Chapter 14 A Decorator

Main concepts

– Polymorphism (subtyping) in contrast to subclassing
– The Decorator pattern:
  – Optional, additional embellishments to an existing method
  – Contrast to Template and Strategy patterns
– Contrasting the Template Method, Strategy, Decorator, Factory Method and Abstract Factory Patterns

Chapter contents

Introduction 2
Background to the Decorator Pattern example 2
Example 14.1 A simple solution 4
The Decorator Pattern 5
Example 14.2 Applying the Decorator pattern 10
  Ex 14.2 step 1 Coding the Decorator 10
  Ex 14.2 step 2 Using the Decorator 12
  Ex 14.2 step 3 Another way to add the decorations 14
  Ex 14.2 step 4 Visualising the iterations 15
Decoration through aggregation or inheritance? 16
Pattern 14.1 The Decorator Pattern 18
Recapitulation 18
Chapter Summary 19
References 20

Introduction

In chapters 11 and 13 we looked at ways of making the 'same' object behave differently under different circumstances by changing the steps of an algorithm, the Template Method pattern, or by changing the algorithm itself, the Strategy pattern, or by changing the behaviour of an entire object or set of objects, the Factory patterns. These are all ways of changing the core operations of a class. In this chapter we look at situations where it is not necessary to change the core characteristics of an object at run time, but rather optionally to be able to add behaviour to the object's existing behaviour. In OO parlance we want to decorate an object and so we'll look at the Decorator pattern.

Background to the Decorator Pattern example

Joe's rather sleazy and cheesy Bioscope is a small-town cinema that shows full length features every Wednesday, Friday and Saturday. Joe's problem is that his attendance figures are falling steadily. In response he has introduced a Golden Oldies Club for senior citizens and a Silver Screen Loyalty Club for regular patrons. Members of these clubs receive tickets with additional customised annotations.

The main user interface screen for his ticket-issuing system has a RadioGroup for selecting the day of the week and CheckBoxes for the Golden Oldies and for the Loyalty Programme (figures 1&2).

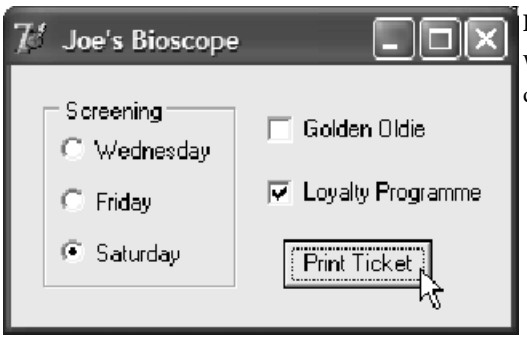

Figure 1 The user interface, with the selections that displayed figure 5

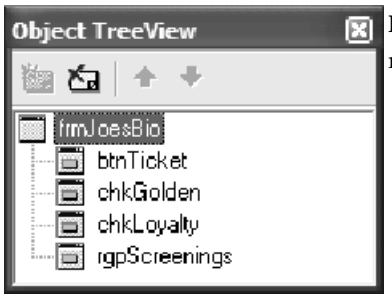

Figure 2 The objects on the revised user interface

We don't have a ticket printer, so we'll simulate it with a ShowMessage box. Figure 3 shows the standard ticket.

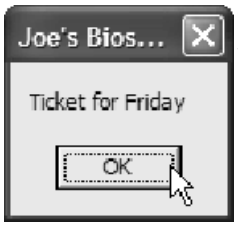

Figure 3 The ticket Message Box

Members of either or both clubs receive individually customised tickets. Golden Oldies get a special welcoming message (figure 4) while Silver Screeners (members of the loyalty programme) get a special message of appreciation (figure 5). (Members of both clubs get both messages.)

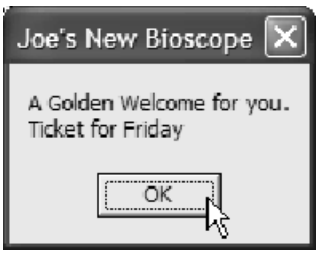

Figure 4 The Golden Oldies welcoming message

Figure 5 Message of appreciation for loyal viewers

Example 14.1 A simple solution

A Delphi program to do this can be very simple:

```
 1 unit JoesBioU;

 2 interface

 3 uses  // Standard RAD generated uses list

 6 type
 7   TfrmJoesBio = class(TForm)
 8     // Standard RAD generated component and method list
13   end; // end TfrmJoesBio = class(TForm)

14 var
15   frmJoesBio: TfrmJoesBio;

16 implementation

17 {$R *.dfm}

18 procedure TfrmJoesBio.btnTicketClick(Sender: TObject);
19 var
20   TicketMsg, Day: string;
21 begin
22   TicketMsg := '';  // initialise
23   // Golden Oldies 'decoration'
24   if chkGolden.Checked then
25     TicketMsg := 'A Golden Welcome for you.' + #13#10;
26   // Basic functionality
27   Day := rgpScreenings.Items[rgpScreenings.ItemIndex];
28   TicketMsg := TicketMsg + 'Ticket for ' + Day;
29   // Loyalty Programme 'decoration'
30   if chkLoyalty.Checked then
31     TicketMsg := TicketMsg + #13#10 +
32                  'Thanks for your return visit.';
33   // Print the required ticket
34   ShowMessage (TicketMsg);
35 end; // end procedure TfrmJoesBio.btnTicketClick

36 end. // end unit JoesBioU
```

This program works, and within this very simple context it's possibly fine as it stands, particularly if it never grows or changes in the future. But if we analyse it from the perspective of programming principles, we see that it has a very closed and ad-hoc structure. The code does not have a ticket object, and the user interface class implements both the user interface and the application logic. This makes future development and reuse difficult. What happens if we have to issue unique ticket numbers, store data about ticket

sales for auditing purposes or introduce a discount scheme with varying discounts on different days and for different groups of patrons?

Making any of these changes means that we will have to hack away at the code for the TfrmJoesBio.btnTicketClick method, and before long we could have a tangled mess. So we can set a few design criteria to improve the structure of this program:

– There should be reasonable separation between the user interface and the application logic.
– There should be a separately identifiable ticket object (class) which can centralise the application logic involved with the ticket.
– It must be easy to introduce variations and changes, both in terms of the overall structure and for each individual ticket object.

These are the issues we'll look at next.

The Decorator Pattern

Joe's Bioscope is a rather silly example, but it poses an interesting question. How does one change an object's behaviour in response to changing run time conditions? We have already looked at three approaches: the Template Method, where different subtypes implement the steps of a template in different ways, the Strategy, where a class can be associated with different algorithms or sets of rules, and the Factories, where entirely different objects are produced.

Here we'll look at a fourth way of varying an object's run time behaviour, through use of the Decorator pattern. Whereas the previous patterns change the inner workings of an object or the object itself, the Decorator adds behaviour around a fixed core object. Because of this characteristic, a Decorator is sometimes also called a Wrapper. (This is rather confusing since Wrapper is also an alternative name for the Adapter pattern. We avoid using the word Wrapper in these notes.)

To add variable behaviour around a fixed core object, we'll need a TRealClass with an Operation and a TDecorator with its Operation that in some way embellishes RealClass.Operation without replacing it. One approach is to subclass for specialisation (figure 6). With this approach, Operation in the TDecorator subclass invokes the inherited Operation and adds additional operations or decorations as specialisations of the inherited Operation (see the note in figure 6). If the additional operations (eg AddOp1 in figure 6) are invoked before the inherited operation, the decoration is applied before the core operation (to give the effect of figure 4). Similarly, if the decoration is invoked after the inherited operation, the decoration is applied after the core operation (figure 5).

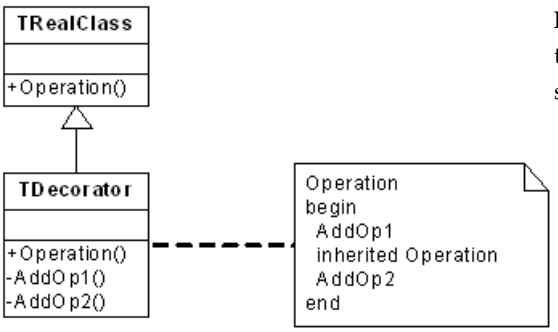

Figure 6 Adding functionality to an operation through subclassing

Another alternative is to use aggregation (figure 7). Here the Decorator also declares decorations (AddOp1, AddOp2) which it can invoke before or after delegating the core operation to the Real Class.

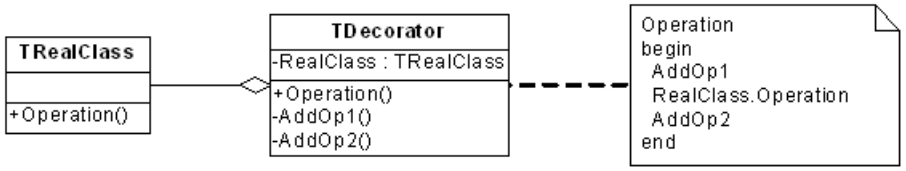

Figure 7 Adding functionality to an operation through aggregation

In either case, using the Decorator is relatively straightforward. When we want the plain, unadorned operation, we'll call RealClass.Operation. When we want the decorated operation, we'll call Decorator.Operation.

If we are wanting a relatively simple decoration, either subclassing or aggregation would be suitable. However, if we are wanting a variety of decorations or a combination of decorations, the inheritance approach rapidly leads to excessive subclassing. Hence the Decorator pattern uses aggregation (figure 7).

It would be nice if at run time the Client could call either RealClass.Operation or Decorator.Operation with the same program statement. This implies polymorphism and so we should derive both these classes from an abstract class that defines the common class interface and that RealClass and Decorator both implement (figure 8).

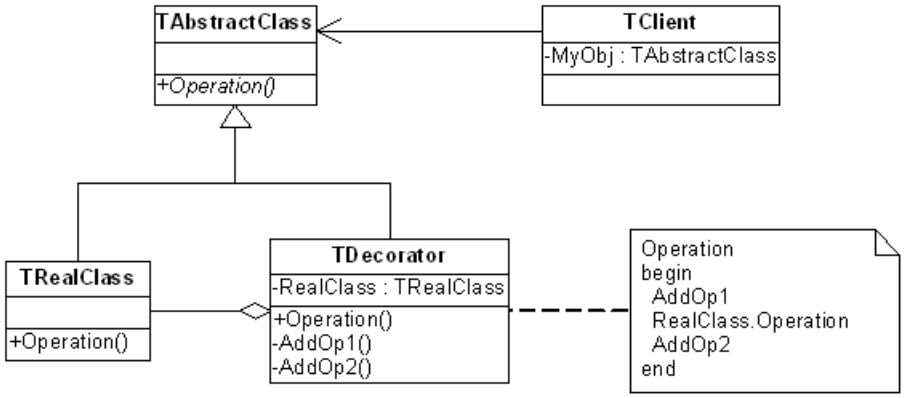

Figure 8 Making TRealClass and TDecorator consistent subtypes

Both TRealClass and TDecorator implement the public method Operation. TDecorator also has the methods AddOp1 and AddOp2, but these are private and are used internally to provide the decoration. TRealClass provides the core version of the Operation method, while Operation in TDecorator calls TRealClass's operation along with the decorations, as shown by the note in figure 8. To the Client, the interfaces of TDecorator and TRealClass are the same except for one aspect. TDecorator carries a reference to TRealClass to access its Operation, and so TDecorator's constructor's parameter list includes a reference to a TRealClass, whereas TRealClass does not have this link. Besides this exception, which we'll come back to again, TDecorator can be substituted polymorphically for a TRealClass at any time.

A TDecorator has a single link to a TRealClass, so it can perform a single decoration. Bearing in mind the 'link through the root' principle we discussed in chapter 12, the link in TDecorator should be to TAbstractClass rather than to TRealClass (figure 9).

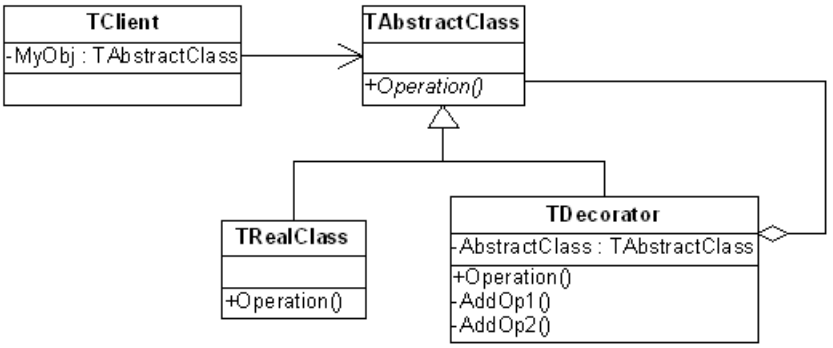

Figure 9 Linking through the root class

Let's look at what this means. At the root of the decoration hierarchy in figure 9 is a TAbstractClass with a public, abstract *Operation*. It is not designed to be instantiated but sets a minimum interface for all its descendants. TRealClass and TDecorator are descendants of

TAbstractClass. Repeated methods in these classes (here, the concrete Operation() methods in TRealClass and TDecorator) provide implementations for the abstract method in the ancestor. Other entries, (the private AddOp1(), AddOp2(), and AbstractClass: TAbstractClass in TDecorator), represent operations and data introduced in the subclasses to provide the decorations.

TDecorator has a unidirectional aggregation link with TAbstractClass. Because TRealClass and TDecorator are polymorphically interchangeable, TDecorator can now, via TAbstractClass, link to either a TRealClass or to another TDecorator. So one TDecorator can link to another TDecorator which in turn can link to a TRealClass. Actually, there can be a whole chain of linked TDecorators ultimately ending in a TRealClass. This is getting really interesting, because it means we can decorate a decoration!

Creating some object diagrams can make this seem less confusing. If the MyObj: TAbstractClass reference in TClient refers to a TRealClass, we get the undecorated case (figure 10).

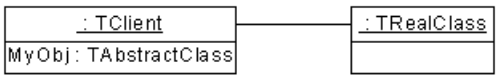

Figure 10 The undecorated case

However, if the MyObj: TAbstractClass reference in TClient refers to a TDecorator, and the AbstractClass: TAbstractClass reference in TDecorator refers to a TRealClass, we get a single decoration (figure 11).

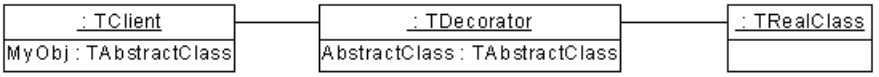

Figure 11 A single decoration

But the TDecorator may refer to another TDecorator which in turn refers to a TRealClass. This gives us a double decoration (figure 12), something not easily possible if we were using inheritance instead of aggregation to implement our decoration.

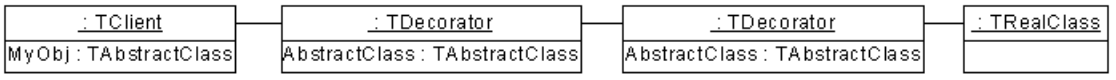

Figure 12 A double decoration

Decorating a decoration sounds good, but at the moment we have only a single decoration. So while we can decorate a decoration of a decoration, it will all be exactly the same decoration we add each time. It would be nice if we had a choice of decorations. Once we

start talking about doing the same things differently, we're talking about polymorphism and subtyping. This means that TDecorator must become a subtyping hierarchy (figure 13). (Don't worry yet about how we code all of this, just stick with the class diagrams for now.)

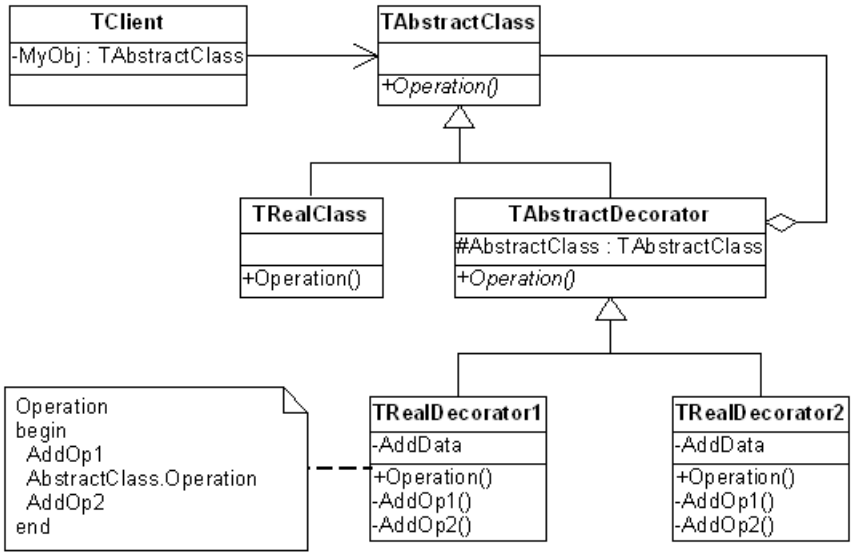

Figure 13 The Decorator pattern

We're using a variation here of the principle of linking through the base class: the aggregation link is between two abstract classes, each at the base of their respective hierarchy. So TAbstractDecorator's descendants, TRealDecorator1 and TRealDecorator2, each inherit a link to a TAbstractClass, which through polymorphism, can refer to either a TRealClass or to a descendant of TAbstractDecorator. So a TRealDecorator1 can carry a link to a TRealDecorator2 which in turn can carry a link to a TRealClass.

Now when we decorate the decoration, we have several RealDecorators to choose from.

Let's return briefly to creating and using a TRealClass and a TRealDecorator. Assume we have declared a variable MyObj of type TAbstractClass. To create and use a TRealClass, we use the code sequence:

```
MyObj := TRealClass.Create;
MyObj.Operation;
```

To create and use a TRealDecorator, we see that the constructor for a TAbstractDecorator requires a parameter of type TAbstractClass. This is because TAbstractDecorator is an aggregation that has a TRealClass as a constituent. One way to create and use a TRealDecorator1 is:

```
TempObj := TRealClass.Create;
MyObj := TRealDecorator1.Create (TempObj);
MyObj.Operation;
```

or, more simply,

```
MyObj := TRealDecorator1.Create (TRealClass.Create);
MyObj.Operation;
```

Beside the constructors, the only data or operations that have public visibility are the Operation() methods in each class. This is the interface established by the abstract root class TAbstractClass, and none of its descendants should add any public operations or data to it (except the Create and Destroy methods, which are not shown on the class diagram). Descendants may add private operations and/or data that they need to accomplish their decorations, but these should not be visible to outsiders. The reference AbstractClass that TAbstractDecorator defines for the aggregation with the constituent TAbstractClass is declared as protected so that it is available to subtypes, even if they are declared in a separate unit, but not to any other classes.

Example 14.2 Applying the Decorator pattern

Maybe this has been enough talking about the principles for now, and it is time to move on to the code and to apply the Decorator principles.

Ex 14.2 step 1 Coding the Decorator

We'll start with the code for the Decorator classes to suit the Joe's Bioscope application.

```
 1 unit TicketU;

 2 interface

 3 type
 4   TAbsTicket = class(TObject)      // the TAbstractClass in figure 13
 5   public
 6     function GetMessage (ADay: string): string; virtual; abstract;
 7   end; // end TAbsTicket = class(TObject)

 8   TRealTicket = class(TAbsTicket)      // the TRealClass in figure 13
 9   public
10     function GetMessage (ADay: string): string; override;
11   end; // end TRealTicket = class(TAbsTicket)
```

```
12   TAbsDecorator = class(TAbsTicket)//the TAbstractDecorator in fig 13
13   protected
14     TheTicket: TAbsTicket; //aggregation: reference to the Real Class
15   public
16     constructor Create(ATicket: TAbsTicket); //propagate construction
17     destructor Destroy; override;            // propagate destruction
18   end; // end TAbsDecorator = class(TAbsTicket)

19   TLoyalty = class(TAbsDecorator)    // the TRealDecorator1 in fig 13
20   public
21     function GetMessage (ADay: string): string; override;     // deco
22   end; // end TLoyalty = class(TAbsDecorator)

23   TGolden = class(TAbsDecorator)     // the TRealDecorator2 in fig 13
24   public
25     function GetMessage (ADay: string): string; override;     // deco
26   end; // end TGolden = class(TAbsDecorator)

27 implementation

28 { TRealTicket }                        // the TRealClass in figure 13

29 function TRealTicket.GetMessage(ADay: string): string;
30 begin
31   Result := 'Ticket for ' +ADay;   //basic functionality, undecorated
32 end; // end function TRealTicket.GetMessage

33 { TAbsDecorator }              // the TAbstractDecorator in figure 13

34 constructor TAbsDecorator.Create (ATicket: TAbsTicket);
35 begin
36   inherited Create;
37   TheTicket := ATicket;              // link in the constituent class
38 end; // end constructor TAbsDecorator.Create

39 destructor TAbsDecorator.Destroy;
40 begin
41   TheTicket.Free;       // propagate Destroy to the constituent class
42   inherited;
43 end; // end destructor TAbsDecorator.Destroy

44 { TLoyalty }                     // the TRealDecorator1 in figure 13

45 function TLoyalty.GetMessage(ADay: string): string;
46 begin
47   Result := TheTicket.GetMessage(ADay) + #13#10;       // 'recursive'
48   Result := Result + 'Thanks for your return visit.';//add decoration
49 end; // function TLoyalty.GetMessage

50 { TGolden }                      // the TRealDecorator2 in figure 13

51 function TGolden.GetMessage(ADay: string): string;
52 begin
53   Result := 'A Golden Welcome for you.' + #13#10;   // the decoration
54   Result := Result + TheTicket.GetMessage(ADay);       // 'recursive'
55 end; // end function TGolden.GetMessage
```

```
56 end. // end TicketU
```

Compare this class definition with the class diagram in figure 13 and check for yourself that the two correspond. It may not all quite make sense yet, but we'll go through it once we have created the code for the user interface that uses the Decorator.

As a comment on programming detail, note that an abstract method must be defined at some point in the hierarchy, but not necessarily in the layer immediately below its declaration or in the final layer. In the code above, TAbsDecorator does not need to either redeclare GetMessage as abstract or provide an override definition, even though it is declared as abstract in its ancestor. This is because GetMessage is overridden in both of TAbsDecorator's descendants. (One could declare a GetMessage override method in TAbsDecorator to act as a default that descendants could override optionally. However it does not seem necessary in this case.)

Although TAbsDecorator is an abstract class, it still declares concrete constructors and destructors that TLoyalty and TGolden inherit. So, though TAbsDecorator has some abstract methods and so is not a class one could instantiate, it can be used to define methods that are common to all its descendants. These methods then need to be defined once only, and not repeatedly in each subclass.

Notice that TAbsDecorator.Create receives a reference to a TAbsTicket as a parameter (line 34), and so does not instantiate an associated TAbsTicket but merely assigns its private reference to this incoming parameter (line 37). (As we saw in the discussion just before this step, the calling statement creates the associated object and passes it as a parameter. We will also see this in the next step.) However, TAbsDecorator.Destroy has the responsibility of freeing the associated object (line 41).

Ex 14.2 step 2 Using the Decorator

The user interface is shown in figures 1 & 2. The code is as follows:

```
 1 unit JoesBioU;

 2 interface

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

 7 type
 8   TfrmJoesBio = class(TForm)
 9     rgpScreenings: TRadioGroup;
10     chkGolden: TCheckBox;
```

```
11      chkLoyalty: TCheckBox;
12      btnTicket: TButton;
13      procedure btnTicketClick(Sender: TObject);
14    private
15      YourTicket: TAbsTicket;
16    end;

17 var
18    frmJoesBio: TfrmJoesBio;

19 implementation

20 {$R *.dfm}

21 procedure TfrmJoesBio.btnTicketClick(Sender: TObject);
22 var
23    Day: string;
24 begin
25    // select the decorations at run time
26    if chkLoyalty.Checked then
27      if chkGolden.Checked then
28        YourTicket :=
29                    TLoyalty.Create(TGolden.Create(TRealTicket.Create))
30      else
31        YourTicket := TLoyalty.Create(TRealTicket.Create)
32    else
33      if chkGolden.Checked then
34        YourTicket := TGolden.Create(TRealTicket.Create)
35      else
36        YourTicket := TRealTicket.Create;

37    // use the (decorated) operation GetMessage
38    Day := rgpScreenings.Items[rgpScreenings.ItemIndex];
39    ShowMessage (YourTicket.GetMessage(Day));
40    YourTicket.Free;
41 end; // end procedure TfrmJoesBio.btnTicketClick

42 end. // end unit JoesBioU
```

Lines 26 to 36 show one way to use the Decorators. Let's start with line 36, which runs if neither CheckBox is checked. Line 36 then simply creates a plain, ordinary TRealTicket without any decorations. Moving backwards, if Golden Oldies is checked but not the Loyalty programme, line 34 runs. This demonstrates how to set up the chain of links between the decorators and the real class object. First line 34 creates an undecorated TRealTicket and then passes a reference to this TRealTicket as a parameter to TGolden's constructor, which creates a Golden decorator around the TRealTicket. Similarly, line 31, which runs if Loyalty is checked but Golden is not, first creates a plain TRealTicket and then wraps a Loyalty decorator around it. Finally, lines 28–29, which run when both Golden and Loyalty are checked, first create a plain TRealTicket, then they wrap a Golden around it, and then wrap a Loyalty around the Golden TRealTicket. In all four of these operations, a

TRealTicket object is created first and then is included as a constituent of if a decorator object is created subsequently.

Whichever of these four operations occurs, the end result is a YourTicket object that the subsequent program steps use to access the GetMessage operation, automatically including whatever decorations it may have (line 39).

Ex 14.2 step 3 Another way to add the decorations

Another approach for adding the decorations is to replace lines 26–36 in step 2 by the following:

```
26   YourTicket := TRealTicket.Create;
27   if chkGolden.Checked then
28     YourTicket := TGolden.Create(YourTicket);
29   if chkLoyalty.Checked then
30     YourTicket := TLoyalty.Create(YourTicket);
```

We can explain this second version by going through it line by line with the assumption that both chkGolden and chkLoyalty are checked. Line 26 creates a TRealTicket with the reference YourTicket (figure 14).

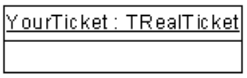

Figure 14 TReal-Ticket instance at the heart of the Decorator

Because chkGolden is checked, line 28 executes. This passes YourTicket as a parameter to TGolden's constructor and then reassigns the YourTicket reference to this TGolden object to achieve the first level of decoration (figure 15).

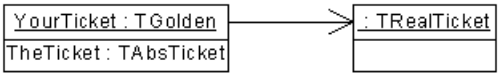

Figure 15 A TGolden decorating the TRealTicket

chkLoyalty is also checked, so line 30 executes next. This passes the YourTicket reference, in other words, the decorated TRealTicket, to TLoyalty's constructor and then reassigns the YourTicket reference to this TLoyalty object to achieve the second level of decoration (figure 16).

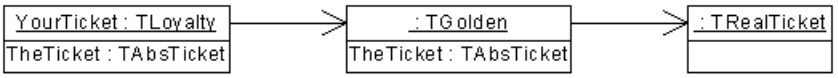

Figure 16 A TLoyalty decoration of the TGolden decorating the TRealTicket

When YourTicket, the first object in figure 16, executes the GetMessage method, this invokes TheTicket's GetMessage method (step 1, line 47). At this stage, TheTicket refers to an object of type TGolden, the second object in figure 16. Thus TGolden's GetMessage executes and creates the GoldenWelcome string (step 1, line 53). It then calls TheTicket's GetMessage method (step 1, line 54). At this stage, TheTicket refers to a TRealTicket object, the final object in figure 16. TRealTicket's GetMessage runs (step 1, line 31) and creates the 'Ticket for ...' string. This method completes, and so the program unwinds to TGolden's GetMessage. This concatenates the two strings (step 1, line 54) and then completes, and so the program unwinds to TLoyalty's GetMessage. This adds the 'Thanks ...' string to the message string (step 1, line 48) and then terminates. The btnTicketClick event handler then shows the final message (ShowMessage in step 2, line 41). Notice that the sequence of object instantiations comes to an end when it encounters the real class, since only a decorator has a reference to a further class.

These object diagrams, drawn from considering the code, match directly the diagrams we drew from the class diagrams earlier (figures 10–12).

Enough said! If you have not already done so, run this program and check that the different decorations work as stated and try the alternative instantiation of the decorators given in lines 26–30 above.

Ex 14.2 step 4 Visualising the iterations

In step 3 above we described how the Decorator leads to the instantiation and use of repeated objects. One way to see this in the program is to single step through this program with the debugger to confirm the description of how decorating works. Another option is to add code to the constructors and destructors to provide trace messages that identify the object currently being created or destroyed. To do this, add a constructor and a destructor to TAbsTicket (step 1, lines 4–7):

```
TAbsTicket = class(TObject)
public
  function GetMessage (ADay: string): string; virtual; abstract;
  constructor Create;                    // for illustration purposes
  destructor Destroy; override;          // for illustration purposes
end; // end TAbsTicket = class(TObject)
```

Add a 'uses Dialogs' clause just after the Implementation keyword and add suitable code for the constructor and destructor:

```
{ TAbsTicket }

constructor TAbsTicket.Create;
begin
  ShowMessage ('Creating ' + ClassName);            // for illustration
  inherited;
end; // end constructor TAbsTicket.Create

destructor TAbsTicket.Destroy;
begin
  ShowMessage ('Destroying ' + ClassName);          // for illustration
  inherited;
end; // end destructor TAbsTicket.Destroy
```

Run the program again, using different combinations of decorations.

Decoration through aggregation or inheritance?

The recursive aggregation of the Decorator pattern at first seems rather complex. Is it not easier just to use inheritance? In this section we'll compare these two approaches. Assume that we have a class RealClass with a core operation Operation and three decorations (each of which is also be called Operation) that can optionally be applied singly, in twos or all three. (To keep things simpler, assume that the sequence of the decorations is not important.)

With the Decorator pattern, which combines aggregation and polymorphism, we need six classes in all (figure 17).

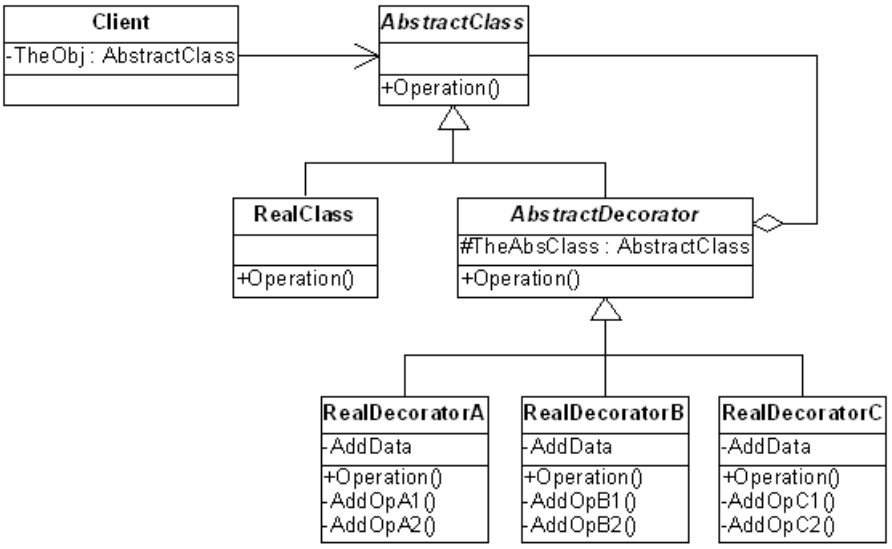

Figure 17 Providing three decorations with the Decorator pattern

To do the same with inheritance alone requires eight classes, the RealClass plus a subclass for each required decoration or combination of decorations (figure 18). Though we have more classes in total than with the Decorator pattern, the structure seems initially to be simpler. However, notice how much repetition the inheritance-only approach requires. The identical method AddOpA1, for example, is implemented in four classes (RealDecoratorA, RealDecoratorAB, RealDecoratorAC and RealDecoratorABC). Each other additional operation is also implemented four times. With the Decorator pattern each additional operation is implemented only once. This is because the inheritance-only approach requires a separate subclass for each decoration combination, while with the Decorator pattern, the Client object builds up the decoration as needed through its aggregation links.

In the inheritance-only approach (eg figure 18), a separate subclass is needed for each combination of decorations. So if the order of the decorations is important, ie if RealDecoratorAB is different from RealDecoratorBA, and if the application requires both of these, the inheritance-only approach requires that each of these is implemented separately. This means even more subclasses and more repetition while the Decorator requires no further classes. If only some of the possible combinations are needed, the classes in figure 18 that are not needed need not be implemented. However, even when only a few combinations are needed it may still be worth using the Decorator pattern since this allows greater flexibility for possible future changes.

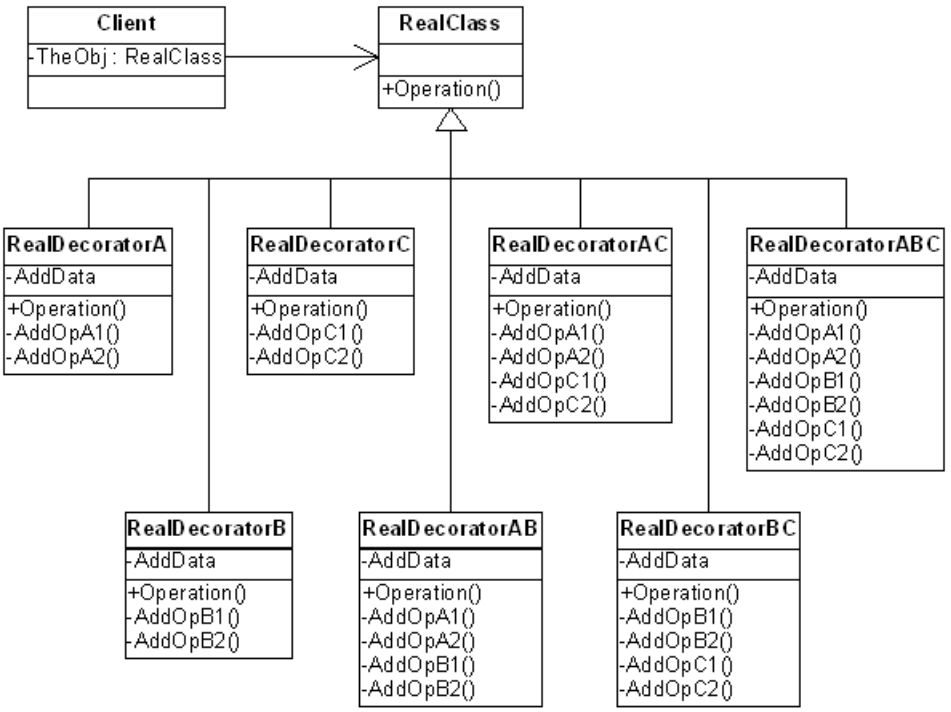

Figure 18 Providing the decorations through inheritance

Pattern 14.1 The Decorator Pattern

We can describe the Decorator pattern as follows.

Consider the case where an existing class performs a certain core operation. However, it is sometimes necessary at run time to add one or more additional operations in varying combinations to this core operation. These additional operations are not permanent and are not always needed. Creating a separate subtype for each individual possibility will be cumbersome because of the number of subtypes that will be needed and/or because inheritance is fixed at compile time and so lacks the required flexibility.

Therefore,

create an abstract superclass for the existing class. From this superclass, derive additional subtypes, the decorators, that wrap the existing class and that provide the additional operations (decorations) before or after delegating the core operation to the existing class as

appropriate. Decorations to the core functionality can be added in small increments and can be applied in different combinations and sequences.

If there are more than just a few required combinations, Decorators lead to a smaller total number of classes than inheritance does and to simpler individual classes. However, Decorators can lead to a proliferation of small objects, and there is the danger that the decorations are applied in inappropriate sequences or combinations.

The Decorator pattern is based on delegation and on one or more related wrappers. It provides variable operations while maintaining a consistent interface, and so differs from the Adapter, which typically adapts the interface of an object without changing its operation. In comparison to other patterns, the Decorator allows a variable number of steps in a varying sequence. The Template Method enforces a specific number and sequence of steps while allowing the operation of each step to vary. The Strategy varies the entire operation.

Recapitulation

The following table consolidates and contrasts the material covered in the last few chapters:

In a …	*the client wants …*
Template Method Pattern	a) the result of a particular operation which has fixed steps, but b) where the details of individual steps may vary (either between subclasses or between applications), and possibly c) where different variations of the steps may have to be added in the future. – variation in *behaviour*. – different steps in an algorithm.
Strategy Pattern	the result of an entire operation, rather than the individual steps of the operations, to depend on a particular context. – variation in *behaviour*. – different algorithms.
Decorator Pattern	to add one or more optional embellishments to an existing method. – variation in *structure.* – different embellishments.

In a …	*the client wants …*
Factory Method Pattern	an object as determined by a particular context. – *creation* of an object. – different objects.
Abstract Factory Pattern	a family of related objects as determined by a particular context. – *creation* of a set of objects. – different sets of objects.

Chapter Summary

– Polymorphism (subtyping) in contrast to subclassing
– The Decorator pattern:
  – Optional, additional embellishments to an existing method
  – Contrast to Template and Strategy patterns
– Contrasting the Template Method, Strategy, Decorator, Factory Method and Abstract Factory Patterns

References

The Decorator pattern is widely known and is discussed in standard texts such as Gamma *et al* and Grand (1998). It is also often presented in more introductory books such as Shalloway and Trott.

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

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

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

Problems

Problem 14.1 Study Chapter 14

Identify the appropriate example(s) or section(s) of the chapter to illustrate each comment made in the summary at the end of chapter 14.

Problem 14.2 A simple Decorator

This problem is interesting in its own right, and is also interesting as a basis for the next problem.

Employees receive a basic wage, which is entered into the program (figures 19 & 20). In addition to this, they may earn a travel allowance, which is a flat amount of R400, and/or a housing subsidy, which is 10% of their basic. Implement this program using the Decorator pattern to add these payments and to provide an enhancement path for future expansion. Integer arithmetic is acceptable.

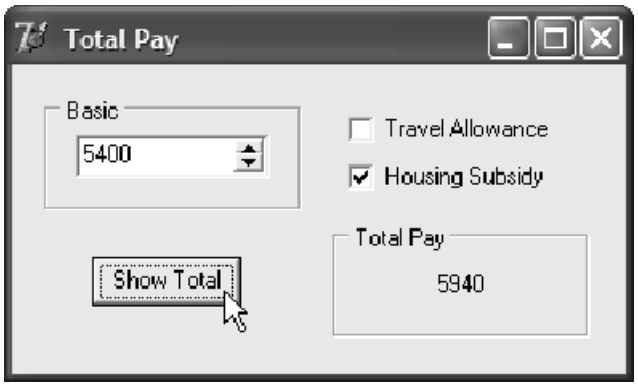

Figure 19 Calculating benefits

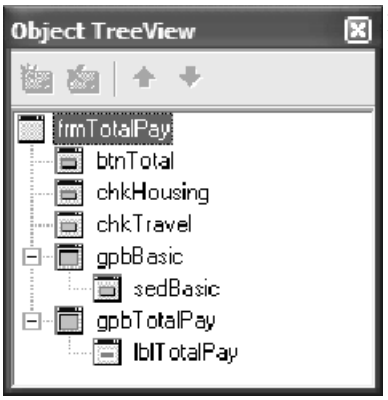

Figure 20 User interface objects

Problem 14.3 Combining patterns: Decorating a Player-Role application

Example 12.5 calculates an employee's basic wage based on his or her category, notch, and so on. Problem 14.2 calculates an employee's total wage by adding benefits payable in addition to the basic wage. Combine these two programs to calculate an employee's total wage after benefits have been added to a basic calculated from category, notch, etc, (figures 21 & 22).

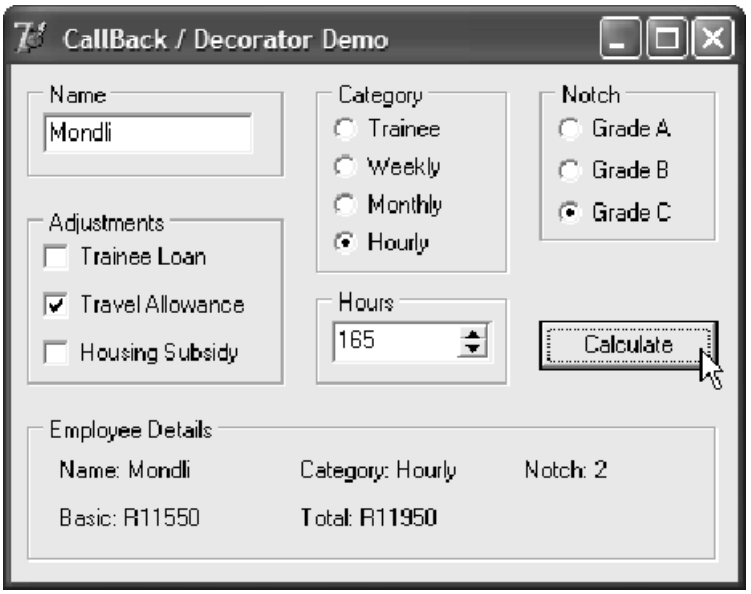

Figure 21 A decorated Player-Role application

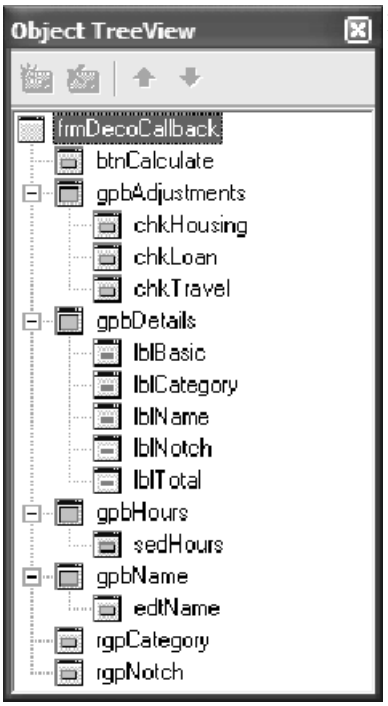

Figure 22 Interface objects