvantage that we can't subclass the factory and change the behaviour of the create method.

What's the advantage of the Simple Factory? It looks like we are simply pushing the problem off to another class.
Remember the factory can have many clients. If we add new products, only one place in the code needs to change.

What is the difference between the Simple Factory and the Factory Method? They are so similar.
Simple Factory is a one shot deal. It gives you a way to encapsulate object creation (things that vary), but doesn't give you the flexibility of the Factory Method because there is no way to vary the products you're creating. With the Factory Method you are creating a framework that lets the subclasses of the factory decide which implementations to use. The subclasses can decide what concrete products to return.