Chapter 13 Factory Patterns

Main concepts

– The Factory Method Pattern: create dynamically configurable objects.
– The Abstract Factory Pattern: create dynamically configurable families of objects.
– Polymorphism in OO designs.
– Using parallel linked hierarchies.

Chapter contents

Introduction

As we mentioned in the introduction to chapter 11, a class's structure is fixed at *compile* time. Since there are occasions when a class needs to take on different characteristics at *run* time, chapter 11 explored the Template Method and Strategy pattern as techniques to accommodate variations in an object's *behaviour*.

This chapter explores the Factory patterns, which, instead of changing an object's behaviour, provide ways of creating different *objects* to suit run time requirements. To clarify the context for factory patterns, we'll quickly review the chapter 11 patterns.

In a Template Method Pattern the client wants:

1. the result of a particular operation which has fixed steps, but
2. where the details of individual steps vary depending on context (either between subclasses or between applications), and possibly
3. where different variations of the steps may have to be added in the future.

In a Strategy Pattern the client wants the result of an entire operation, rather than the individual steps of the operations, to depend on a particular context.

However, in the Factory Method Pattern discussed in this chapter the client wants the actual object that is being instantiated, and not just its behaviour, to depend on context. In the Abstract Factory pattern, an entire family of objects is created dependent on context.

Factory patterns are widely used, and there are several variations. The examples that follow clarify the general principles and the conditions under which they are appropriate.

Example 13.1 A simple Factory Method

A Factory is an important concept in OO programming. Stated briefly, a Factory is an object that creates an unrelated object on behalf of another unrelated object. As the following example shows, the intent of using a Factory is to reduce the coupling between classes, and to make a class more reusable by making it independent of other classes.

We'll start with a simple example and then go on to explore an Abstract Factory in example 13.2.

Ex 13.1 step 1 Without the Factory Method

We'll extend the furniture program of example 10.2 to use a Factory Method. Reload that project and simplify the user interface a bit from example 10.2 step 3 by removing the two buttons and their associated event handlers and modifying the RadioGroup's event handler as follows (line 35 below):

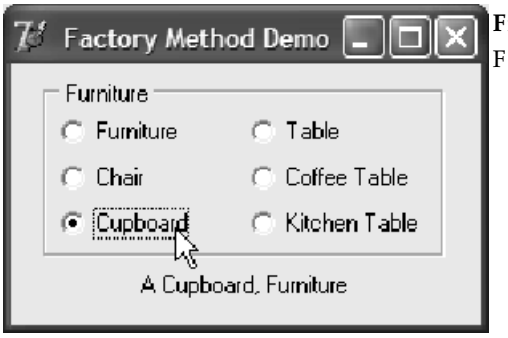

Figure 1 Driver for the simple Factory Method

Figure 2 Objects on the user interface

```
24 procedure TfrmFactoryMethod.rgpFurnitureClick(Sender: TObject);
25 begin
26   FreeAndNil (MyFurniture);
27   case rgpFurniture.ItemIndex of
28     0: MyFurniture := TFurniture.Create;
29     1: MyFurniture := TChair.Create;
30     2: MyFurniture := TCupboard.Create;
31     3: MyFurniture := TTable.Create;
32     4: MyFurniture := TCoffeeTable.Create;
33     5: MyFurniture := TKitchenTable.Create;
34   end;
35   lblKind.Caption := MyFurniture.GetKind;
36 end; // end procedure TfrmFactoryMethod.rgpFurnitureClick
```

Make these changes and check that the program works before continuing. (You could add a Try … Except construct around lines 27 to 35 in case options are added to or removed from rgpFurniture in future without matching changes in code.)

Ex 13.1 step 2 Separation of concerns

We have talked in the past about the principle of the *separation of concerns* – the user interface should not be concerned about the details of the application classes and *vice versa*. But here the user interface is very bound up with the application classes. In lines 28 to 33 it needs to know that the application classes include a TFurniture, a TChair, and so on. So the user interface is responsible for two things: it must determine which piece of furniture is involved, and then it must associate this with a particular class. It would be better to separate these concerns, letting the user interface simply determine the required physical object and letting some other class decide which application object this physical object refers to. In terms of actual code, we want to replace Step 1, lines 24–36 with something more cohesive:

```
24 procedure TfrmFactoryMethod.rgpFurnitureClick(Sender: TObject);
25 begin
26   FreeAndNil (MyFurniture);
27   MyFurniture:= FurnFactory.CreateFurniture(rgpFurniture.ItemIndex);
28   lblKind.Caption := MyFurniture.GetKind;
29 end; // end procedure TfrmSubstitution.rgpFurnitureClick
```

We want a single program statement, line 27, to replace the entire case statement in the previous version (step 1, lines 27–34). Let's look at line 27 to get an idea of our goal.

Instead of using rgpFurniture's ItemIndex property in the Case statement, we want to send this integer value as a parameter to a method called CreateFurniture that belongs to an object called FurnFactory. As its return value it must give us an appropriate TFurniture object. So the user interface now is responsible only for determining which piece of furniture the user has selected on the RadioGroup and then for displaying this piece of furniture's GetKind text. The user interface is not actually concerned with what the piece of furniture might be, just that it is a valid TFurniture object that it can use. (In step 1 the user interface must know that a value of 5, for example, represents a Kitchen Table (line 33). In step 2 we want to offload this responsibility to a different object, FurnFactory.)

Now we'll work through the detail of these changes.

Ex 13.1 step 3 The Factory Method

Add a new unit to the project to provide the factory class as follows:

```
1 unit FurnFactoryU;

2 interface
```

```
 3 uses FurnitureU;

 4 type
 5   TFurnFactory = class(TObject)
 6   public
 7     function CreateFurniture (ASelector: integer): TFurniture;
 8                                                 virtual; abstract;
 9   end; // end TFurnFactory = class(TObject)

10   TFurnCreator = class(TFurnFactory)
11   public
12     function CreateFurniture (ASelector: integer): TFurniture;
13                                                       override;
14   end; // end TFurnFactory = class(TFurnFactory)

15 implementation

16 { TFurnCreator }

17 function TFurnCreator.CreateFurniture(ASelector: integer):
18                                                          TFurniture;
19 begin
20   case ASelector of
21     0: Result := TFurniture.Create;
22     1: Result := TChair.Create;
23     2: Result := TCupboard.Create;
24     3: Result := TTable.Create;
25     4: Result := TCoffeeTable.Create;
26     5: Result := TKitchenTable.Create;
27   else Result := nil;
28   end;
29 end; // end function TFurnCreator.CreateFurniture

30 end. // end FurnFactoryU
```

We have created an abstract Factory Method class, TFurnFactory, and from this derived a concrete Factory Method class, TFurnCreator, which implements the CreateFurniture method. CreateFurniture effectively takes over the Case selection from the user interface (lines 20–28) on the basis of the selector it receives as an input parameter value (line 17 above). The Case creates a TFurniture of the required type and returns a reference to it to the calling routine.

The class diagram of the factory is shown in figure 3.

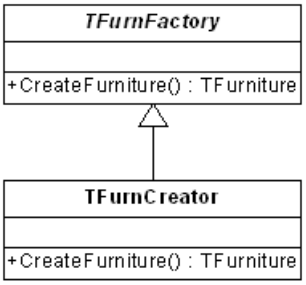

Figure 3 The Factory class structure

Ex 13.1 step 4 Using the Factory

To use the Factory, we need to add a reference to it in the user interface class definition (line 14 below and the uses clause reference to FurnFactoryU in line 6) before invoking its CreateFurniture method to create the required subtype of TFurniture (lines 24–25).

```
 1 unit FactoryDemoU;

 2 interface

 3 uses
 4   Windows, Messages, SysUtils, Classes, Graphics, Controls,
 5   Forms, Dialogs, StdCtrls, ExtCtrls,
 6   FurnFactoryU, FurnitureU;

 7 type
 8   TfrmFactoryMethod = class(TForm)
 9     lblKind: TLabel;
10     rgpFurniture: TRadioGroup;
11     procedure rgpFurnitureClick(Sender: TObject);
12     procedure FormCreate(Sender: TObject);
13   private
14     FurnCreator: TFurnCreator;
15     MyFurniture: TFurniture;
16   end; // TfrmFactoryMethod = class(TForm)

17 var
18   frmFactoryMethod: TfrmFactoryMethod;

19 implementation

20 {$R *.DFM}

21 procedure TfrmFactoryMethod.rgpFurnitureClick(Sender: TObject);
22 begin
23   FreeAndNil (MyFurniture);
24   MyFurniture := FurnCreator.CreateFurniture
25                                     (rgpFurniture.ItemIndex);
26   lblKind.Caption := MyFurniture.GetKind;         // polymorphic call
27 end; // end procedure TfrmSubstitution.rgpFurnitureClick

28 procedure TfrmFactoryMethod.FormCreate(Sender: TObject);
29 begin
30   FurnCreator := TFurnCreator.Create;
31 end; // end procedure TfrmFactoryMethod.FormCreate

32 end. // unit FactoryDemoU
```

Compare lines 21–31 above with step 1 lines 24–36. We see that the user interface now knows only that it works with a MyFurniture: TFurniture reference that has a GetKind method, and that it uses a Factory Method to create the required subtype (step 4, lines 24–25, 14–15). It is completely decoupled from the details of TFurniture and need not know all its subtypes.

Notice that FurnCreator's CreateFurniture method in line 24 effectively acts as a constructor and so it is important to free any previous version of the object in line 23.

Ex 13.1 step 5 An alternative selector

In step 4, the user interface selects the required subtype by passing an integer selector value, rgpFurniture.ItemIndex. The Factory then uses this value to create the appropriate subtype. Depending on the application, it may be better to use a logical selector. To do this we can use a logical identifier such as a string rather than hard linking to the RadioGroup's ItemIndex. We can generate an appropriate string by using the caption of the item in the RadioGroup (lines 28–29 below). If the order of items in the RadioGroup now changes, this will still continue to make the right selection (though not if any of the captions change!).

```
23 procedure TfrmFactoryMethod.rgpFurnitureClick(Sender: TObject);
24 var
25   Selector: string;
26 begin
27   FreeAndNil (MyFurniture);
28   Selector := rgpFurniture.Items[rgpFurniture.ItemIndex];
29   MyFurniture := FurnFactory.CreateFurniture (Selector);
30   lblKind.Caption := MyFurniture.GetKind;      // polymorphic message
31 end; // end procedure TfrmFactoryMethod.rgpFurnitureClick
```

Ex 13.1 step 6 Modifying the Factory Method

The factory method must now respond to a string rather than to an integer:

```
17 function TFurnCreator.CreateFurniture(ASelector: string):
18                                                          TFurniture;
19 begin
20   if ASelector = 'Furniture' then
21     Result := TFurniture.Create
22   else if ASelector = 'Chair' then
23     Result := TChair.Create
24   else if ASelector = 'Cupboard' then
25     Result := TCupboard.Create
```

```
26   else if ASelector = 'Coffee Table' then
27     Result := TCoffeeTable.Create
28   else if ASelector = 'Kitchen Table' then
29     Result := TKitchenTable.Create
30   else if ASelector = 'Table' then
31     Result := TTable.Create
32   else
33     Result := nil;
34 end; // end function TFurnCreator.CreateFurniture
```

Remember also to change the parameter type in the declaration to a string. This version of the factory method works as well as the previous version.

There is a slight irony in the Factory Method pattern. Because the pattern selects between subtypes, it depends heavily on polymorphism. Polymorphism is meant to replace long strings of conditional evaluations, yet the code above is purely a long multiple if statement! This is because the appropriate polymorphic subtype of TFurniture has not yet been created. Once it has, no Ifs or Cases are needed elsewhere in the program. The Factory method concentrates this messy logical translation in one place. If we add further objects to the program that interact with TFurniture, they need only send an appropriate value for the selector parameter through to the Factory Method and need not become involved in conditional statements or in the specific details of the TFactory subtypes.

Ex 13.1 step 7 Class diagram for the Factory Method

The class diagram for this factory method is quite intricate (figure 4).

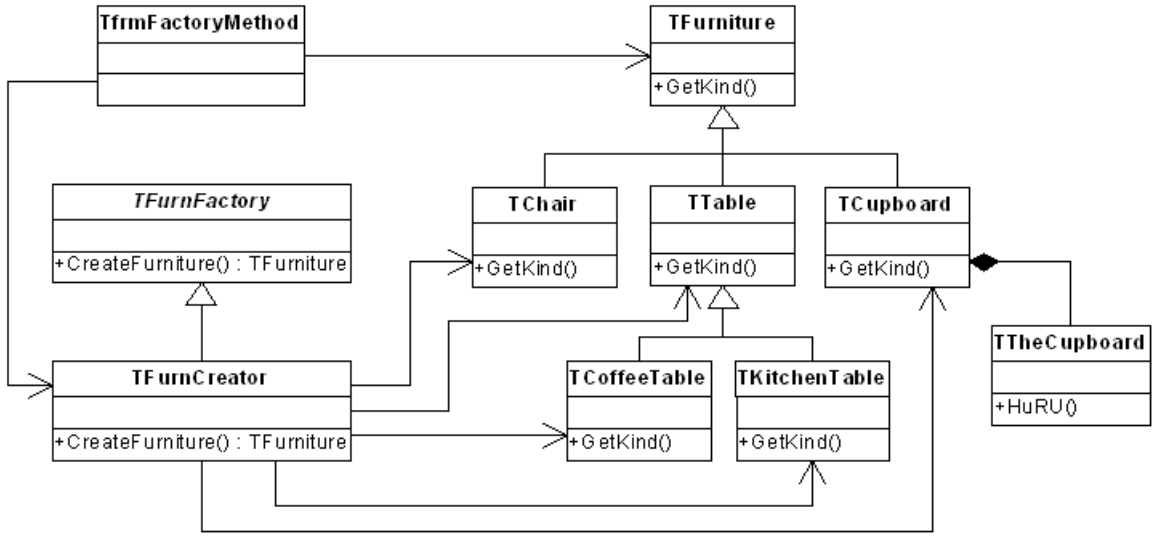

Figure 4 Class diagram for the Furniture Factory example

However, at run time this collapses to much simpler object diagrams. Assume that the user has selected the Kitchen Table. Figure 5 shows the object relationships in TFurnCreator's CreateFurniture method (step 3, line 26 or step 6, line 29).

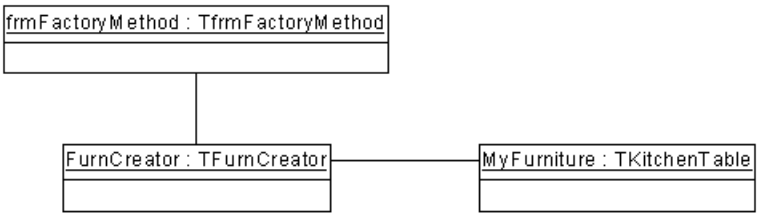

Figure 5 The Factory Method creating a concrete product

Figure 6 shows the relationships when the user interface is using the TFurniture subtype (step 4, line 26 or step 5, line 29).

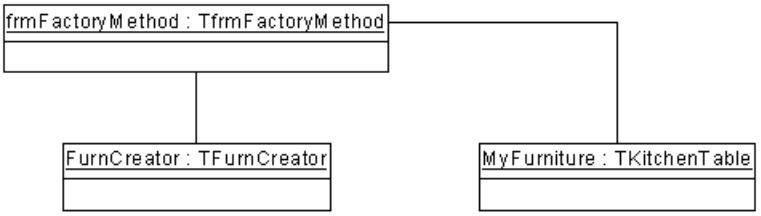

Figure 6 The context using the concrete product

Staying with the factory metaphor, and since a factory produces a product, the objects that the Factory Method instantiates, any of the Tfurniture subtypes in this case, are often referred to as Products.

Not surprisingly, one can combine patterns. Figure 4 shows, in addition to the Factory Method Pattern, the Adapter Pattern for TCupboard that we used in chapter 11.

Ex 13.1 step 8 Managing object lifetimes

It's always important to be aware of object lifetimes and the need to avoid either memory leakage or dangling references. In a set of interactions as complex as this pattern's, it helps to draw a sequence diagram to show object creation and destruction explicitly (figure 7).

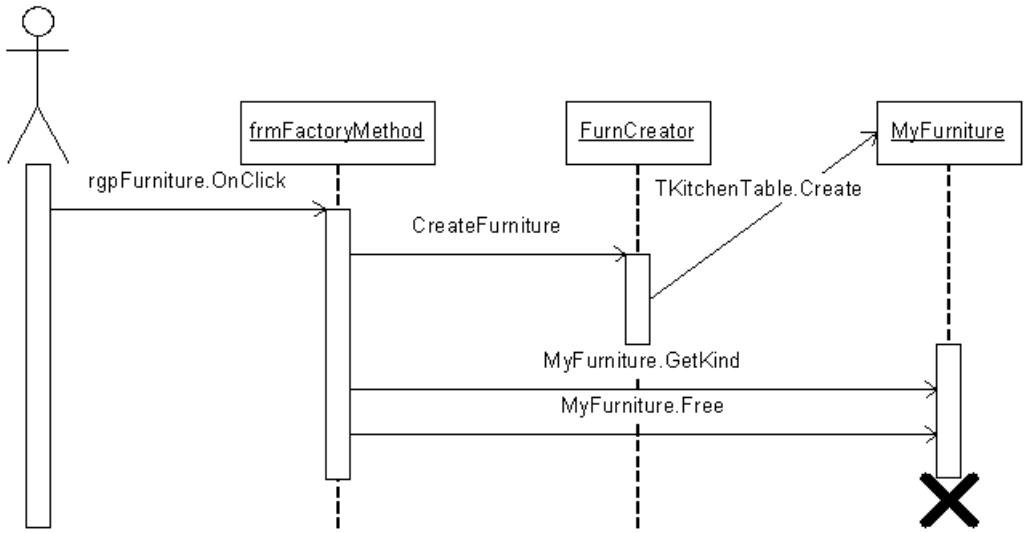

Figure 7 Creating and destroying the TFurniture subtype

Although TFurnCreator creates the TFurniture object, it knows it only as the temporary Result variable within the CreateFurniture method (steps 3 or 6). In rgpFurnitureClick TfrmFactoryMethod first destroys any existing TFurniture object before creating a new one (steps 4 or 5). If it does not do this, the previous instance remains in memory but is inaccessible, leading to memory leakage.

Is it worth the fuss?

Is the Factory Method worthwhile – after all, it adds its own complexity to a system? For a system as limited as this example, there is relatively little motivation to use it. However, as the system becomes more complex, the design issues change. The product classes become more elaborate and there are more objects using them. With the Factory Method, these product users need not concern themselves with all the specifics of the product classes. Instead this intricacy is encapsulated in a separate class, the Factory, that all the other objects can use. As the size of a program increases, an increase in complexity and coupling is unavoidable. The Factory Method is one way of structuring this complexity and of encapsulating some of it in a specific class where it is easily identifiable and where it is separated from other issues.

Example 13.2 The Abstract Factory Pattern

The Factory Method gives us a way of building a particular class, a chair or a cupboard in the example above, to a client's logical request. What happens if we need to be able to build a whole set of classes on the basis of a client's request? A bank, for example, may have a number of different types of ATM connected to its central database. The central database should not need to be concerned with the details of the different ATMs. It would be better for it to be able to initiate a set of generic interactions with the ATMs using a single selection point for the type of ATM. This can be done with a whole set of Factory Methods, an approach that is called the Abstract Factory Pattern.

Ex 13.2 step 1 The Tap Factory

For our example, we'll consider a plumbing supplier that carries different ranges of taps. The generic tap types are KitchenTap, BathTap, and so on. A house designer may wish to be able to select a particular range of Taps, possibly the Nouveau range or the Victoriana range, and from then on just refer to a KitchenTap or a BathTap knowing that it will be from the selected range. To demonstrate the approach, let's assume we start with the following interface (figure 8):

Figure 8 Abstract Factory demonstration

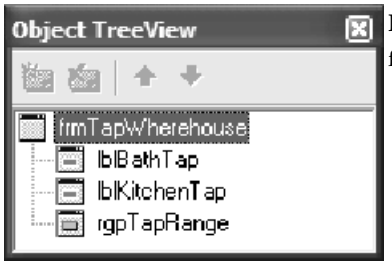

Figure 9 User interface objects for the Abstract Factory demo

The user selects a particular range of taps (Futurismo, Nouveau or Victoriana in figure 8). The user interface then selects a 'factory' that will create the objects (lines 24–29 below) and then uses this *generic factory* to create each *generic item* as it needs it (lines 31, 34) and then uses this generic object in its processing (lines 33, 36). Once the appropriate Abstract Factory is selected in lines 24–29, the user interface does not need to know which range is being used. Ranges can be added and removed and the user interface will carry on unconcernedly provided it has been given a reference to an appropriate factory object.

```
 1 unit AbsFactoryDemoU;

 2 interface

 3 uses
 4   Windows, Messages, SysUtils, Variants, Classes, Graphics,
 5   Controls, Forms, Dialogs, StdCtrls, ExtCtrls,
 6   TapFactoryU, BathTapU, KitchenTapU;

 7 type
 8   TfrmTapWherehouse = class(TForm)
 9     rgpTapRange: TRadioGroup;
10     lblKitchenTap: TLabel;
11     lblBathTap: TLabel;
12     procedure rgpTapRangeClick(Sender: TObject);
13   private
14     TapFactory: TAbsTapFactory;
15     KitchenTap: TAbsKitchenTap;
16     BathTap: TAbsBathTap;
17   end; // end TfrmTapWherehouse = class(TForm)

18 var
19   frmTapWherehouse: TfrmTapWherehouse;

20 implementation

21 {$R *.dfm}

22 procedure TfrmTapWherehouse.rgpTapRangeClick(Sender: TObject);
23 begin
24   // select the required factory
25   case rgpTapRange.ItemIndex of
26     0: TapFactory := TFutTapFactory.Create;
27     1: TapFactory := TNouTapFactory.Create;
28     2: TapFactory := TVicTapFactory.Create;
29   end;

30   // make and use the components
31   KitchenTap := TapFactory.GetKitchenTap;      // polymorphic factory
32   lblKitchenTap.Caption := 'Kitchen tap: R' +
33               FloatToStrF (KitchenTap.GetPrice, ffFixed, 8, 2);

34   BathTap := TapFactory.GetBathTap;            // polymorphic factory
35   lblBathTap.Caption := 'Bath tap: R' +
36               FloatToStrF (BathTap.GetPrice, ffFixed, 8, 2);
```

```
37   // free everything
38   FreeAndNil(BathTap);
39   FreeAndNil(KitchenTap);
40   FreeAndNil(TapFactory);
41 end; // end procedure TfrmTapWherehouse.rgpTapRangeClick

42 end. // end unit AbsFactoryDemoU
```

If the user selects the Futurismo range (eg figure 8), line 26 will instantiate a Futurismo tap factory (ie a TFutTapFactory). The code then makes a polymorphic call to the designated tap factory to create the appropriate kitchen tap (line 31) and bath tap (line 34).

With this framework in mind, let's look at the different parts of this pattern.

Ex 13.2 step 2 The product ranges: the Bath Tap

We need a specification for each generic type. In this step we specify the possibilities for the BathTap, and so define the abstract bath tap plus the concrete Futurismo, Nouveau and Futurismo versions.

```
 1 unit BathTapU;

 2 interface

 3 type
 4   TAbsBathTap = class(TObject)      // the abstract, generic bath tap
 5   public
 6     function GetPrice: double; virtual; abstract;
 7   end; // end  TAbsBathTap = class(TObject)

 8   TFutBathTap = class(TAbsBathTap)  // the concrete Futurismo version
 9   public
10     function GetPrice: double; override;
11   end; // end  TFutBathTap = class(TAbsBathTap)

12   TNouBathTap = class(TAbsBathTap)    // the concrete Nouveau version
13   public
14     function GetPrice: double; override;
15   end; // end  TNouBathTap = class(TAbsBathTap)

16   TVicBathTap = class(TAbsBathTap)     // the concrete Victoriana tap
17   public
18     function GetPrice: double; override;
19   end; // end  TVicBathTap = class(TAbsBathTap)

20 implementation

21 { TFutBathTap }
```

```
22 function TFutBathTap.GetPrice: double;          // Futurismo bath tap
23 begin
24   Result := 900.0;
25 end; // end function TFutBathTap.GetPrice: double

26 { TNouBathTap }

27 function TNouBathTap.GetPrice: double;            // Nouveau bath tap
28 begin
29   Result := 350.0;
30 end;  // end function TNouBathTap.GetPrice

31 { TVicBathTap }

32 function TVicBathTap.GetPrice: double;         // Victoriana bath tap
33 begin
34   Result := 835.0;
35 end;  // end function TVicBathTap.GetPrice

36 end. // end unit BathTapU
```

TAbsBathTap, lines 4–7 above, is an abstract class that defines the generic bath tap. Any specific, concrete Bath Tap must comply with this interface and implement it. The bath tap in the Futuristic range is called TFutBathTap and is declared in lines 8–11. It implements the GetPrice method (lines 10, 22–25), overriding the abstract declaration in line 6. In a realistic problem it is probable that the operations will be far more complex than the simple GetPrice shown here. The bath taps for the Nouveau range (TNouBathTap) and the Victoriana range (TVicBathTap) are defined similarly, but with different prices.

Ex 13.2 step 3 The kitchen taps

The definition of the Kitchen taps, in unit KitchenTapU, follows the same format. Create this unit (unit KitchenTapU) by copying unit BathTapU and substituting the names shown in the class diagram (figure 10).

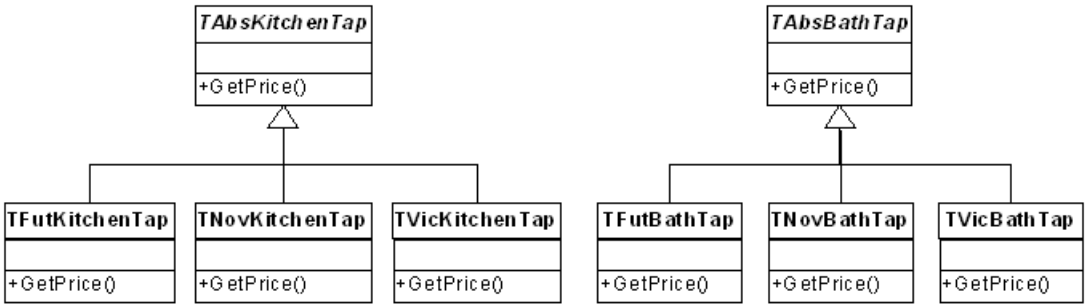

Figure 10 The different product ranges

Ex 13.2 step 4 The Abstract Tap Factory

The remaining code is the Abstract Tap Factory itself:

```
 1 unit TapFactoryU;

 2 interface

 3 uses BathTapU, KitchenTapU;           // the available product ranges

 4 type
 5   TAbsTapFactory = class(TObject)             // the abstract factory
 6   public
 7     function GetBathTap: TAbsBathTap; virtual; abstract;
 8     function GetKitchenTap: TAbsKitchenTap; virtual; abstract;
 9   end;  // end TAbsTapFactory = class(TObject)

10   TFutTapFactory = class(TAbsTapFactory)        // a concrete factory
11   public
12     function GetBathTap: TAbsBathTap; override;
13     function GetKitchenTap: TAbsKitchenTap; override;
14   end;  // end TFutTapFactory = class(TAbsTapFactory)

15   { concrete factories TVicTapFactory & TNouTapFactory are similar }

25 implementation

26 { TFutTapFactory }

27 function TFutTapFactory.GetBathTap: TAbsBathTap;
28 begin
29   Result := TFutBathTap.Create;
30 end; // end function TFutTapFactory.GetBathTap

31 function TFutTapFactory.GetKitchenTap: TAbsKitchenTap;
32 begin
33   Result := TFutKitchenTap.Create;
34 end; // end function TFutTapFactory.GetKitchenTap

35 { concrete factories TVicTapFactory & TNouTapFactory are similar }

53 end. // end unit TapFactoryU
```

One starts with an Abstract Factory declaration (lines 5–9) that lists all the generic products that a factory can create. Here the products are bath taps and kitchen taps and so the Abstract Factory declares GetBathTap and GetKitchenTap as abstract methods (lines 7&8) that return references to each generic product.

One then creates a concrete factory for each available product range. Here we have Futurismo, Nouveau and Victoriana ranges and so we need a concrete factory for each of these (eg lines 10–14 above for Futurismo) with override methods for each product within

the range (lines 12&13). Manufacturing the product in the factory requires creating the specific product within the specific range. So manufacturing a Bath Tap in the Futuristic range involves instantiating a TFutBathTap (line 29), while manufacturing a Bath Tap in the Nouveau range involves instantiating a TNovBathTap, and so on.

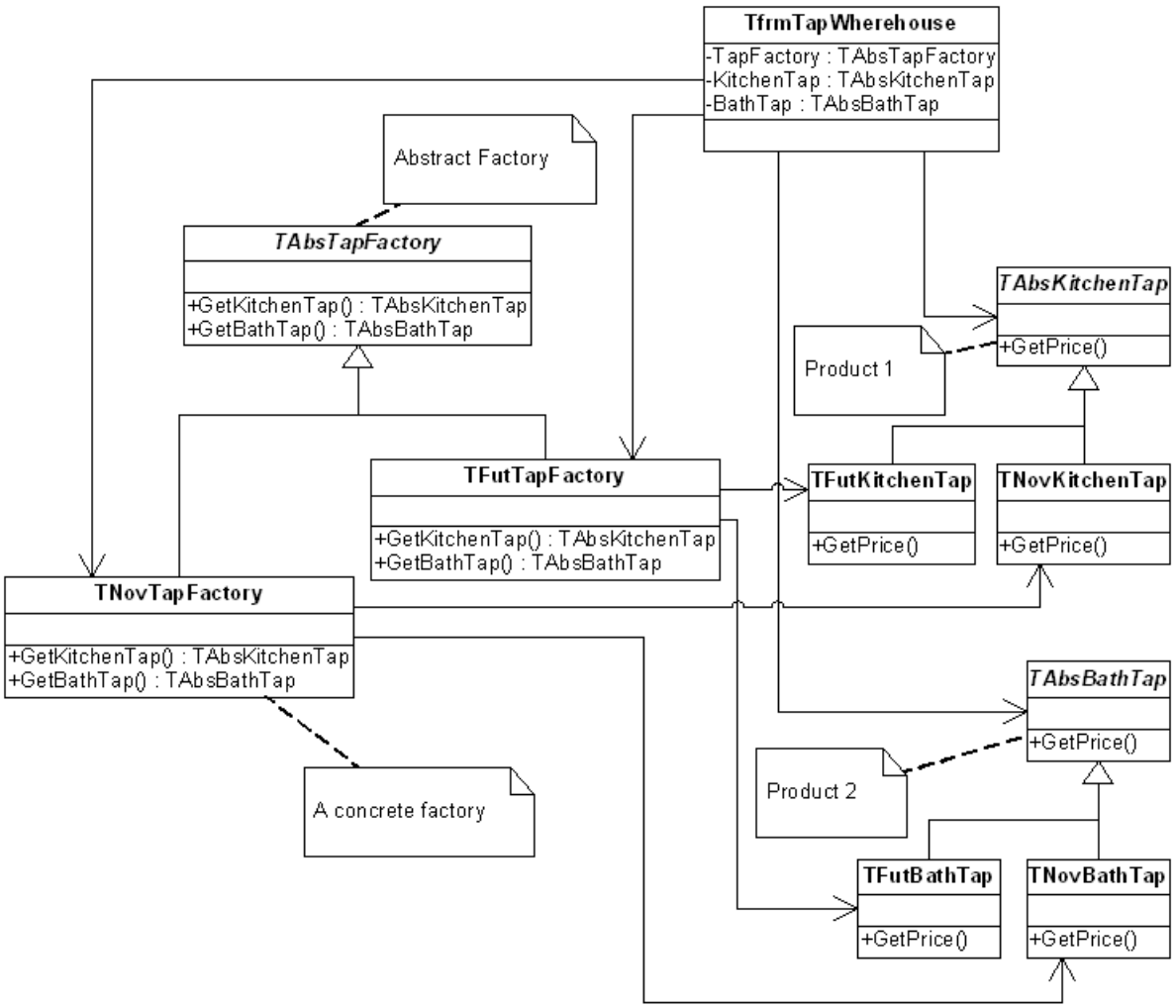

Figure 11 The Abstract Factory Pattern

While the interaction between the various classes is a little complex, each individual method is quite simple. The methods follow standard configurations and so adding product ranges is a relatively mechanistic process. Figure 11 shows the class relationships. (To keep the diagram a little simpler, we show only two products. The links to the Victoriana range repeat the pattern of the Futurismo and Nouveau ranges.) As with the Factory Method there is quite an elaborate network of potential links, and these all appear on the class diagram. However, once one starts instantiating the objects at runtime, the actual links are considerably simpler.

When the user selects one of the ranges (Futurismo, say), the OnClick event handler for the RadioGroup fires (Example 14.2, step 1, lines 22–41). No objects beside the user interface exist yet, but by the end of the Case (line 29), one of the available factories, TFutTapFactory, has been instantiated. The event handler then uses this factory (line 31) to create a KitchenTap, TFutKitchenTap (figure 12).

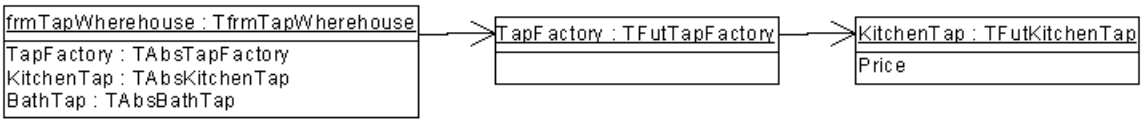

Figure 12 Instantiating a 'Futurismo' kitchen tap

The client (ie the user interface) now uses this KitchenTap (lines 32–33) for whatever operations it requires of it (figure 13).

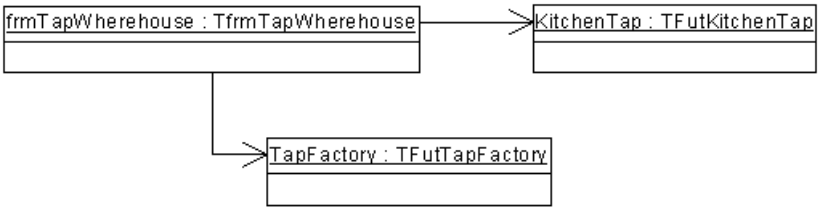

Figure 13 Using the KitchenTap

At its most complex, after all the objects have been instantiated and used (line 36), there are a total of four objects (figure 14) with very simple linking from the client frmTapWhereHouse. So out of a total of thirteen classes available in the system (nine classes if one excludes the factories), the greatest number of objects and of object types that will exist at any one moment is four.

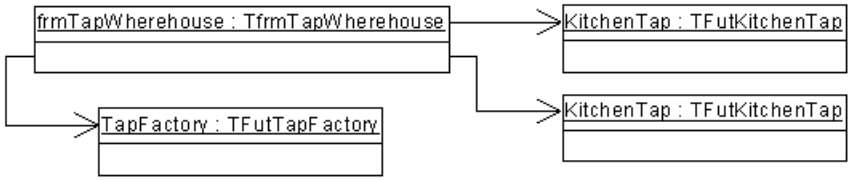

Figure 14 The final instantiations

Notice that except for the user interface object frmTapWherehouse, the identity of the other objects is determined by a single line in the program. If the user selects the Nouveau range instead of the Futurismo range, example 14.2, step 1, line 27 executes instead of line 26.

Consequently, the TNovTapFactory is instantiated instead of TFutTapFactory. This in turn manufactures the Noveau taps instead of the Futurismo taps.

The simplicity that the Abstract Factory Pattern introduces despite the extra classes is what makes this pattern attractive and useful. It also results in high cohesion and low coupling, facilitating future change and enhancement. (If you want to explore this aspect of simplicity, rewrite this application without using either the Abstract Factory or the Factory Methods, ie with all the code in the RadioGroup's OnClick event handler.)

It is worth comparing these class and object diagrams (figures 11–14) with those for the Factory Method (figures 4–6). Figures 6 & 13 show the same result. The difference between the Factory Method and the Abstract Factory patterns is that Abstract Factory uses not just a single Factory Method, but a set of Factory Methods (each with its own product range) that are selected through the Abstract Factory class (cf figures 4 & 11).

The advantages of using the Abstract Factory are therefore very similar to those for using the Factory Method.

Variations on the Factory Method

A number of variations in the Factory Method are possible. In figure 4 the client, TfrmFactoryMethod, is a user interface class. This simplifies the example but is a little unusual for this pattern since the client is often an application independent class.

Also in figure 4, the actual factory class, TFurnCreator, is derived from the abstract class TFurnFactory. TFurnFactory apparently has no role here and can in this case be removed. Typically the actual factory class (eg TFurnCreator) is derived from the client class (eg TfrmFactoryMethod).

Putting these considerations together, and simplifying the product hierarchy (eg TFurniture and its descendants) provides a slightly different class diagram (figure 15) to that suggested by figure 4, and possibly clarifies the derivation of the name 'Factory Method'.

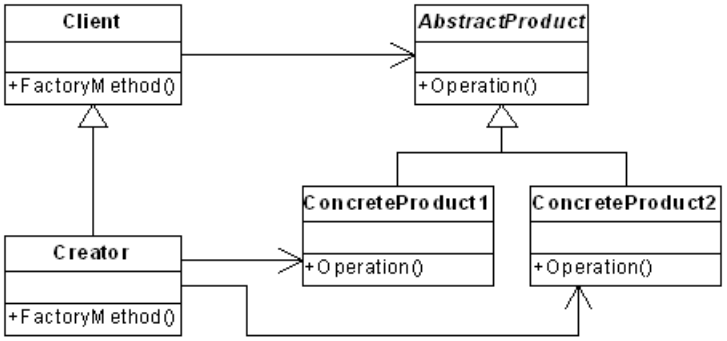

Figure 15 The basic structure of the Factory Method

If Client is designed as application independent but with a virtual FactoryMethod declaration and a reference to an AbstractProduct, Client can be reused in different applications by subclassing an appropriate Creator class and providing the required ConcreteProduct subclasses.

Referring to the Abstract Factory Pattern (figure 11), we can see that GetKitchenTap, for example, is a factory method, with an abstract declaration in TAbsTapFactory and concrete definitions in TNovTapFactory and TFutTapFactory.

Pattern 13.1 The Factory Method Pattern

We can describe the Factory Method Pattern as follows.

Consider a situation where a client object needs to create and use one of a set of related classes and where it is possible to define an abstract class that provides a suitable interface for subclassing all the required classes. For the sake of reduced coupling (to promote subsequent extensions or reuse), the client should not be concerned with the details of the subclass it uses. The client may not be able to anticipate which class it should create and it should only need to specify logical criteria for which subclass is to be created. It should be possible subsequently to add further subclasses that comply with the abstract interface. In situations designed for reuse, the client is typically application independent while the subclasses are application dependent.

Therefore,

delegate the responsibility for instantiating an appropriate subclass that meets the client's needs to an application-specific 'factory' class (either a subclass of the client or a separate class) that produces for the (often application independent) client a suitable application dependent object (or 'concrete product') meeting the generic interface (or 'abstract product'). This allows the client to be independent of specific subclasses and for the subclasses to vary over time. Thus, on the basis of selection data provided by the client, the factory class provides a decision-making class that returns one of several possible subclasses of an abstract base class. The factory thus encapsulate the knowledge of how to select a suitable subclass and keeps the client shielded from the various subclasses that support the particular interface the client expects.

The Factory Method is often used to implement the Abstract Factory.

Pattern 13.2 The Abstract Factory Pattern

We can describe the Abstract Factory Pattern as follows.

Consider a situation where a client object needs to interact with a set of specific object types (ie sets of subclasses complying with specific interfaces) and where it is possible to define a set of related abstract classes that provide suitable interfaces for all the various required subclasses. For the sake of reduced coupling (to promote subsequent extensions or reuse), the client should not be concerned with the details of the subclasses it uses. The client may not be able to anticipate which subclasses it should create and it should only need to specify logical criteria for the subclasses to be created. It should be possible subsequently to add further sets of subclasses that comply with these abstract interfaces. In situations designed for reuse, the client is typically application independent while the subclasses are application dependent.

Therefore,

delegate the responsibility for instantiating an appropriate set of subclasses that meet the client's needs to a subclass of a separate, application-specific 'abstract factory' that declares factory methods to produce suitable application dependent objects (or 'concrete products') meeting the generic interfaces (or 'abstract products'). This allows the client to be independent of the subclasses and for the subclasses to vary over time. On the basis of selection data provided by the client, the factory classes provide decision-making methods that return one of several possible sets of subclasses of a set of abstract base classes. The factories thus keep the client shielded from the various subclasses that support the interfaces the client expects and encapsulate the knowledge of how to select a suitable set of subclasses. Methods, possibly helper methods, that are common to all the concrete factories may be generalised up to the Abstract Factory class.

The Abstract Factory often uses Factory Methods to manufacture the appropriate subclasses.

The Template Method and Strategy patterns are often referred to as kinds of *behavioural* patterns while the Factory patterns are *creational* patterns. In the next chapter we explore another way of introducing alternative behaviour, but this time through a *structural* pattern called Decorator, that allows one to add different behaviour to an existing object in different contexts.

Chapter Summary

– The Factory Method Pattern: create dynamically configurable objects.
– The Abstract Factory Pattern: create dynamically configurable families of objects.
– Polymorphism in OO designs.
– Using parallel linked hierarchies

References

Standard texts such as Gamma *et al*, Grand (1998) and Larman discuss the Factory Method and Abstract Factory patterns. These patterns are also often presented in more introductory books such as Shalloway and Trott.

Gamma, E., Helm, R., Johnson, R. and Vlissides, J. 1995. *Design Patterns: Elements of reusable object-oriented software.* Addison-Wesley, Reading, MA.

Grand, M. 1998. *Patterns in Java, vol 1.* Wiley: New York.

Larman, C. 2001. *Applying UML and patterns: An introduction to object-oriented analysis and design and the Unified Process,* 2nd ed. Prentice Hall: New Jersey.

Shalloway, A. and Trott, J. 2002. *Design Patterns Explained: A new perspective on object-oriented design*. Addison-Wesley.

Problems

Problem 13.1 Study Chapter 13

Identify the appropriate example(s) or section(s) of the chapter to illustrate each comment made in the summary at the end of chapter 13.

Problem 13.2 A language Factory

An important use of Factory Methods and Abstract Factories is to adapt programs to various operating systems (eg Windows or Linux). As a simpler example here, we can use the Factory approach to provide multilingual text. For example, in figure 5 the three captions on the right provide equivalents in different languages depending on which RadioButton is selected. We'll extend this concept in the next problem to adapt a user interface to a user's choice of language.

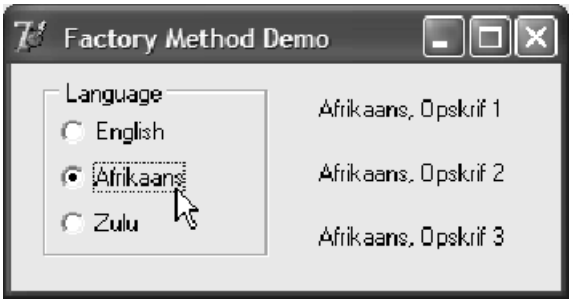

Figure 16 A Factory Method to adapt to different languages

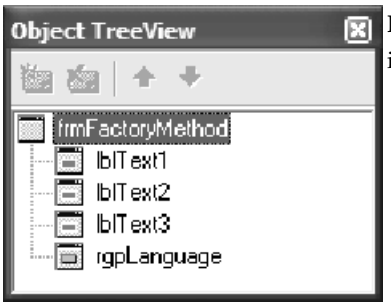

Figure 17 Objects on the user interface

At first glance this looks very much like an adaptation of the Abstract Factory example in chapter 13 of the notes, and this can be done by using an Abstract Factory. But we can do it more simply with a Factory Method approach if, instead of the single GetKind method of example 13.1, each product has several methods.

Write this program to provide equivalent texts in three languages of your choice.

Problem 13.3 Modifying a user interface to a user's language preference

Choose one of the examples that we have used in the notes for this module. Add a way of selecting between three or four different languages, and then use a factory to provide the

necessary captions and text for the user interface in these different languages. We chose example 10.1 to give the user interface in figure 18.

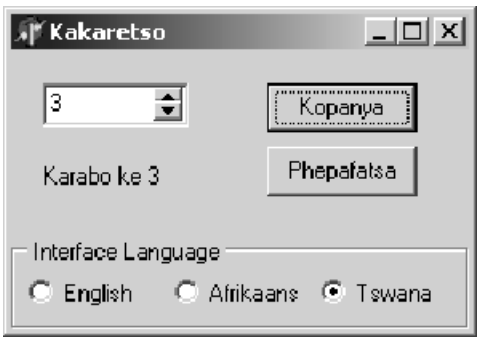

Figure 18 Ex 10.1 modified to provide the user with different interface languages