Chapter 11 Some Patterns for varying behaviour

Main concepts

– The Template Method Pattern: varying the separate steps of a fixed algorithm.
– The Strategy Pattern: varying an entire algorithm.
– The Player–Role pattern: associating a Player with different Roles.

Chapter contents

Introduction

The structure of a class is fixed when the class is compiled. However, there are occasions when an operation may need to take on different characteristics at run time as circumstances vary. In other words, it might be necessary that the method call `MyObject.DoSomething` behaves differently at different times.

This chapter discusses techniques that make variations in a method's behaviour possible at run time, even after compilation, either through the use of inheritance structures or through delegation. The first technique, the Template Method pattern, breaks a method into several steps and then allows subclasses to override these individual steps as needed. The second technique, the Strategy pattern, delegates the operation to different subclasses within an associated hierarchy depending on what behaviour is required. The third technique, the Player–Role pattern, overlaps in several ways with the Strategy pattern, though its application is more specific. All three these techniques help to produce code that is robust and relatively easy to modify in the face of changing requirements.

Variable steps within a set sequence

Sometimes a class needs a method with a particular sequence of steps but the details of the steps may vary depending on circumstances. Let's say that the local music school needs a simple information kiosk for the summer school it is organising. When someone requests information about the summer school, the program must give a welcome message, details of the venue, and the date for paying the deposit. The summer school will offer tuition for a number of different instruments and the details of the venue will vary from instrument to instrument. The information kiosk must be available before all the details of the summer school are finalised, and so the software must be highly modifiable in case tuition subsequently becomes available for additional instruments.

Based on this set of requirements, our program design must accommodate several factors:

1. It will always perform a known and fixed sequence of operations (display the welcome, the tuition venue and the date for the deposit);
2. Details of some of the operations vary under different circumstances (different venues for different instruments);
3. There is the possibility of adding additional cases (for possible tuition in other instruments); and,
4. We need default operations for cases where no special operation applies.

Example 11.1 A standard polymorphic solution

We can create a separate class for each type of instrument and provide the necessary operations within each class. This allows operations to be customised easily for each instrument, and additional classes can be defined should additional instruments be included. Each of these classes will have a public ShowMessage procedure and a private Message data field, and the information kiosk user interface object will have a unidirectional 1:1 link to each of these classes (figure 1).

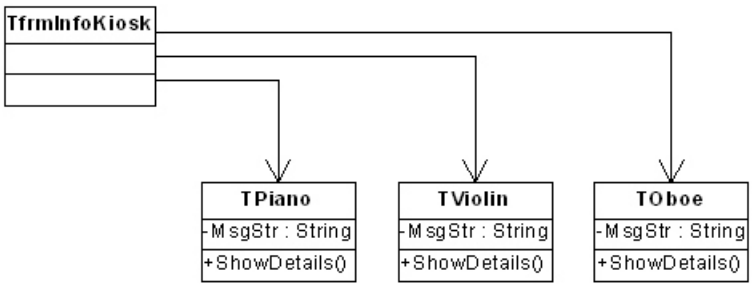

Figure 1 Having a separate, unrelated object for each instrument

Because each instrument has a MsgStr data field and a ShowDetails method, we can introduce some generalisation and, along with it, inheritance and polymorphism. We can make an abstract base class, TInstrument, the general case. It declares MsgStr and ShowDetails and so defines a standard interface for all the instruments. The subclasses inherit this interface, and override ShowDetails as needed (figure 2). Since the subclasses are all true subtypes of TInstrument, any subclass can substitute for TInstrument, allowing TfrmInfoKiosk to link to any object in the hierarchy through the TInstrument interface. (Note that MsgString is a data field declared in *TInstrument*. Through inheritance, each subclass has this data field to use as it wishes and so the subclasses do not declare the data field again, unlike the methods, which are declared in each subclass. If this seems a bit confusing, the code below should help to clarify it.)

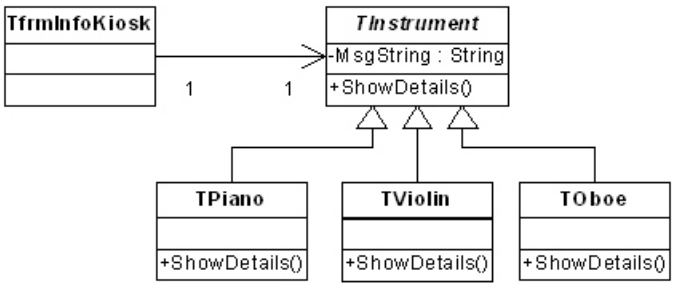

Figure 2 Converting to a generalised, polymorphic hierarchy

Ex 11.1 step 1 Defining the instruments

Since these are very simple classes, we'll put all the instruments in the same unit.

```
1 unit InstrumentsU;

2 interface

3 type
4   TInstrument = class(TObject)
5   protected
6     MsgStr: string;
7   public
8     procedure ShowDetails; virtual; abstract;
9   end; // end TInstrument = class(TObject)

10  TPiano = class(TInstrument)
11  public
12    procedure ShowDetails; override;
13  end; // end TPiano = class(TInstrument)

14  TViolin = class(TInstrument)
```

```
15   public
16     procedure ShowDetails; override;
17   end; // end TViolin = class(TInstrument)

18   TOboe = class(TInstrument)
19   public
20     procedure ShowDetails; override;
21   end; // end TOboe = class(TInstrument)

22 implementation

23 uses Dialogs;  // for ShowMessage

24 { TPiano }

25 procedure TPiano.ShowDetails;
26 begin
27   MsgStr := 'Welcome to the Music Summer School.';
28   MsgStr := MsgStr+#13#10+ 'For the piano, meet at the Great Hall.';
29   MsgStr := MsgStr + #13#10 + 'Submit your deposit by 30/11/05.';
30   ShowMessage (MsgStr);
31 end; // end procedure TPiano.ShowDetails

32 { TViolin }

33 procedure TViolin.ShowDetails;
34 begin
35   MsgStr := 'Welcome to the Music Summer School.';
36   MsgStr := MsgStr+#13#10+ 'For the violin, meet at the Strad Room.';
37   MsgStr := MsgStr + #13#10 + 'Submit your deposit by 30/11/05.';
38   ShowMessage (MsgStr);
39 end; // end procedure TViolin.ShowDetails

40 { TOboe }

41 procedure TOboe.ShowDetails;
42 begin
43   MsgStr := 'Welcome to the Music Summer School.';
44   MsgStr := MsgStr+#13#10+ 'For the oboe, meet at the Denner Room.';
45   MsgStr := MsgStr + #13#10 + 'Submit your deposit by 30/11/05.';
46   ShowMessage (MsgStr);
47 end; // end procedure TOboe.ShowDetails

48 end. // end InstrumentsU
```

Ex 11.1 step 2 Creating the client

The information kiosk has a very simple user interface (figure 3).

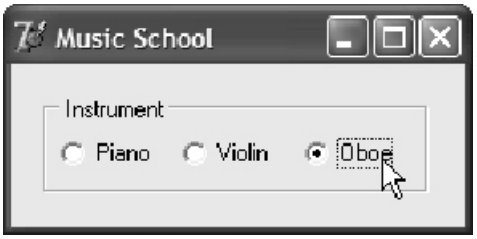

Figure 3 The music school's information kiosk

Figure 4 The user interface objects

A brief, customised message comes up when an instrument is selected (figure 5).

Figure 5 The welcome, venue and deposit information display

```
1 unit SummerSchoolU;

2 interface

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

7 type
8   TfrmInfoKiosk = class(TForm)
9     rgpInstrument: TRadioGroup;
10    procedure rgpInstrumentClick(Sender: TObject);
11  private
12    Instrument: TInstrument;
13  end; // end TfrmMusicSchool = class(TForm)

14 var
15  frmInfoKiosk: TfrmInfoKiosk;

16 implementation

17 {$R *.dfm}
```

```
18 procedure TfrmInfoKiosk.rgpInstrumentClick(Sender: TObject);
19 begin
20   case rgpInstrument.ItemIndex of
21     0: Instrument := TPiano.Create;                  // substitutions
22     1: Instrument := TViolin.Create;
23     2: Instrument := TOboe.Create;
24   end;
25   if Instrument = nil then
26     raise Exception.Create('Invalid Instrument object')
27   else
28     Instrument.ShowDetails;                       // a polymorphic call
29   FreeAndNil (Instrument);
30 end; // end procedure TfrmMusicSchool.rgpInstrumentClick

31 end. // end SummerSchoolU
```

It is possible that with future changes, we add more options to the RadioGroup but forget to add these cases to the Case statement. The error detection of lines 25–26 will pick up this error.

Ex 11.1 step 3 Assessing the program

This program works, but there is room for improvement. The main problem is all the repetition – except for one line, the instruments' ShowDetails procedures are identical (unit InstrumentsU, lines 28, 36 & 44). This means a lot of repeated typing and the possibility of errors, especially if further instruments are added. The program also does not enforce the presence of each step (welcome, venue and deposit) and the programmer could easily skip one of the steps, or not change all of the subclasses should some detail change, such as the date for the deposit.

In a situation like this with procedural programming we would create a single ShowDetails procedure. This in turn would call various subroutines and so avoid unnecessary duplication and enforce some consistency. The Template method pattern allows us to do something similar but within the object oriented style.

The Template Method

At the start of this chapter we listed four factors operating in this application. We repeat them with comments that show the development of the Template Method pattern. The requirements are:

1. The sequence of steps is fixed. To enforce this, we'll define a TemplateMethod in the base class that specifies the required sequence of operations and that each subclass will

inherit. (See TemplateClass and the note in figure 6.) The TemplateStep methods may be abstract or concrete (discussed further in point 4 below).

2. Details of the steps may vary. To allow this, the subclasses will provide different implementations for those operations that vary by overriding the operations declared in the base class. (See TemplateImplementation1 and TemplateImplementation2 in figure 6.)
3. New variations of the steps may be needed. Any new cases will be accommodated in additional subclasses containing the necessary override operations (Template-Implementation3, …. introduced in future as needed).
4. Default steps. Default steps are implemented as virtual concrete TemplateSteps in the base class. Steps that do not need defaults are declared as virtual abstract TemplateSteps in the base class.

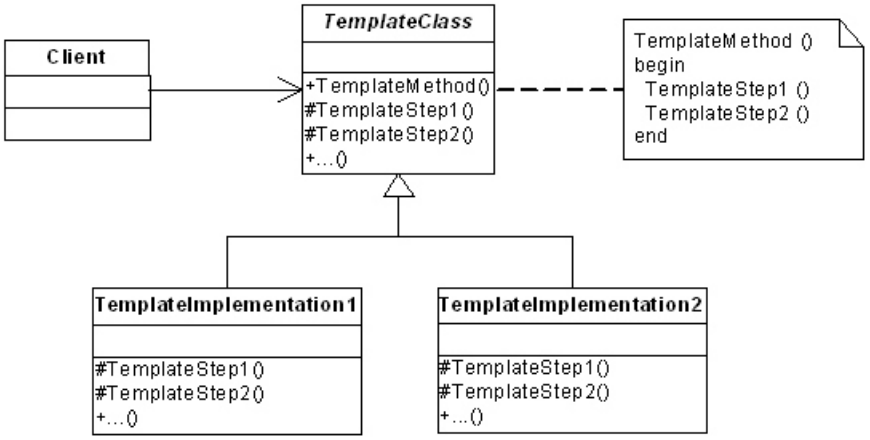

Figure 6 The subtypes inherit TemplateMethod but provide their own TemplateSteps

Example 11.2 The Template Method pattern

To implement the Template Method pattern as shown in figure 6 for this application, we'll define a ShowDetails method in the base class only (TInstrument) and not in any of the subclasses. We'll convert the ShowDetails procedure into a Template Method that invokes a series of other methods, one for each of the steps, and the subclasses will inherit this ShowDetails method. We implement any steps that are common to all (or several) cases in the ancestor class. We implement those steps that vary in the descendant classes. Through polymorphism, although all descendants will invoke the inherited ShowDetails method, this method will in turn invoke the class's own methods for the steps.

Ex 11.2 step 1 Converting to a Template Method

```
1 unit InstrumentsU;

2 interface

3 type
4   TInstrument = class(TObject)
5   protected
6     MsgStr: string;
7     procedure AddWelcome; virtual;          // TemplateStep1: default
8     procedure AddVenue; virtual; abstract; // TemplateStep2: variable
9     procedure AddDeposit; virtual;          // TemplateStep3: default
10  public
11    procedure ShowDetails;  // Template Method, base only
12  end; // end TInstrument = class(TObject)

13  TPiano = class(TInstrument)
14  protected
15    procedure AddVenue; override;           // TemplateStep2 override
16  end; // end TPiano = class(TInstrument)

17  TViolin = class(TInstrument)
18  protected
19    procedure AddVenue; override;           // TemplateStep2 override
20  end; // end TViolin = class(TInstrument)

21  TOboe = class(TInstrument)
22  protected
23    procedure AddVenue; override;           // TemplateStep2 override
24  end; // end TOboe = class(TInstrument)

25 implementation

26 uses Dialogs;                                    // for ShowMessage

27 { TInstrument }

28 procedure TInstrument.ShowDetails;
29 begin
30   // primitives in the template
31   AddWelcome;
32   AddVenue;
33   AddDeposit;
34   ShowMessage (MsgStr);
35 end; // end procedure TInstrument.ShowDetails

36 procedure TInstrument.AddDeposit;
37 begin
38   MsgStr := MsgStr + #13 + 'Submit your deposit by 20/12/06.';
39 end; // end procedure TInstrument.AddDeposit
```

```
40 procedure TInstrument.AddWelcome;
41 begin
42   MsgStr := 'Welcome to the Music Summer School.';
43 end;  // end procedure TInstrument.AddWelcome

44 { TPiano }

45 procedure TPiano.AddVenue;              // override step in base class
46 begin
47   MsgStr := MsgStr + #13 + 'For the piano, meet at the Great Hall.';
48 end;  // end procedure TPiano.AddVenue

49 { TViolin }

50 procedure TViolin.AddVenue;             // override step in base class
51 begin
52   MsgStr := MsgStr + #13 + 'For the violin, meet at the Strad Room.';
53 end;  // end procedure TViolin.AddVenue

54 { TOboe }

55 procedure TOboe.AddVenue;               // override step in base class
56 begin
57   MsgStr := MsgStr + #13 + 'For the oboe, meet at the Denner Room.';
58 end;  // end procedure TOboe.AddVenue

59 end.  // end InstrumentsU
```

Although we have redefined TInstrument in comparison to example 11.1 step 1, it still has the same public interface (procedure ShowDetails declared in line 11). However, while all the descendants override ShowDetails in the first version (example 11.1, step 1, lines 12, 16 & 20), none of the descendants override it in the revised version. The descendants therefore all inherit ShowDetails from their ancestor and so none of the descendants declare a public interface (example 11.2, step 2, lines 13–24).

However, we have completely restructured ShowDetails. It is now a Template Method that lists the necessary steps but does not give any of the implementation details (lines 28–35 above). The operations that previously were part of ShowDetails are now relegated to three step methods, AddWelcome, AddVenue and AddDeposit (lines 31–33 above, and compare with example 11.1, step 2, lines 25–47). AddWelcome and AddDeposit are default steps common to all, and so they are defined in the superclass (lines 7, 9, 36–43 above). AddVenue varies between classes and so this is made abstract in the base class (line 8) and overridden in the subclasses (lines 45–58).

What effect does this have? If the instance is, say, a TOboe, it *inherits* the ShowDetails method from TInstrument. This first invokes the AddWelcome method (line 31). TOboe does not have an AddWelcome method and so it invokes its parent's AddWelcome method (lines 40–43). Next, ShowDetails invokes the AddVenue method. TOboe has its own

AddVenue method, and so, because of the dynamic binding, it invokes its own AddVenue (lines 55–58) to provide the special variation for this case. ShowDetails now invokes the AddDeposit method. TOboe does not have an AddDeposit method, and so it invokes the default AddDeposit method it inherits from its parent (lines 36–39).

In this example, only the AddVenue template step varies. It is the only step declared as abstract (line 8 above) and so it is the only specialisation that is expected in the subclasses.

Template Steps are typically declared as virtual in the superclass (lines 7–10) and as override in the subclasses (lines 15, 19, 23). If a fixed or default operation is needed for a step (as is the case here), the step method in the superclass is concrete (lines 7, 9). If a fixed or default operation is not needed for a step, the step method in the superclass is abstract (line 8 above) and is implemented only in the subclasses. We can summarise this by saying that operations which *must* be overridden in the subclasses are declared as abstract in the superclass. Those that may or may not be overridden are made concrete. Template Steps are typically all made virtual in the base class, even if not being currently overridden, so that they may be overridden if needed during future enhancement.

Run and test this new version of InstrumentU using the user interface class of example 11.1 step 2. It performs as before, and the client is not aware that a Template Method is being used. The client, frmInfoKiosk, simply issues the method call `Instrument.ShowDetails` (example 11.1 step 2 line 30 or example 11.2 step 2 line 36). *The behaviour of this ShowDetails method varies under different circumstances*. In example 11.1 this variation in behaviour resulted from conventional polymorphism and necessitated a lot of duplicated code. In example 11.2 the variation comes from the Template Method pattern with minimal duplication.

Between example 11.1 and 11.2 we were able to restructure the TInstrument hierarchy quite extensively without the user interface object being aware of this, and the Template Method pattern can often be retrofitted to a program with minimal disruption to the remainder of the system.

Ex 11.2 step 2 Avoiding repeated object creation and destruction

Besides the Template Method pattern, this example can also illustrate an important aspect about object creation and destruction. In example 11.1 step 2 we create and destroy a TInstrument object each time we display the information. Doing this repeatedly can cause memory fragmentation. So some programmers might prefer to create a single object of each type once, at program start up, and then simply switch the reference as needed.

```
 1 unit SummerSchoolU;

 2 interface

 3 uses
 4   { as before }

 7 type
 8   TfrmInfoKiosk = class(TForm)
 9     { as before }
12   private
13     Piano: TPiano;
14     Violin: TViolin;
15     Oboe: TOboe;
16   end; // end TfrmMusicSchool = class(TForm)

17 var
18   frmInfoKiosk: TfrmInfoKiosk;

19 implementation

20 {$R *.dfm}

21 procedure TfrmInfoKiosk.rgpInstrumentClick(Sender: TObject);
22 var
23   Instrument: TInstrument;
24 begin
25   case rgpInstrument.ItemIndex of
26     0: Instrument := Piano;     // switch reference instead of Create
27     1: Instrument := Violin;
28     2: Instrument := Oboe;
29   else Instrument := nil; // for error detection
30   end;
31   if Instrument = nil then
32     raise Exception.Create ('Invalid Instrument object')
33   else
34     Instrument.ShowDetails;                    // Template Method call
35 end; // end procedure TfrmMusicSchool.rgpInstrumentClick

36 procedure TfrmInfoKiosk.FormCreate(Sender: TObject);
37 begin
38   Piano := TPiano.Create;
39   Violin := TViolin.Create;
40   Oboe := TOboe.Create;
41 end; // end procedure TfrmInfoKiosk.FormCreate

42 end. // end SummerSchoolU
```

Instead of declaring a private TInstrument as part of TfrmInfoKiosk, we now declare a TPiano, a TViolin and a TOboe (lines 13–15). When we create a TfrmInfoKiosk we also instantiate these three objects (lines 38–40). We declare a TInstrument as a local variable in the event handler (lines 22–23) and then assign this to refer to one of the existing objects in the Case statement (lines 26–28). Local references are not initialised to nil and so the Case

statement must include an else statement to nil the reference if the RadioGroup's ItemIndex is beyond the range recognised by the event handler (line 29). We don't FreeAndNil this reference as we did previously.

This approach is fractionally faster than that in example 11.1 step 2 since it does not create and free objects repeatedly. More importantly, it does not introduce the possibility of memory fragmentation that extensive use may cause in example 11.1 step 2. This approach, however, requires slightly more memory storage because of the additional data fields in TfrmInfoKiosk. Since we only ever instantiate TfrmInfoKiosk once, this additional memory requirement is not significant. Therefore the approach of example 11.2 step 2 is probably preferable to that of example 11.1 step 2 in this situation. Other situations may have different forces operating.

We have previously seen similar approaches to object creation and destruction in example 6.3 step 2 and example 6.4 step 2.

Frameworks and reuse with the Template Method pattern

The example above is, of course, highly simplified, and a problem as small as this may not warrant the use of the Template Method pattern. However for learning purposes keeping the problem simple makes it is easier to see the underlying principles of the pattern and these principles are what you should concentrate on: a Template Method is defined in a base class with calls to Template Step methods that can be overridden in the subclasses.

A widespread use of the Template Method pattern is for reuse and for frameworks. The Template Method specifies a possibly very complex set of generic business rules or algorithm steps in a base class without necessarily defining the implementation details. This Template Method and its set of business rules can be reused in different applications by deriving application specific subclasses from the base class. These subclasses specify the new implementation details by selectively overriding the required steps. They need to specify only those steps that vary, and so the entire set of business rules does not need to be recoded.

Thus a typical application of the Template Method arises in the context of reuse, where the same generic algorithm is required in different applications even though details of some of the steps may vary.

Pattern 11.1 The Template Method Pattern

We can state the Template Method pattern as follows:

In some situations we may have a generic algorithm to solve a particular problem. Some parts of the algorithm may be fixed but at least some parts vary according to context. It is important to ensure that each step is implemented. It may be necessary to provide default steps that can be overridden in particular contexts. Fixed parts of the algorithm should be defined once only.

Therefore,

to allow a subclass to redefine particular steps in an algorithm without changing the algorithm's structure:
– define a separate primitive method for each step of the algorithm;
– define a template method that invokes each separate primitive method;
– implement the template method and the fixed or default primitive methods as concrete methods in the superclass;
– declare the varying methods that have no default implementation as abstract methods in the superclass;
– declare any methods which may be varied as virtual methods in the superclass; and
– override the varying methods in the subclasses.

Figure 6 illustrates the Template Method pattern.

Example 11.3 The Strategy Pattern

The Template Method pattern allows us to vary the steps of an algorithm depending on different contexts. The next pattern we are going to look at, the Strategy pattern, allows us to vary the entire algorithm in response to the context. The Template Method is based on an *inheritance hierarchy*, with clever use of polymorphism (figure 6). The Strategy is based on *composition*, with the composed objects being polymorphic.

Our example to illustrate the Strategy pattern is to calculate a Salary based on whether the person is a Trainee, is Weekly paid or is Monthly paid. There are three notches within each category. The user interface of the program is in figure 7.

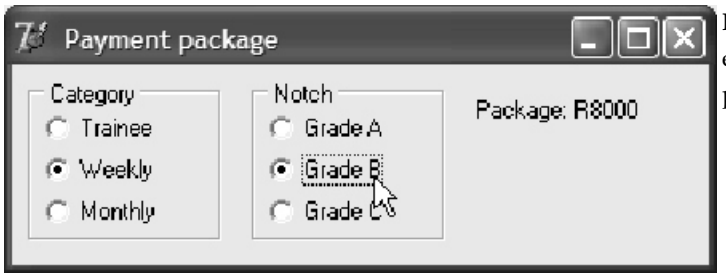

Figure 7 Calculating an employee's monthly payment package

Figure 8 The objects on the user interface

Ex 11.3 step 1 The user interface

The user interface has the following listing:

```
 1 unit BenefitsU;

 2 interface

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

 7 type
 8   TfrmPackage = class(TForm)
 9     rgpCategory: TRadioGroup;
10     rgpNotch: TRadioGroup;
11     lblPackage: TLabel;
12     procedure ChangeContext(Sender: TObject);
13     procedure FormCreate(Sender: TObject);
14     procedure FormDestroy(Sender: TObject);
15   private
16     Package: TPackage;  // business logic object
17   end; // end TfrmPackage = class(TForm)

18 var
19   frmPackage: TfrmPackage;
```

```
20 implementation

21 {$R *.dfm}

22 procedure TfrmPackage.ChangeContext(Sender: TObject);
23 var Value: integer;
24 begin
25   // Perform the calculation
26   Value := Package.Calculate (rgpCategory.ItemIndex,
27                     rgpNotch.ItemIndex);
28   // Display the results of the calculation
29   lblPackage.Caption := 'Package: R' + IntToStr (Value);
30 end; // end procedure TfrmPackage.btnCalculateClick

31 procedure TfrmPackage.FormCreate(Sender: TObject);
32 begin
33   Package := TPackage.Create;
34 end; // end procedure TfrmPackage.FormCreate

35 procedure TfrmPackage.FormDestroy(Sender: TObject);
36 begin
37   Package.Free;
38 end; // end procedure TfrmPackage.FormDestroy

39 end. // end unit BenefitsU
```

The OnClick events of both rgpCategory and rgpNotch trigger the ChangeContext event (lines 22–30 above). There are several ways to set up this triggering, and you can use whichever method suits you. One way is to start by placing just the components on the form and setting the necessary properties in the Object Inspector. Next, type in the event handler declaration (line 12 above, under the published section of the type definition as shown). With the cursor on line 12, press <Ctrl+Shift+C>. This invokes class completion and will generate a skeleton for the ChangeContext method (lines 22, 24 & 30). Add the body of the method (lines 23, 25–29). To link rgpCategory's OnClick event to this handler, select rgpCategory on the form. Select the Events tab in the Object Inspector and single click alongside OnClick. Click on the down pointing arrow to open the drop down box and select ChangeContext. Repeat this for rgpNotch. Now create the form's OnCreate and OnDestroy event handlers in the usual way.

The client, here the user interface TfrmPackage, simply stipulates a particular behaviour, Package.Calculate, along with a particular context, the parameters rgpCategory.ItemIndex and rgpNotch.ItemIndex, (lines 26–27) and then uses the method's return value. How the Package object performs the calculation is of no concern to the client.

We'll consider two ways of performing this calculation. The first, in step 2, uses a single TPackage class that uses conditional statements to calculate the Package based on the category and the notch (figure 9).

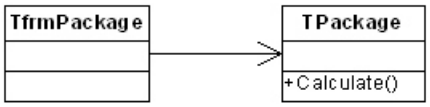

Figure 9 Separating the calculation into a separate class

The second approach, in step 3, uses the Strategy pattern as an alternative. The interface between TfrmPackage and TPackage will be unchanged, but TPackage will use several strategy classes instead of a single monolithic method to calculate the salary package.

Ex 11.3 step 2 Varied behaviour through conditional statements

To calculate the package using a single method, we'll create a simple object, TPackage, as follows:

```
 1 unit PackageU;

 2 interface

 3 type
 4   TPackage = class(TObject)
 5   public
 6     function Calculate (ACategory, ANotch: integer): integer;
 7   end; // end TPackage = class(TObject)

 8 implementation

 9 { TPackage }

10 function TPackage.Calculate (ACategory, ANotch: integer): integer;
11 { Parameters set the context for the required calculation }
12 begin
13   case ACategory of
14     0: Result := 5000;                                    // trainee
15     1: case ANotch of                                      // weekly
16          0: Result := 6000;
17          1: Result := 8000;
18          2: Result := 9000;
19        else Result := 0;
20        end; // end case ANotch of
21     2: Result := 5000 + ANotch * 2500;                    // monthly
22   else Result := 0;
23   end; // end case ACategory of
24 end; // end procedure TPackage.Calculate

25 end. // end PackageU
```

Based on the two parameters, the Case statements calculate the value of the package. Using Case statement in this way is quite manageable in a simple example like this, but if the

application were more complex the series of conditionals would become more difficult to understand, to test thoroughly and to modify. It would be helpful if the calculation could be partitioned logically, and this is what the Strategy pattern does.

Ex 11.3 step 3 Alternative forms of the algorithm

The Strategy pattern provides a separate subclass for each variation on the algorithm, and so we start by taking each case of the outer Case statement (step 2, lines 13–23) and placing it into a separate subclass. Case 0, line 14, is encapsulated in the TTrainee class; case1, lines 15–20, is encapsulated in the TWeekly class, and case 2, line 21, is encapsulated in the TMonthly class. These subclasses are derived from an abstract base class, TCategory, that defines the common interface for all the subclasses (figure 10).

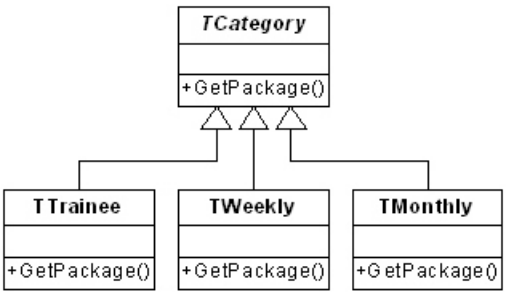

Figure 10 The Strategy hierarchy with three available strategies

We code this as follows:

```
1 unit CategoryU;

2 interface

3 type
4   TCategory = class(TObject)                  // the abstract strategy
5   public
6     function GetPackage (ANotch: integer): integer;
7                                          virtual; abstract;
8   end; // end TCategory = class(TObject)

9   TTrainee = class(TCategory)                   // a concrete strategy
10  public
11    function GetPackage (ANotch: integer): integer; override;
12  end; // end TTrainee = class(TCategory)

13  TWeekly = class(TCategory)                    // a concrete strategy
14  public
15    function GetPackage (ANotch: integer): integer; override;
16  end; // end TWeekly = class(TCategory)

17  TMonthly = class(TCategory)                   // a concrete strategy
```

```
18   public
19     function GetPackage (ANotch: integer): integer; override;
20   end; // end TMonthly = class(TCategory)

21 implementation

22 { TTrainee }

23 function TTrainee.GetPackage(ANotch: integer): integer;
24 begin
25   Result := 5000;
26 end;  // end function TTrainee.GetPackage

27 { TWeekly }

28 function TWeekly.GetPackage(ANotch: integer): integer;
29 begin
30   case ANotch of
31     0: Result := 6000;
32     1: Result := 8000;
33     2: Result := 9000;
34   else Result := 0;
35   end;
36 end; // end function TWeekly.GetPackage

37 { TMonthly }

38 function TMonthly.GetPackage(ANotch: integer): integer;
39 begin
40   Result := 5000 + ANotch * 2500;
41 end; // end function TMonthly.GetPackage

42 end. // end CategoryU
```

At the base of this hierarchy is the abstract TCategory class. This defines the interface, the GetPackage method, to this set of strategies (lines 4–8). The subclasses are all pure subtypes of their ancestor and override the GetPackage method to provide the different implementations. These implementations can be varied easily and independently of one another since they are packaged separately in individual classes.

None of these classes provides any data storage. Instead, each provides a different implementation of a standard operation that accepts an input parameter and returns a value. The classes are polymorphic, and so interchangeable. Consequently, the details of the transformation of the input data to output data depends on which of the subclasses receives the method call. As we'll see in the next step, we can now modify TPackage to select the appropriate subclass in response to the message it receives form the user interface. In OO-speak, we create a context class that, on the basis of the parameters sent by the client class, delegates the operation to the appropriate worker class.

Ex 11.3 step 4 The context class

We now modify TPackage's Calculate method (step 2) to use the strategies defined in step 3 (figure 11). So that the interface with the user interface remains unchanged, TPackage must maintain the same method signature for Calculate as before. Thus the new version of the Calculate method must encapsulate the interaction with the TCategory hierarchy.

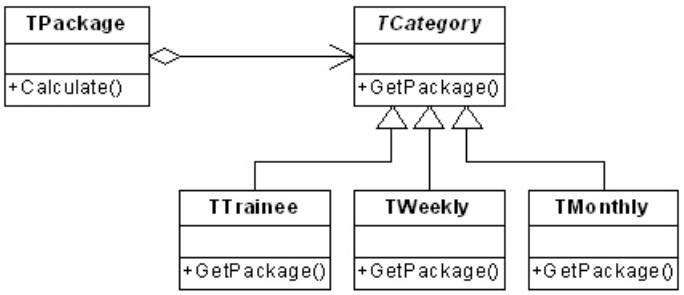

Figure 11 The Context class, TPackage, interacting with the strategies

```
1  unit PackageU;

2  interface

3  uses
4    CategoryU;

5  type
6    TPackage = class(TObject)
7    private
8      Trainee: TTrainee;                                  // the strategies
9      Weekly: TWeekly;
10     Monthly: TMonthly;
11   public
12     function Calculate (ACategory, ANotch: integer): integer;
13     constructor Create;
14     destructor Destroy; override;
15   end; // end TPackage = class(TObject)

16 implementation

17 { TPackage }

18 function TPackage.Calculate (ACategory, ANotch: integer): integer;
19 // Incoming parameters set the context for the required calculation
20 var
21   CategoryObj: TCategory;          // Reference to associated strategy
22 begin
23   // Select and use the associated strategy
24   case ACategory of
25     0: CategoryObj := Trainee;
26     1: CategoryObj := Weekly;
27     2: CategoryObj := Monthly;
28   else CategoryObj := nil;
29   end;
```

```
30   if assigned (CategoryObj) then              // simple error checking
31     Result := CategoryObj.GetPackage(ANotch)  // polymorphic strategy
32   else
33     Result := 0;
34 end; // end procedure TPackage.Calculate

35 constructor TPackage.Create;
36 begin
37   inherited;
38   Trainee := TTrainee.Create;                    // propagate creation
39   Weekly := TWeekly.Create;
40   Monthly := TMonthly.Create;
41 end; // end constructor TPackage.Create

42 destructor TPackage.Destroy;
43 begin
44   Trainee.Free;                                 // propagate destruction
45   Weekly.Free;
46   Monthly.Free;
47   inherited;
48 end; // end destructor TPackage.Destroy

49 end. // end PackageU
```

Calculate now uses the incoming parameters, sometimes called the 'context', to select which of the strategies (lines 24–27) to delegate to perform the required operation (line 31). It returns this value to its client. TPackage, the Context class, retains the same public interface, the single Calculate method (lines 12–14 & figure 12). Thus TfrmPackage, the client, is unaware that the strategies even exist.

TPackage creates three objects, one for each of the strategies (lines 8–10, 35–41). The Calculate method declares a local object reference, CategoryObj, of type TCategory (lines 20–21). TCategory is an abstract class. We can declare a reference to it but we cannot instantiate it. However, CategoryObj can refer to any concrete descendant of TCategory. So, depending on the calculation context set by the value of the ACategory parameter, the Calculate method assigns CategoryObj to one of the existing TTrainee, TWeekly or TMonthly strategy objects (lines 24–27). Then, through delegation, it uses CategoryObj polymorphically to calculate the value for the appropriate Package in accordance with the notch (line 31). Through this delegation, TPackage gives the appearance of varying the operation of its Calculate method in response to run time conditions.

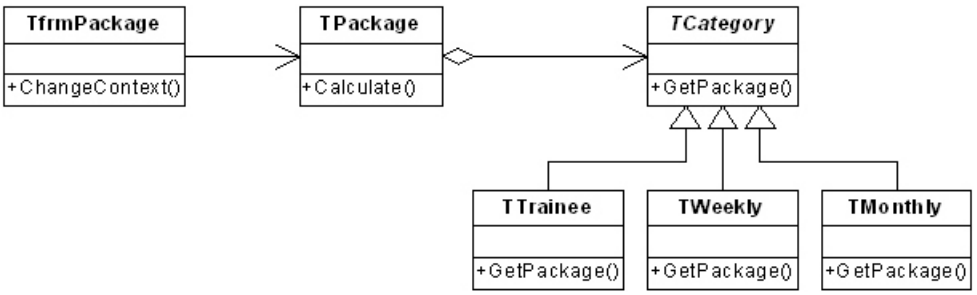

Figure 12 Applying the Strategy pattern

Is the fuss worth it?

Using the Strategy pattern results in additional classes and programming. Is it worth the effort? Probably not for such a small program as we have here. But as the size of the program increases, different factors become prominent.

One reason for using the Strategy is to enhance the program's cohesion and coupling, and consequently its modifiability. So we take some extra care now to simplify our job in future. The Strategy partitions a complex class with conditional statements into a set of clearly defined and structured subclasses. This encapsulates the different algorithms and provides good visibility into the structure of the problem. This also provides well-defined access points for possible future changes, and limits the effects of any change – one can alter a strategy subclass without a detailed understanding of the surrounding system.

Another possible reason to use the Strategy pattern is to provide order to an existing set of classes. The Strategy pattern allows an object to change its behaviour dynamically by reassigning its associated classes. Thus if there are a number of closely related classes with slight differences in behaviour, a Strategy may provide a way to refactor these and to combine them into a single hierarchy.

While a client must be aware of what variations are available and must establish the context accordingly, the Strategy hides all the details of the algorithms from the client and so supports the separation of concerns.

Potential drawbacks of the Strategy are that it increases the number of classes in the system, possibly making the overall system more difficult to understand at first, and that certain of the strategies do not require all the data in the standard method signature (as in the Trainee strategy, which ignores the value of the ANotch parameter – example 11.3 step 3 lines 23–26).

So if we have a small, contained problem, or if the system is guaranteed not to change, or if it does not need to change its behaviour dynamically, it is probably not worth the trouble of introducing the Strategy pattern. If these conditions do not hold, and the problem can be successfully packaged into separate strategies, the extra initial effort of the Strategy pattern is probably a good investment.

Pattern 11.2 The Strategy Pattern

We can state the Strategy pattern as follows:

Suppose a program must offer a number of variations of a particular algorithm or behaviour. These variations can be specified separately in classes that have identical or nearly identical interfaces. The class using these different variations does not need to interact with the implementation of the variations, but may need to select different variations under different run time conditions.

Therefore,

Separate the selection of the algorithm or behaviour from its implementation, allowing the selection to be determined at run time:

1. Set up an inheritance hierarchy (figure 13). The base class of this hierarchy, *AbstractStrategy*, defines the standard interface that each ConcreteStrategy implements. It is usually abstract. (In figure 12 TCategory is the *AbstractStrategy*.)
2. Each required ConcreteStrategy is a pure subtype in this hierarchy and implements the abstract methods declared in the base class (figure 13). (In figure 12, TTrainee, TWeekly and TMonthly are ConcreteStrategies derived from TCategory.)
3. The Context class that uses the Strategy has a private reference to the base of the Strategy hierarchy. (In figure 12, TPackage is the Context class.)
4. The Context class assigns its reference to the required subtype to allow for the required polymorphic substitution based on the context set by the Client. (In figure 12, TfrmPackage is the Client class.)
5. The Client interacts only with the Context and is unaware of the Strategies that are delegated to do the actual work.

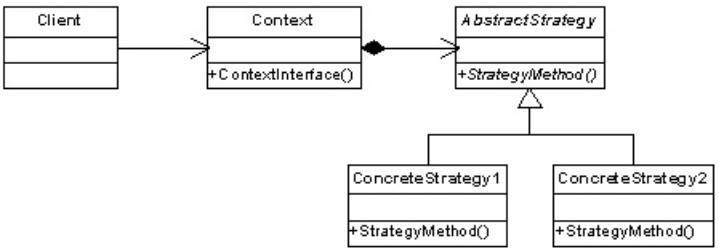

Figure 13 The Strategy Pattern

The Player–Role Pattern

Patterns are intended to help us understand a problem and its possible solutions more clearly. So different patterns, associated with different contexts or with different ways of looking at different situations, may provide solutions similar to each other.

The Player–Role pattern (although it is not always identified as a pattern) has many similarities to the Strategy pattern, and we'll apply it here briefly to a similar problem to that we used for the Strategy pattern.

Let's assume that we need to keep a record of employees. They have a Name, a Category (Monthly, Weekly, Trainee), a Notch (Grade A, B or C) and a remuneration Package based on their Category and Notch. There are about a thousand employees, but for this example we'll work with just a single one. The test user interface appears in figure 14.

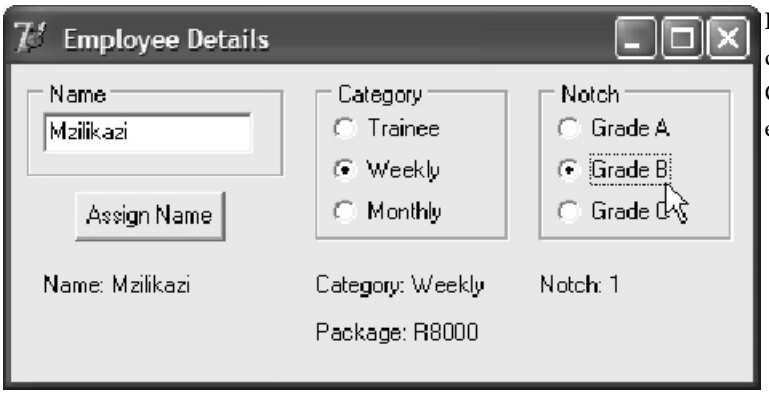

Figure 14 A test interface to check that we record the Name, Category and Notch of an employee correctly

Figure 15 Objects on the test user interface

How do we model an employee in an OO system? A common approach is to create a TEmployee class to carry the data common to all employees (such as Name) and then to subclass it to handle the differences between the Monthly, Weekly and Trainee categories (figure 16).

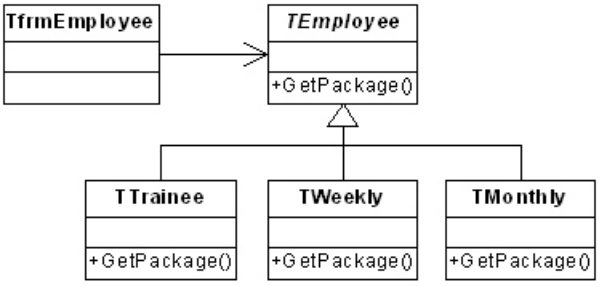

Figure 16 Subclassing the different kinds of employee

TEmployee declares an abstract GetPackage method, and each of its subclasses implements the method appropriately. This can work but there are at least two issues here for concern. First, if too many more types of employee are added to this system, the number of subclasses could become excessive. For example, maybe all employees can be on either a temporary or a permanent contract (figure 17). Or we might need to add hourly paid workers, or keep track of permanent and non-permanent residents for tax purposes. The number of subclasses could rapidly become very difficult to work with effectively.

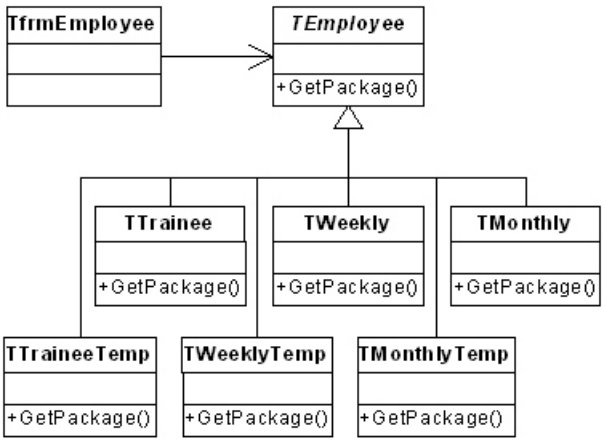

Figure 17 Subclassing the additional employees

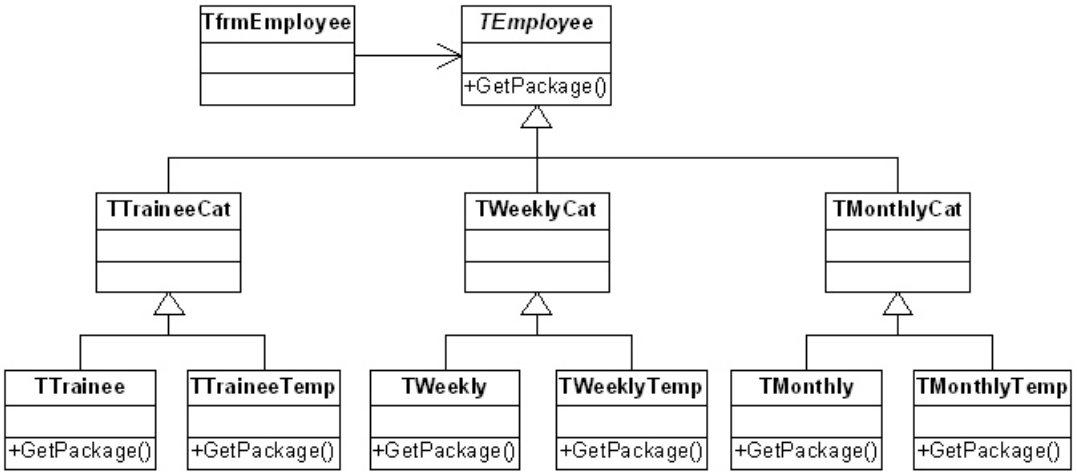

Figure 18 Subclassing for greater structure

One way to deal with an explosion of subclasses is to introduce an additional layer into the hierarchy. We could, for example, group all permanent staff in one subhierarchy and temporary staff in another. Or we could group Trainees together, with Weekly staff and Monthly staff each in their own subhierarchy (figure 18). Whichever way we do it, though, it becomes unwieldy and there is going to be duplication of code. For example, if we take the approach of figure 18, we find that there are permanent staff in each subhierarchy and temporary staff in each subhierarchy. We don't have multiple inheritance, so there is no way in which a class can inherit from both a Trainee superclass and a Permanent superclass. Using subclassing, our only alternative is to duplicate code, and that introduces a lot of opportunities for error. And in figure 18 we have not yet made allowance for different taxation rules. (Notice that we are elaborating here on anti-pattern 7.3.)

A second problem with this approach is that these subclasses are actually about *roles*, not about classes, and so our modelling is not semantically correct. For example, we can easily

imagine that a company may hire someone as a temporary trainee for a certain probation period. After completing the probation period successfully, the person may remain as a trainee but get a permanent contract. Once the training period has elapsed, the person would then be granted a permanent contract as either a weekly or a monthly employee.

If we use the subclassing approach, as a player (here, an employee) moves from role to role we have to create a series of new subclass objects to describe each new role, copy the relevant data to each new subclass and then destroy the previous objects for the previous roles that no longer apply.

So our problem is how to develop a model that allows us to distinguish between the *player*, here the Employee, and the possible *roles* a player may assume (here Trainee, Weekly or Monthly), and then to associate a player with one or more roles. As we saw in chapter 7, subclassing is not an effective way of modelling roles. Instead, the Player–Role pattern suggests, we can create separate hierarchies representing the players and the roles independently of each other. We then establish a dynamic association between a player and the role currently being played. The association changes as the player assumes different roles. Figure 14 of chapter 7 illustrates this.

Applying this pattern to this problem brings us to a similar design to the Strategy pattern (figure 12), but we have come to it through considering the roles involved rather than how to select between different algorithms (figure 19, which extends figures 17 & 18 to allow for different Tax rules too). By encapsulating these differences into separate, decoupled role classes, we leave the Employee class free to model the *player* and then to associate with separate *role* classes (the equivalent of the strategy classes in figure 12). The employee class is now more streamlined and cohesive, and its possible roles are clearly identified.

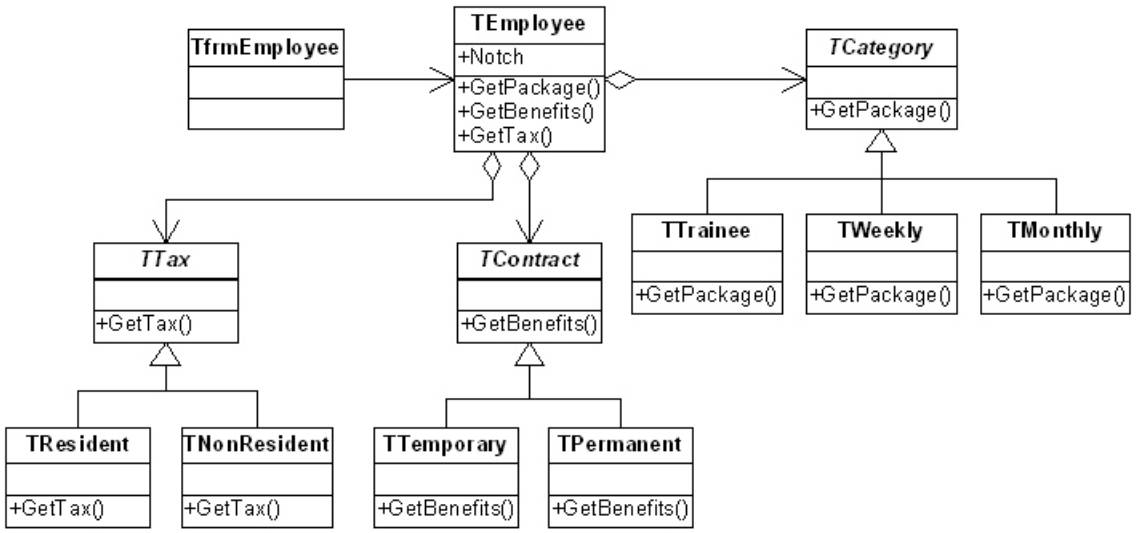

Figure 19 Modelling three different roles that the player TEmployee undertakes

TEmployee carries references to a TCategory subclass, a TContract subclass and a TTax subclass. Should the employee change status and become a permanent resident, TEmployee switches its association from a TNonResident to a TResident, and so on. Notch affects the calculations of both the TWeekly and TMonthly strategies and is not an independent role of its own. We therefore model it as a data field in TEmployee and not as a set of classes.

In the final example of this chapter, we'll model an employee with the TCategory role. TContract and TTax can be added in a similar way if needed.

Example 11.4 A Player–Role example

We've chosen this example so that we can reuse code from example 11.3 and so that we can look at a number of differences between these two applications. The user interface has several differences; the TPackage class is replaced by a TEmployee class. There are some overlaps between the two, and the TCategory class has a minor change to highlight an interesting use of polymorphism and RTTI.

Ex 11.4 step 1 The user interface

The user interface is shown in figures 14 & 15. The matching code, which you can adapt from example 11.3 or start anew, is:

```
 1 unit BenefitsU;

 2 interface

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

 7 type
 8   TfrmEmployee = class(TForm)
 9     { standard RAD declarations }

23   private
24     Employee: TEmployee;                    // business logic class
25   end; // end TfrmEmployee = class(TForm)

26 var
27   frmEmployee: TfrmEmployee;

28 implementation

29 {$R *.dfm}
```

```
30 procedure TfrmEmployee.btnNameClick(Sender: TObject);
31 begin
32   Employee.Name := edtName.Text;
33   lblName.Caption := 'Name: ' + Employee.Name;
34 end; // end procedure TfrmEmployee.btnNameClick

35 procedure TfrmEmployee.rgpNotchClick(Sender: TObject);
36 begin
37   Employee.Notch := rgpNotch.ItemIndex;
38   lblNotch.Caption := 'Notch: ' + IntToStr (Employee.Notch);
39   lblPackage.Caption := 'Package: R'+ IntToStr(Employee.GetPackage);
40 end; // end procedure TfrmEmployee.rgpNotchClick

41 procedure TfrmEmployee.rgpCategoryClick(Sender: TObject);
42 begin
43   Employee.AssignRole(rgpCategory.ItemIndex);
44   lblCategory.Caption := 'Category: ' + Employee.GetRole;
45   lblPackage.Caption := 'Package: R'+ IntToStr(Employee.GetPackage);
46 end; // end procedure TfrmEmployee.rgpCategoryClick

47 procedure TfrmEmployee.FormCreate(Sender: TObject);
48 begin
49   Employee := TEmployee.Create;
50 end; // end procedure TfrmEmployee.FormCreate

51 procedure TfrmEmployee.FormDestroy(Sender: TObject);
52 begin
53   Employee.Free;
54 end; // end procedure TfrmEmployee.FormDestroy

55 end. // end unit BenefitsU
```

Our TEmployee class will have a Name property, and to test it we set the property in line 32 and read it in line 33. TEmployee also has a Notch property which we set in line 37 and read in line 38. Since the Notch affects the value of Package, we display the new value of Package in line 39.

The main reason for using the Player–Role pattern is because it introduces the possibility of switching a player between different roles. To demonstrate this, TEmployee has an AssignRole method which we call in line 43. Then, in lines 44 & 45, we display the values of the Role and of the resultant Package.

Ex 11.4 step 2 The Player class

The code for TEmployee, which is the Player class, is as follows:

```
1 unit EmployeeU;

2 interface
```

```
3 uses
4   CategoryU;

5 type
6   TEmployee = class(TObject)                              // the Player
7   private
8     FName: string;
9     FNotch: integer;
10    CategoryObj: TCategory;                  // a role; initially nil
11  public
12    property Name: string read FName write FName;       // player data
13    property Notch: integer read FNotch write FNotch;     // attribute
14    procedure AssignRole (ACategory: integer);
15    destructor Destroy; override;
16    function GetPackage: integer;
17    function GetRole: string;
18  end; // end TEmployee = class(TObject)

19 implementation

20 { TEmployee }

21 procedure TEmployee.AssignRole(ACategory: integer);
22 begin
23   CategoryObj.Free;                              // free previous role
24   // Create the new associated role object
25   case ACategory of
26     0: CategoryObj := TTrainee.Create;
27     1: CategoryObj := TWeekly.Create;
28     2: CategoryObj := TMonthly.Create;
29   else CategoryObj := nil;                    // range / error control
30   end;
31 end; // end procedure TEmployee.AssignRole

32 destructor TEmployee.Destroy;
33 begin
34   CategoryObj.Free;                             // propagate destruction
35   inherited;
36 end; // end destructor TEmployee.Destroy

37 function TEmployee.GetPackage: integer;
38 begin
39   if assigned (CategoryObj) then              // simple error checking
40     Result := CategoryObj.GetPackage(Notch) // polymorphic delegation
41   else
42     Result := 0;
43 end; // end procedure TEmployee.Calculate

44 function TEmployee.GetRole: string;
45 begin
46   if assigned (CategoryObj) then              // simple error checking
47     Result := CategoryObj.GetRole           // polymorphic delegation
48   else
49     Result := 'Unassigned';
50 end; // end function TEmployee.GetRole
```

```
51 end. // end EmployeeU
```

Here we treat the role as a constituent class of TEmployee, and so its reference is private (line 10). The Name is intrinsically part of the player, and is not a role, and so it is modelled as a property of TEmployee (line 12). As mentioned earlier, Notch is modelled as an attribute (line 13).

The functionality of the Calculate method of example 11.3 is split between the AssignRole and the GetPackage methods. AssignRole (lines 21–31) is a form of factory that creates the appropriate new role object (CategoryObj) after freeing the existing one, thereby switching the player's role. (We look at factories in chapter 13.)

Two methods, GetPackage and GetRole, use CategoryObj. GetRole delegates the identification of the current role to CategoryObj (line 47) while GetPackage delegates to CategoryObj the calculation based on the value of the Notch property (line 40). (To promote encapsulation, GetPackage uses the Notch property rather than the private data FNotch.)

Finally, here the lifetime of the role is the responsibility of the player, and so we provide a destructor to propagate the destruction of the player (lines 32–36).

Ex 11.4 step 3 The Roles

We reuse the strategy hierarchy from example 11.3 as the roles for this example, but with the addition of the GetRole method.

```
 1 unit CategoryU;

 2 // Defines the roles
 3 // Stores no state

 4 interface

 5 type
 6   TCategory = class(TObject)                    // the abstract role
 7   public
 8     function GetPackage (ANotch: integer): integer;
 9                                             virtual; abstract;
10     function GetRole: string;
11   end; // end TCategory = class(TObject)

12   { TTrainee, TWeekly and TMonthly as in example 11.3 step 3 }

24 implementation

25 { TCategory }
```

```
26 function TCategory.GetRole: string;        // inherited by subclasses
27 begin
28   Result := Self.ClassName;                        // dynamic binding
29   Delete (Result, 1, 1);                       // Remove the T prefix
30 end; // end function TCategory.GetRole

31 { TTrainee, TWeekly and TMonthly as in example 11.3 step 3 }

51 end. // end CategoryU
```

GetRole is an interesting method. It is defined in the superclass, which is abstract, and so is used only by the subclasses, which inherit it. Line 28 uses the Self identifier, which identifies the object from which the method is called. Through dynamic binding, Self identifies the actual subclass and neither the class in which the method is defined nor the type in the type declaration. ClassName uses RTTI to identify the class (as described in example 5.2). Line 29 simply removes the leading T from the class name for the sake of legibility.

Run and test this program and confirm that it works and that you understand how it works!

Pattern 11.3 The Player–Role Pattern

We can state the Player–Role pattern as follows:

At times there may be one or more players in a system that can take on various roles. Because of the copy-delete cycle involved when changing roles, it is not practical to combine the player and the role in the same class. There may also be occasions when a player can take on more than one role simultaneously, and modelling each possible combination of roles as a separate class causes a proliferation of classes with considerable redundancy. Packaging all the roles into a single class would become extremely unwieldy. Taking these factors into account, it would make the system more cohesive if the data and behaviour associated with each role is packaged separately in its own class.

Therefore,

create separate classes for the Players and the Roles. The Player class represents the fixed characteristics of the particular object and often represents a domain object. The Roles represent sets of characteristics assumed by a Player in particular circumstances. The Roles are arranged in a hierarchy, with an abstract class at its base to define the interface that all the Role subtypes will implement. The Player class carries one or more references to one or more Role hierarchies to allow dynamic associations between a Player and the Roles.

The implementations of the Player–Role and the Strategy patterns are closely similar.

References

Xavier Pacheco provides a Delphi-specific discussion of the Template Method pattern. Standard texts such as Gamma *et al*, Grand (1998) and Larman discuss both the Template Method and Strategy patterns. Lethbridge & Laganière present the Player–Role pattern.

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.

Lethbridge, T. and Laganière, R. 2001. *Object-oriented software engineering*. McGraw-Hill: Maidenhead.

Pacheco, X. 1999. *The Template Method Pattern*. The Delphi Informant. Available online at: www.delphizine.com/features/1999/05/di199905xp_f/di199905xp_f.asp

Chapter summary

Main points:

1. The Template Method Pattern: varying the separate steps of a fixed algorithm.
2. The Strategy Pattern: varying an entire algorithm.
3. The Player–Role pattern: associating a Player with different Roles.

Objects as interacting entities: Setting up links between objects and changing these links at run-time to change behaviour dynamically to meet differing circumstances and contexts.

Problems

Problem 11.1 Study Chapter 11

Identify the appropriate example(s) or section(s) of the chapter to illustrate each comment made in the summary at the end of chapter 11.

Problem 11.2 Extending the music school

Recode the Template example (example 11.2) as follows. One unit (the definition layer or framework) contains an abstract superclass to define a 'Festival Details' framework. A second unit (the default layer) contains a default class derived from the superclass. This implements all the template steps that different subclasses have in common. A third unit (the concrete layer) contains the individual concrete classes derived from the default layer to implement the specialised steps.

Also add a TTrumpet class for the Music Summer School. The venue is the Miles Davis room and the closing date for the deposit is 15/12/05.

Background for the next two questions

A savings bank needs a program to allow deposits, withdrawals and interest payments to be credited against an account (figure 20).

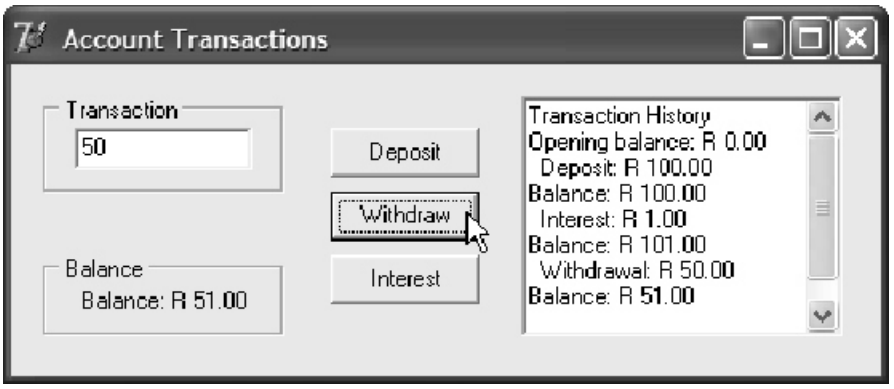

Figure 20 Interface for the savings bank application

Figure 21 Objects on the user interface

The code for the user interface class is as follows:

```
 1 unit TemplateAccU;

 2 interface

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

 6 type
 7   TfrmAccount = class(TForm)
 8     { standard RAD declarations }
20   private
21     procedure UpdateDisplay(const Balance: double;
22                             const LogStr: string);
23   end;  // end TfrmAccount = class

24 var
25   frmAccount: TfrmAccount;

26 implementation

27 uses OperationsU;

28 var
29   Balance: double = 0;
30   Log: string = '';
31   TheDeposit: TDeposit;
32   TheWithdrawal: TWithdrawal;
33   TheInterest: TInterest;

34 {$R *.dfm}

35 procedure TfrmAccount.btnDepositClick(Sender: TObject);
36 begin
37   TheDeposit.Compute(StrToFloat(edtTransaction.Text), Balance, Log);
```

```
38   UpdateDisplay(Balance, Log);
39 end; // end procedure TfrmAccount.btnDepositClick

40 procedure TfrmAccount.btnWithdrawClick(Sender: TObject);
41 begin
42   TheWithdrawal.Compute(StrToFloat(edtTransaction.Text), Balance,
43 Log);
44   UpdateDisplay(Balance, Log);
45 end;  // end procedure TfrmAccount.btnWithdrawClick

46 procedure TfrmAccount.btnInterestClick(Sender: TObject);
47 begin
48   TheInterest.Compute(StrToFloat(edtTransaction.Text), Balance, Log);
49   UpdateDisplay(Balance, Log);
50 end;  // end procedure TfrmAccount.btnInterestClick

51 procedure TfrmAccount.UpdateDisplay(const Balance: double;
52                                      const LogStr: string);
53 begin
54   lblBalance.Caption := 'Balance: ' +
55                         FloatToStrF(Balance, ffCurrency, 10, 2);
56   memHistory.Lines.Add(LogStr);
57 end;  // end procedure TfrmAccount.UpdateDisplay

58 procedure TfrmAccount.FormCreate(Sender: TObject);
59 begin
60   TheDeposit := TDeposit.Create;
61   TheWithdrawal := TWithdrawal.Create;
62   TheInterest := TInterest.Create;
63 end;  // end procedure TfrmAccount.FormCreate

64 end.  // end unit TemplateAccU
```

Problem 11.3 Applying the Template Method pattern

Implement the code for the TDeposit, TWithdrawal and TInterest classes as required to complete the program given above in the 'Background' section. Implement these classes in accordance with the *Template Method Pattern*.

Problem 11.4 Applying the Strategy pattern

Implement the code for the TDeposit, TWithdrawal and TInterest classes as required to complete the program given above in the 'Background' section. Implement these classes in accordance with the *Strategy Pattern*.

Problem 11.5 UML diagrams for the Strategy program

In example 11.3 the user selects a weekly employee on Grade A on the user interface.

1. Draw the sequence diagram for the interactions that occur.
2. Draw an object diagram to show the objects and their relationships after the Strategy objects have been created but not yet destroyed.

Problem 11.6 Differentiating between the Template and Strategy patterns

At first glance, the Template and Strategy patterns may appear to be alternative solutions to the same type of problem. Draw up a set of guidelines to help decide which of the two is better in a particular context.