Chapter 2 Introduction to class inheritance

Main concepts

– Class inheritance.
– Subclassing for reuse: extension, specialisation, generalisation.
– Overriding inherited methods.
– Encapsulation.
– Visual Form Inheritance (VFI).
– Delphi's VCL class hierarchy.
– Navigation (linkage) between objects.
– Navigation on a UML class diagram.
– Visibility for message passing.

Chapter contents

Introduction

In chapter 1 we talked about three perspectives on objects: objects as independent entities, objects as derived entities and objects as interacting entities. In this chapter we will take a first look at objects as derived entities by using one of Delphi's RAD facilities called *Visual Form Inheritance* (VFI). This allows us to define a form and then use it as the basis for further forms through inheritance. We'll use VFI to reuse an existing form through inheritance and then we'll extend and specialise these derived forms through our own code. And because objects in isolation are not much use, we'll look briefly at the interaction between these objects.

Example 2.1 Subclassing through Visual Form Inheritance (VFI)

In this example, we'll use VFI to create the structure in figure 1. For now we show only the class names and suppress the attributes and the methods in the class diagrams. Later diagrams will show more detail.

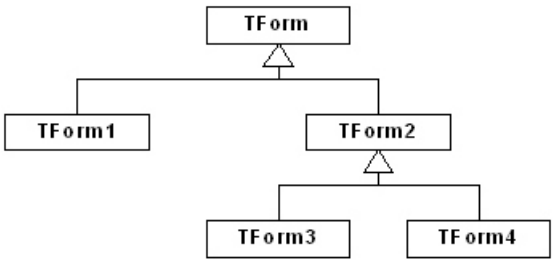

Figure 1 Inheritance through VFI

Ex 2.1 step 1 The master form

Start a new application. We'll write an event handler for this form (TForm1 in figure 1) to display other forms (TForm3 and TForm4) in a moment, but for now, just change the size of this TForm1 to about 300 x 150[1].

Ex 2.1 step 2 The base class

VFI is the RAD process of creating a template form, the base class, and then inheriting (or subclassing) other forms from it. First we create the base class.

Using File | New | Form, add a second form to the project (TForm2 in figure 1) and place a button on it. This will form a base class from which we will inherit two subforms. Make the form a convenient size (about 200 x 100) and create an OnClick event handler for button1 in Unit2:

```
13 implementation

14 {$R *.dfm}

15 procedure TForm2.Button1Click(Sender: TObject);
16 begin
17   ShowMessage ('Button ' + (Sender as TButton).Caption + ' clicked');
18 end; // end procedure TForm2.Button1Click

19 end. // end Unit2
```

The descendant forms (TForm3 and TForm4 in figure 1) will inherit this method, and we'll use the Sender parameter to identify the Caption of the button that has been clicked.

---

[1] These dimensions refer to a low resolution screen. With higher resolution screens 452 x 225 or 600 x 300 may be better. It just needs to be a convenient size: big enough to use easily and small enough not to obscure other forms.

Line 17 is the first appearance of the Sender parameter and we'll see it several times again through the course of this module.

Ex 2.1 step 3 A derived form

TForm2, defined in step 2, is the base form from which we will derive further forms.

To create TForm3 in Delphi 4 to 7, select the menu sequence File | New | Other and then select the Project1 tab (or Project2, 3, ... depending on the name of your current project).To create TForm3 in Delphi 2005, select the menu sequence File | New | Other | Delphi Projects | Inheritable Items. Form1 and Form2 are listed. Select Form2 (this is the base form class from step 2), make sure that the Inherit RadioButton lower down is selected and click OK.

Up until now, all our forms have been derived from TForm. Because we are using TForm2 as a basis, Delphi derives our new form, TForm3 in Unit3, *from TForm2* (line 7 below).

Units are encapsulated from each other, and so for Unit3 to be able to refer to TForm2 in the class declaration (line 7), Delphi inserts Unit2, which defines TForm2, in the global uses clause (line 5) of Unit3.

```
 1 unit Unit3;

 2 interface

 3 uses
 4   Windows, Messages, SysUtils, Variants, Classes, Graphics,
 5   Controls, Forms, Dialogs, Unit2;

 6 type
 7   TForm3 = class(TForm2)  // inherited from TForm2, not TForm
 8   private
 9     { Private declarations }
10   public
11     { Public declarations }
12   end;

13 var
14   Form3: TForm3;

15 implementation

16 {$R *.dfm}

17 end.
```

TForm3 inherits all the data fields (properties) and behaviour (methods and event handlers) from TForm2, and so a button, defined in TForm2, appears as part the screen display of object Form3 (figure 2) even though it is *not* listed in TForm3's type definition (lines 7–12 above). Using the Object Inspector, change the button's Caption to btnForm3 and the form's position (the Top and Left properties)so that it does not obscure either of the other forms. Don't make any other changes to Form3 at this stage.

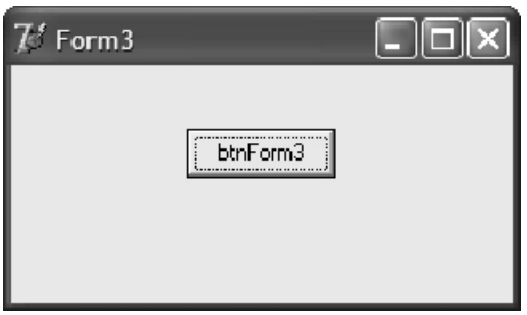

Figure 2 TForm3 as derived from TForm2

Ex 2.1 step 4 Another derived form

We can inherit any number of classes from an existing class, so inherit another new form and unit (TForm4 in Unit4) from Form2 following the process outlines in step 3. Change the Caption of the button on TForm4 to btnForm4. Position the four forms so that they do not overlap, but don't change any other properties.

Inheritance means that the subclass inherits all the data *fields* and methods from the superclass. Because this is a RAD application, the subclasses also have as default the initial data *values* of the superclass. To see this, change, say, the width of Form2. The widths of Form3 & Form4 change as well.

If you're in any doubt about what we have done so far, study the code for each of these units and relate this back to figure 1.

Ex 2.1 step 5 Linking the forms

We now have an inheritance structure with a number of different classes. However, a class structure on its own does not do much – we need to introduce some interaction paths between the classes. We want to be able to display Form3 and Form4 from Form1. We can extend our class diagram to show these paths (figure 3).

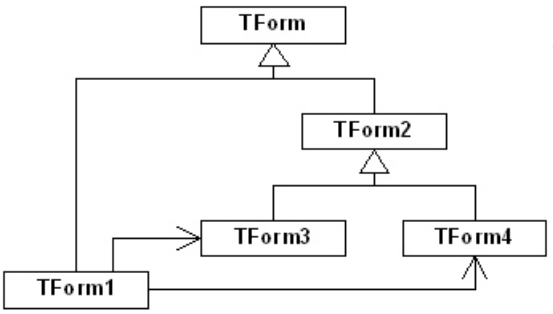

Figure 3 Communication links between classes

The UML convention uses closed, unfilled arrowheads for inheritance and open arrowheads for navigation links (also called association links). Figure 3 shows open arrowheads on the links between TForm1 and TForm3 & TForm4, pointing from TForm1 to the other two forms. These reflect that Form1 can access Form3 and Form4, but that no access is available in the reverse direction.

To implement these links, return to Unit1. Using the Component / Tool palette, place a button on TForm1. Set its Name property to btnShowForms and its Caption to Show Forms. Create an OnClick event handler for it that sends messages to Form3 and Form4, asking them to Show themselves (lines 20–21 below).

Press <F9>, ostensibly to run the program. Delphi will show an Information Box saying: "Form 'Form1' references 'Form3' declared in 'Unit3' which is not in your USES list. Do you wish to add it?" Click the Yes button for this question. Run it again. It repeats the question, but this time for Unit4. Click Yes to add Unit4 to the uses clause as well. Delphi adds a *local* clause 'uses Unit3, Unit4;' to Unit1 (ie in its private, implementation section and not in the public, interface section) (line 16 below).

```
15 implementation

16 uses Unit3, Unit4;

17 {$R *.dfm}

18 procedure TForm1.btnShowFormsClick(Sender: TObject);
19 begin
20   Form3.Show;
21   Form4.Show;
22 end; // end procedure TForm1.btnShowFormsClick
```

Run the program again. Form1 (only) appears. Click on it to invoke its OnClick event handler and Form3 and Form4 appear. Form2 does not appear because it has not been instructed to Show itself. A Click on Form1 can send a Show message to Form3 and to Form4 because there is an association between TForm1 and TForm3 and between TForm1

and TForm4 (figure 3). This linkage is possible because of the uses clause in Unit1, which lists Unit3 and Unit4 (line 16 above).

Click the button on Form3 or Form4, and a message identifying the button by its Caption appears (figure 4).

Figure 4 A confirmation that btnForm3 has been clicked

Neither TForm3 nor TForm4 have defined any data fields or operations. However they are both derived from TForm2, which defines a TButton data field and its OnClick event handler, and so both TForm3 and TForm4 inherit the button and its event handler from TForm2. Thus, through inheritance, TForm3 and TForm4 reuse TForm2 and in this case do not need any code of their own.

We set different Captions for the Buttons on Form3 and Form4 to show that we are working with *different* objects. To identify these separate objects, we use the Sender parameter in line 17 of Form2's OnClick event handler (example 2.1, step 2). Since every instance has its own data, Form3 and Form4 also have their own values for Top, Left, and so on, which is why they can take up different positions on the screen.

Save this project as we'll use it as a starting point for examples 2.2 and 2.3.

Example 2.1 Summary: The main OO principles in this example are *inheritance* and *communication*. TForm2 inherits the knowledge of 'how to be a form' from TForm. It adds its own data field (a button) and a method (the button's OnClick event handler). TForm3 and TForm4 are both derived from TForm2 and neither adds anything of its own. Yet, because of inheritance, each one has the knowledge of how to be a form (from the VCL class TForm), and a button and its event handler (from TForm2). We also created navigation links (simple association) between one form and another.

From a Delphi perspective, we have seen how with VFI we can define a base form (Form2 in this case) and then derive further forms from it. We gained all the functionality available in the base form without having to redefine forms separately as in chapter 1. The buttons on the derived forms even all invoke the base form's event handler. So VFI is a useful illustration of OO and inheritance and the way subclassing can promote reuse.

Example 2.2 Subclassing for extension

In example 2.1 we didn't add any code to units 3 or 4. Because of inheritance each of their Buttons uses the same event handler, the one inherited from the base form. However, a subclass can also *extend* the superclass's functionality. Subclassing for extension involves giving a subclass further features in addition to those it inherits. In this example we'll give each subclass its own separate method: they will each manipulate Form1's position (figure 5).

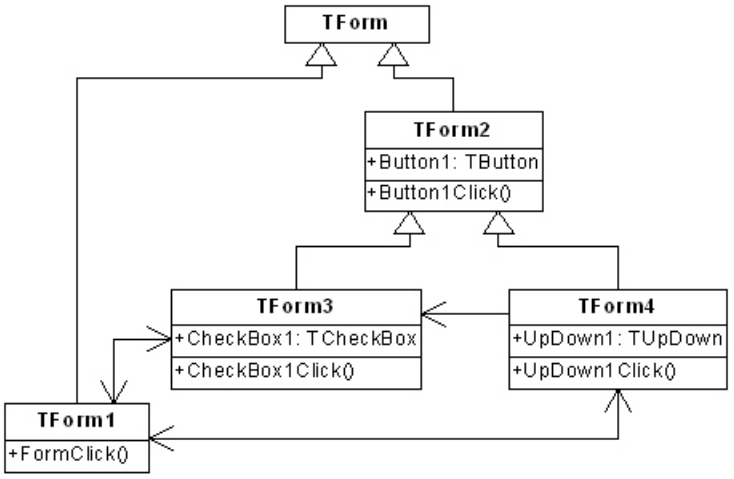

Figure 5 Subclassing for extension and adding bidirectional navigation links

We reflect the bidirectional navigability on our UML class diagram through double openheaded arrows on the association links (figure 5). Here we show the relevant attributes and methods that extend the superclass. To present a slightly different style, the inheritance paths are shown as separate arrows, and not as combined 'tree graphs' as in the previous class diagrams.

Ex 2.2 step 1 Extension in a subclass

Start with the program of example 2.1. Add a TCheckBox to Form3 (figure 6), give it the Caption 'Top', and create an OnClick event handler by double-clicking on the CheckBox:

```
procedure TForm3.CheckBox1Click(Sender: TObject);
begin
  inherited;
  if CheckBox1.Checked then
    Form1.Top := 10;
end;
```

There's a slight difference here from our previous event handlers. Since TForm3 is derived from TForm2 (and not TForm), Delphi automatically inserts the 'inherited' statement as the first program line in this event handler. Leave it there for now even though it does not do anything here – we'll discuss the significance of 'inherited' in the next example.

TForm3 now inherits a button and event handler from TForm2 and adds a CheckBox and another event handler of its own.

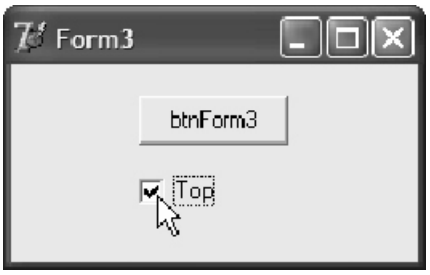

Figure 6 Extending Form3 with a CheckBox component

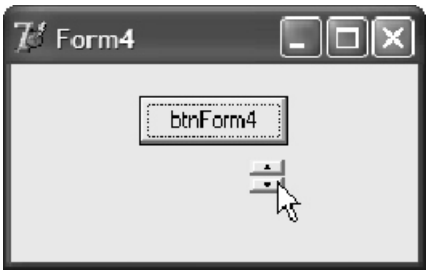

Figure 7 Extending Form4 with an UpDown component

Ex 2.2 step 2 A different extension in another subclass

An important aspect of subclassing for extension is that each subclass can extend the superclass differently. To see this, add an UpDown component (Win32 tab) to Form4 (figure 7), set its Wrap property to True, and create an OnClick event handler:

```
procedure TForm4.UpDown1Click(Sender: TObject; Button: TUDBtnType);
begin
  inherited;
  if Button = btNext then
    Form1.Top := Form1.Top - 20
  else
    Form1.Top := Form1.Top + 20;
  Form3.CheckBox1.Checked := False;
end;
```

Here again, Delphi automatically inserts the 'inherited' statement. We include a message from Form4 to Form3 to clear the CheckBox whenever Form4 repositions Form1.

Ex 2.2 step 3 Bidirectional communication

When using VFI we often send messages from one form to another, and in the steps above both Form3 and Form4 communicate with Form1, and Form4 communicates with Form3. We therefore need links between forms, and Delphi can often set these links up for us. To see this, run the program (press <F9>). Delphi issues a series of messages informing us that various form use other forms but do not include them in their respective uses lists. Click Yes each time for Delphi to insert the appropriate uses clauses. Notice that Delphi inserts *local* uses clauses into Unit3.pas and Unit4.pas (ie in the implementation section).

Attempt again to run the program. This time, when you show Form3 and Form4 by clicking on Form1, each of these has the structure it derives from Form2 and also has its own, individual extension that we have just introduced (figures 6&7).

Example 2.2 Summary: This example introduces *subclassing for extension*. Using inheritance for extension has the characteristics that:
– The subclass inherits all the functionality of the superclass,
– The subclass adds new functionality,
– The subclass makes no change to the (inherited) superclass's functionality.

Subclassing for extension is what people often have in mind when they extol the virtues of inheritance as an important approach to the problem of reuse.

We also used this example as an opportunity to look briefly at bidirectional navigation between objects and at representing navigation on a class diagram.

Example 2.3 Subclassing for specialisation

Another very important use of inheritance is subclassing for *specialisation*. Here, the subclass is in some way a special case of the superclass and either overrides or extends the superclass's methods. We'll give Form3 its own additional Button1Click event handler to extend the operation of the event handler it inherits.

Ex 2.3 step 1 Specialising TForm3

Open the project from example 2.1. (If you don't want to overwrite it, save all the units and project file in a different subdirectory for use in this example).

TForm3 and TForm4 are empty classes that inherit all their functionality (TForm2's OnClick event handler) from TForm2. We are going to *specialise* this OnClick event handler in the subclasses by adding extra functionality to it in TForm3 and in TForm4.

In design mode, select Form3 and double-click on Button1. Delphi creates an event handler skeleton. Add the additional line of code shown below (line 17).

```
 1 unit Unit3;

 2 interface

 3 uses
 4   Windows, Messages, SysUtils, Variants, Classes, Graphics,
 5   Controls, Forms, Dialogs, StdCtrls, Unit2;

 6 type
 7   TForm3 = class(TForm2)
 8     procedure Button1Click(Sender: TObject);
 9   end;

10 var
11   Form3: TForm3;

12 implementation

13 {$R *.dfm}

14 procedure TForm3.Button1Click(Sender: TObject);
15 begin
16   inherited;
17   ShowMessage ('Another msg from ' + (Sender as TButton).Caption);
18 end;

19 end.
```

Run this program. Clicking on the button in Form3 brings up the original message generated by TForm2's Button1Click event handler (example 2.1, step 2, line 17). Accepting this message then brings up another message generated by TForm3's Button1Click event handler. By contrast, clicking on the button in Form4 gives just the functionality derived from Form2.

So with TForm3 we have a case where we use inheritance to add to the inherited functionality, a process called *specialisation*.

Let's look at what is happening here. When you double-click on the Button1 in Form3 during design mode, Delphi creates a new Button1 event handler as a method of *TForm3* (declared in line 8 above). For any object of class TForm3, the OnClick event handler in TForm3 will *override* the inherited event handler in TForm2. (By contrast, TForm4 does not

declare its own event handler, so Form4 therefore goes up a level and executes the event handler inherited from TForm2.)

Delphi also generates the event handler skeleton (lines 14–16 & 18 above). What's different here from the event handler skeletons we saw in chapter 1 is that Delphi inserts an 'inherited' statement (line 16 above and also in the event handlers in example 2.2). Why is this? Usually, with specialisation, we want to add some functionality to that available from the superclass. However, in the previous paragraph we said that the event handler in the derived class completely replaces the event handler defined in the superclass. The inherited keyword in the subclass method calls the corresponding method in the superclass.

So the subclass's method overrides completely the method in the superclass. If we still want the (overridden) superclass's method to run, we insert the inherited keyword, usually as the first statement in the (overriding) subclass's method.

Having called and executed the superclass's method, we now move on to line 17, which provides the specialisation operations.

For interest, you may wish to change the order of lines 16 &17 and then run the program again. The message from TForm3's Button1Click now appears before TForm2's.

Ex 2.3 step 2 Specialising TForm4

We can give TForm4 a different specialisation to TForm3. We will also completely override the superclass's method by deleting the inherited statement that Delphi inserts automatically. Select Form4, double-click on its Button1 and create the following event handler, remembering to remove the 'inherited' statement.

Because this event handler refers to Unit1, this unit also needs a 'uses Unit1' clause. You can insert this yourself (line 17 below) or have Delphi insert it for you as before. (Unit3 does not refer to Unit1, and so we did not insert a 'uses Unit1' clause in step 1 above.)

```
16 implementation

17 uses Unit1;

18 {$R *.dfm}

19 procedure TForm4.Button1Click(Sender: TObject);
20 begin
21   Form1.Top := 300;
22 end;
```

Run this program. Clicking on the button in Form4 repositions Form1 with its top at 300 pixels without displaying any message. The event handler in the superclass does not run

because TForm4's Button1 OnClick event handler (lines 15–18 above) does not contain the 'inherited' keyword and so does not invoke the superclass's (TForm2's) Button1 OnClick event handler.

Example 2.3 Summary: In this example we have briefly explored *inheritance for specialisation*, where a subclass specialises a superclass's functionality. This specialisation can be additional code to that in the superclass's method, in which case the subclass method contains the inherited keyword. The specialisation can also completely replace the superclass's method, in which case the subclass's method does not contain the inherited keyword.

A word of caution: when we completely replace the superclass's method in the subclass we affect the subtyping of the subclass. We have not dealt with subtyping yet. We'll come to it in a little while and then see that interfering with the subtyping should be done only after careful thought since it invalidates polymorphism, a central concept in OO.

Example 2.4 Global and local declarations

Notice that with any RAD generated form, the unit defining the class (form) also declares a global reference to an instance of that class in its interface section. This makes it possible for any unit to access any form declared in another unit provided the appropriate uses clauses are in place. To illustrate encapsulation, we can move the global declaration in Unit4 into the implementation section to see what happens (lines 12–13 below). (The implementation section is accessible only to this unit.)

```
 1 unit Unit4;

 2 interface

 3 uses
 4   Windows, Messages, SysUtils, Variants, Classes, Graphics,
 5   Controls, Forms, Dialogs, StdCtrls, Buttons, ComCtrls, Unit2;

 6 type
 7   TForm4 = class(TForm2)
 8     procedure Button1Click(Sender: TObject);
 9   end;

10 implementation

11 uses Unit1;

12 var
13   Form4: TForm4;
```

```
14 {$R *.dfm}

15 procedure TForm4.Button1Click(Sender: TObject);
16 begin
17   Form1.Top := 300;
18 end;

19 end.
```

When we try to compile it, we get two error messages. Even though Unit1 has Unit4 in its uses clause, Unit1 sees Form4 as an undeclared identifier because Form4 is now declared in Unit4's (private) implementation section and not in its (public) interface section. To emphasise that a declaration in the implementation section is only for a unit's local use, Delphi generates a warning that variable Form4 is declared but not used in Unit4. Restore the variable declaration to its rightful place before the implementation keyword and the program functions correctly again.

Although we showed inheritance for extension and inheritance for specialisation separately in examples 2.3 and 2.4, they often appear in combination. OO programs typically involve quite a sophisticated network of derivation (through inheritance and composition) and communication (through association).

Example 2.5 Hierarchies and generalisation

Both extension and specialisation work from the perspective of an existing class that we reuse to meet a new requirement by extending and/or specialising it in some way. Hierarchical generalisation, on the other hand, looks at what is shared between a number of existing classes (ie what is general between them) and then moves this common code out of the existing classes into a superclass. By generalising these shared characteristics up into a superclass from which the subclasses inherit, these shared characteristics need to be coded only once, in the superclass, and can then be reused by the subclasses. Often the superclass, like TForm2 in the previous examples, is not even instantiated but serves instead simply as a way of storing the characteristics shared by its subclasses.

The mechanisms for both extension and specialisation and for generalisation are much the same and the distinctions between these usually result from a particular set of circumstances. Extension or specialisation often starts with an existing class which is modified in some way to meet a new requirement. Generalisation often starts with the identification of common characteristics between subclasses and results in the creation of a possibly 'artificial' superclass containing these shared characteristics.

Just as, in the previous examples, we can say that TForm3 and TForm4 either extend or specialise TForm2, we can equally say that TForm2 *generalises* TForm3 and TForm4 since

TForm2 embodies the operations and data that are appear in both TForm3 and TForm4 (except where TForm4's Button1Click method overrides TForm2's Button1Click).

So when we analyse a system and find a group of objects it is worth looking out for data and behaviour that different objects share and then generalising this data and behaviour up the hierarchy to gain the benefits of reuse.

Delphi's VCL – an example of subclassing

While the principles we've looked at in the subclassing examples so far have been important, the programs themselves have been trivial. To see a very effective application of subclassing which involves extension and generalisation, and to a less obvious extent, specialisation, we need look no further than Delphi's Visual Component Library (VCL).

The VCL[2] provides the components (via the Component / Tool palette) for building up the user interface. If we look at the available components on the Component / Tool palette, we notice that they have varying degrees of similarity and difference. Comparing a TButton and a TLabel, we see that there are similarities between them: they are both visible on a form at a specific position and so on. But we also see major differences. However, if we compare a TButton and a TBitBtn we see that the similarities are much greater. How do we accommodate varying degrees of similarity and difference effectively?

In principle, we'd like to code whatever is the same between components only once and to use new code exclusively for the *additional* characteristics in the new component.

OO systems make this possible through the class hierarchy and inheritance. Figure 8 shows a *highly* simplified representation of Delphi's VCL class hierarchy.

(A set of standard utility classes like the VCL has an extensive and deep hierarchy. Class hierarchies we define ourselves are usually much shallower.)

[2] Version 6 of Delphi and later also contain the CLX library, introduced for portability with Linux. Version 8 includes the .NET library. Here we'll just stick with the original VCL which is common to all versions.

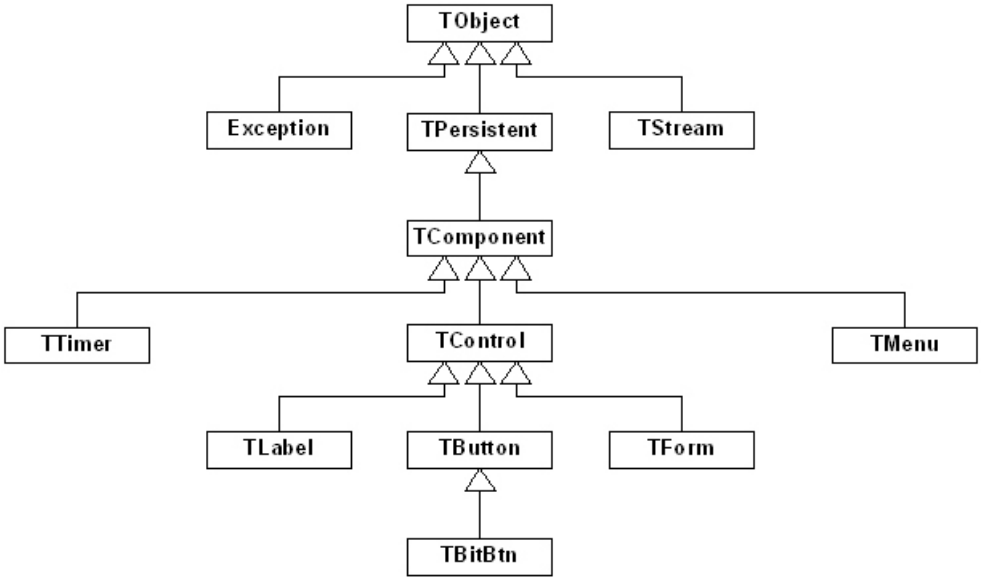

Figure 8 A partial, highly simplified diagram of Delphi's VCL class hierarchy

In a hierarchy like this we look for commonality between the various components and then code this commonality in a (possibly artificial) superclass. Following this approach, we take whatever is common between a TLabel, a TButton and a TForm and code this in a new class, TControl. We now look at what is common between TTimer, TControl and TMenu and code this in a new class, TComponent and so we continue up the hierarchy. We may never instantiate the higher level classes such as TControl, TComponent or TPersistent. Instead they provide the opportunity to code shared characteristics once only in an ancestor. All the descendants then use this code instead of writing it over from scratch each time.

Looking specifically at the VCL, whatever is fundamental to all components and objects appears as high as possible in the hierarchy, in the base class. All Delphi classes must be able to create and destroy instances of themselves and so TObject, which is the root of Delphi's inheritance hierarchy, has these abilities. All Delphi objects except for Exceptions and TStream objects can be saved, and so the next level of the hierarchy separates those that can be saved (TPersistent) from exceptions and TStream objects. Some persistent classes are components and others are not, and so the TComponent type appears below TPersistent in the hierarchy. TComponent introduces the ability to appear on the Component / Tool palette and to be manipulated in the Form Designer, and properties such as Name. It also *inherits* the ability to be saved from TPersistent and to create and destroy objects from TObject via TPersistent.

Some components are visual and others, like TTimer or TMenu, are not. Visual components are visible at run-time and so they need additional abilities such as Left and

Top to give their position on the form, the pop-up hints and the ability to respond to mouse actions. These abilities are packaged in TControl.

Going further down the hierarchy we finally reach the actual components that we use in our programs, such as TLabel and TButton. These inherit all the abilities available in TControl which in turn inherits all the TComponent abilities, which in turn inherits all the TObject abilities.

Common abilities are coded high up the hierarchy, and are available to all the classes lower down that branch of the hierarchy. Each new level in the hierarchy need only code *additional* abilities and does not need to repeat what is available from higher up. So the more general the characteristic, the higher up the hierarchy it is coded. The more specialised characteristic, the lower down it is coded. This characteristic gives rise to phrases like *generalisation hierarchy* and to the term *generalisation* that UML uses when referring to inheritance.

As we saw, TControl introduces properties for Left and Top, and so it and all its descendants have the Left and Top property. Referring to Figure 8, we see that TControl, TLabel, TButton, TForm and TBitBtn all have the Left and Top properties. However, TMenu, TComponent and TObject, which are not descendants of TControl, do not inherit the Left and Top properties.

Help on the VCL hierarchy

The online Help provides a lot of documentation on the VCL Components. To explore this, place a Timer on a form, select the Timer by clicking on it once with the mouse, and press <F1> for Help. A short description appears (figure 9).

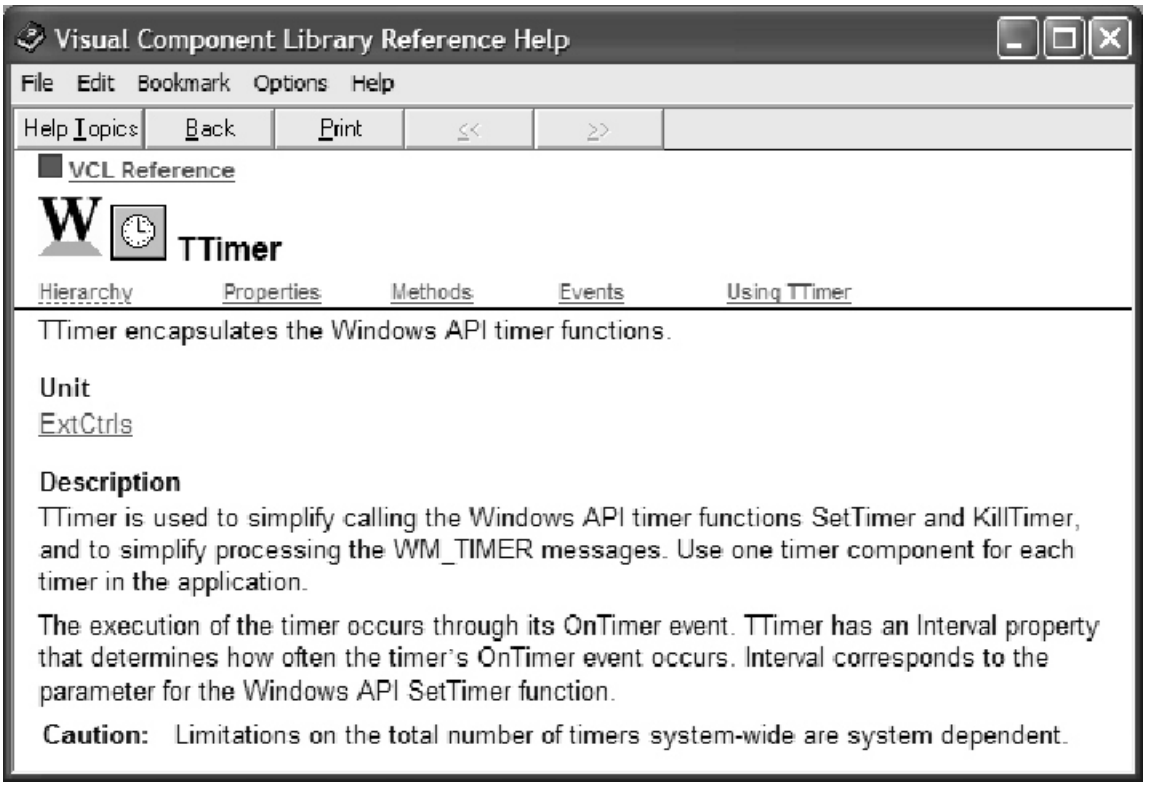

Figure 9 Delphi's VCL Help for the TTimer Component in Delphi 7

Near the top of the display is a list: Hierarchy, Properties, Methods, Events, Using TTimer. Select Hierarchy. The display (figure 10) confirms the hierarchy illustrated in figure 8:

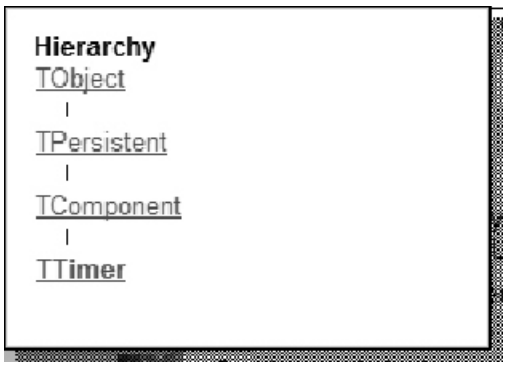

Figure 10 TTimer's position in the VCL hierarchy as shown by Help in Delphi 7

If we select the Properties link, we see a display of the properties that TTimer itself declares and those it inherits from TComponent (figure 11). We can similarly get a display of TTimer's methods (figure 12). This is a much longer list than the properties, and if we scroll down through the list we see methods from each of its ancestors.

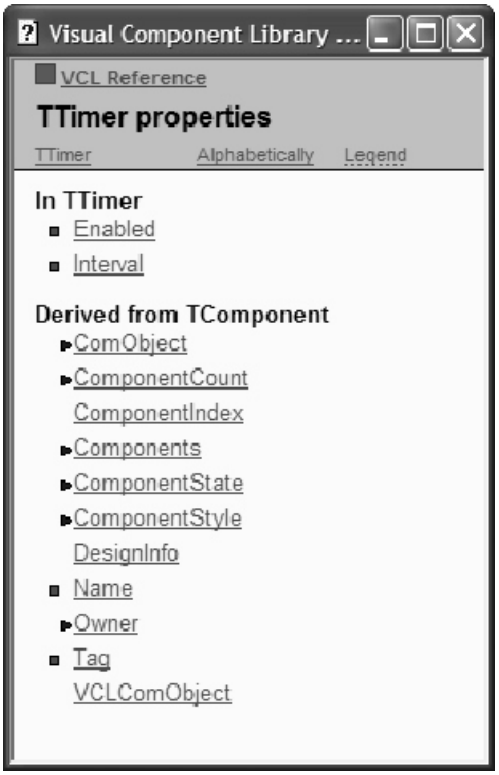

Figure 11 The Help display shows TTimer's local and inherited properties

Figure 12 A partial display of TTimer's methods

Some concluding comments on inheritance

Different types of inheritance

In this chapter we have been looking at inheritance for *subclassing*. There is also another form of inheritance called *subtyping*. Subtyping is linked to the concepts of late (or dynamic) binding and polymorphism. Polymorphism (and subtyping) are important and powerful aspects of object orientation that allow us, for example, to introduce additional subtypes into a system with minimal impact on the rest of the program, so simplifying future changes. We will return later to look at this in more detail.

Multiple inheritance

Delphi, like Java and unlike C++, does not implement multiple inheritance. To achieve an effect similar to multiple inheritance in Delphi or Java, we can use composition and/or interfaces. We will return to this when we look at composition.

Composition – an alternative to inheritance

While inheritance is very powerful and useful, it is not the only the way to re-use existing classes, and in some situations *composition* is preferable. Composition is a way to combine a group of objects to create a 'super-object'. Delphi uses it extensively in generating user interface forms In example 1.2, for instance, the user interface (class TfrmStructureDemo) is derived from the TForm type and contains two Buttons and two methods (the event handlers). We can see this by looking at the type declaration in that program:

```
type
  TfrmStructureDemo = class(TForm)
    btnYellow: TButton;
    btnBlue: TButton;
    procedure btnYellowClick(Sender: TObject);
    procedure btnBlueClick(Sender: TObject);
  end;
```

btnYellow and btnBlue are part of the TfrmStructureDemo class: TfrmStructureDemo HasA btnYellow and btnBlue. Figure 3 of chapter 1 implies composition. As we saw in figure 5 of chapter 1, we can redraw this diagram to emphasise it, using a diamond shape to indicate composition (figure 13 below). We use multiplicity indicators on the link to show that the TfrmStructureDemo class is composed of two TButtons.

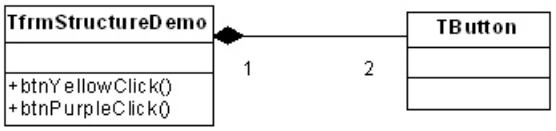

Figure 13 A form composed of two buttons

In summary, building a class from one or more other classes is called *composition*. The aim behind composition is similar to inheritance – to be able to re-use existing objects and classes and so to write code only for additional characteristics. In OO, composition and inheritance are two major tools for creating new classes and for reuse.

We'll investigate composition in chapters to come.

Summary of class inheritance (subclassing)

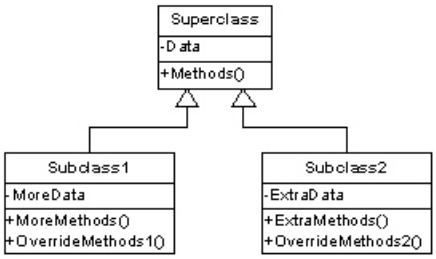

Figure 14 Class inheritance

Characteristics of class inheritance

Subclassing facilitates reuse through programming by difference since the subclass IsA superclass. All the superclass's data fields and methods are available to its subclasses (except for data and methods in the superclass that have private scope). Since the subclass inherits the superclass's functionality it needs to implement (only) the additional data fields and methods and the override methods (figure 14).

Uses of class inheritance

Extension: With extension, the subclass adds new functionality (MoreData, MoreMethods and ExtraData, ExtraMethods in figure 14) but makes no changes to the superclass's existing functionality.

Specialisation: With specialisation, the subclass is a special case of the superclass and so extends or overrides some of the superclass's methods (OverrideMethods1 & OverrideMethods2 in figure 14). Excessive use of specialisation may indicate the need for a redefinition of the hierarchy, particularly where it completely replaces the superclass's behaviour, since this invalidates the subtyping.

Generalisation: For generalisation, create a (possibly artificial) superclass. Remove the *common* data and functionality found in the subclasses and implement these in the superclass (*generalise up*). Do this repeatedly between each set of levels in the hierarchy. The consequence is that the common functionality, ie the generalisations common to the subclasses, is found in the superclasses. The specific functionality, ie the extension and/or specialisation, is found in individual subclasses The classical example of a generalisation hierarchy is the biological taxonomy (eg a cat IsA mammal IsAn animal ...).

Although we have treated these three different types of subclassing separately, they are often combined in practice. It's quite possible and common for a single subclass both to extend a superclass and to specialise an aspect of the superclass, and for the superclasses to have been identified through a process of generalisation.

Chapter summary

Main points:

1. Visual Form Inheritance (VFI): RAD generated reuse of user interface classes.
2. Visual Component Library (VCL): Generalisation, inheritance for reuse.
3. Reuse: Inheritance for extension, specialisation (through overriding) and generalisation.

Objects as derived entities: Class inheritance; composition.
Objects as interacting entities: Simple messages, visibility of other units and objects.
UML diagrams: Inheritance and association in Class diagrams.

Problems

Some of the problems in chapter 3 test principles covered in this chapter.

Problem 2.1 Study Chapter 2

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

Problem 2.2 Using Help

Example 2.2 step 2 includes the code:

```
if Button = btNext then
    Form1.Top := Form1.Top - 20
  else
    Form1.Top := Form1.Top + 20;
```

What is the significance of btNext? How can you find this out using Help? How would you recode this If statement so that the Then part instead adds 20 and the Else part subtracts 20 to give the same functionality as this?