FIVE

Designing programs

The projects that we have worked on so far have been relatively straightforward to program directly on the computer. With more complex projects, however, it is necessary to carefully plan the work beforehand to ensure that the finished program will meet the client's requirements. In this chapter we will look at some of the design methods used by programmers.

Railway tickets

A narrow gauge railway in North Wales requires a computer program to calculate ticket prices and keep a record of the total value of tickets issued each day.

The fares are calculated according to the following rules:

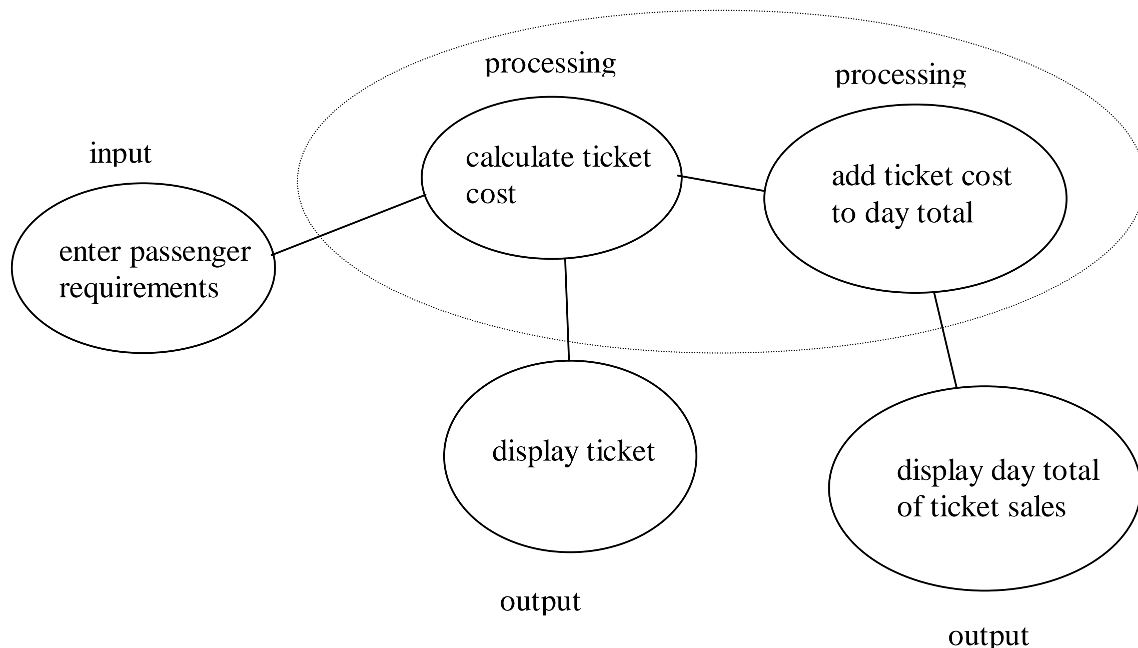
- The adult single fare for a journey along the line is £3.20.
- The child single fare is 60% of the adult fare.
- The return fare is 1½ times the single fare.
- Each passenger wishing to travel first class pays a supplement of 80 pence, which covers either a single or return journey.
- Groups of 4 or more people travelling together receive a discount of 10% on the total fare.

You are asked to design and produce a program to issue the tickets.

This is quite a complex problem. The first step in program design is often to produce a schematic diagram. We begin by writing down each of the tasks which the program is to carry out. These tasks can then be classified as either:

- INPUT, where information is entered using the keyboard or mouse
- PROCESSING, where the computer carries out calculations or sorting of the data
- OUTPUT, where results are displayed on the screen or printed out on paper.

PROGRAM SCHEMATIC



The stages of the program have been linked by arrows, and the processing stages have been enclosed with a dotted line to make the structure clearer. We now have a better idea of the work which will be required.

First we might turn our attention to the input of the passenger requirements. A useful way of planning this stage is to use a technique called an '**algorithm progressive refinement sequence**'. We begin by writing down the overall objective:

1. input passenger requirements

This can then be subdivided into:

1.1 input the number of passengers
1.2 input single/return journey
1.3 input first/second class

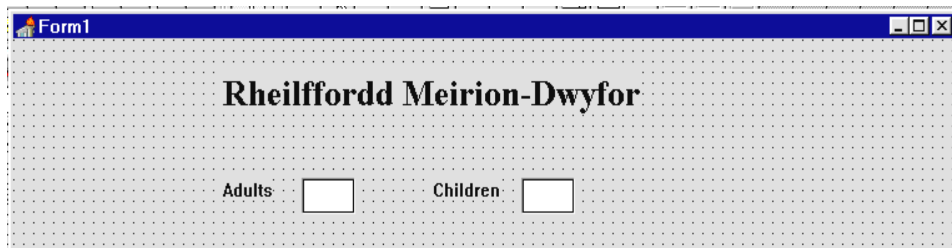
Step 1.1 could be further subdivided:

1.1.1 input the number of adults
1.1.2 input the number of children
1.2 input single/return journey
1.3 input first/second class

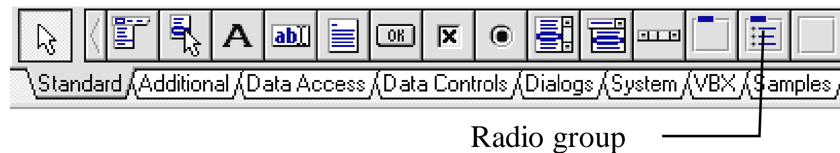
The process of refining the design has continued until we know exactly what inputs will be needed. Now is the time to begin work on the program:

Set up a new sub-directory called TICKETS. Open a Delphi project and save it into the sub-directory. Use the Object Inspector to maximize the form, and drag the grid larger to nearly fill the screen.

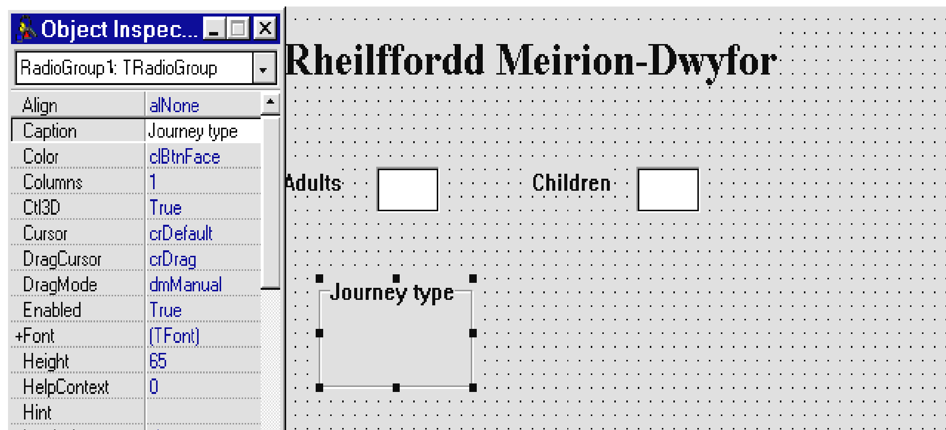
From the refinement sequence above, we can see that four inputs will be required. The numbers of adults and children can be entered using edit boxes; add these to the form and place labels alongside:



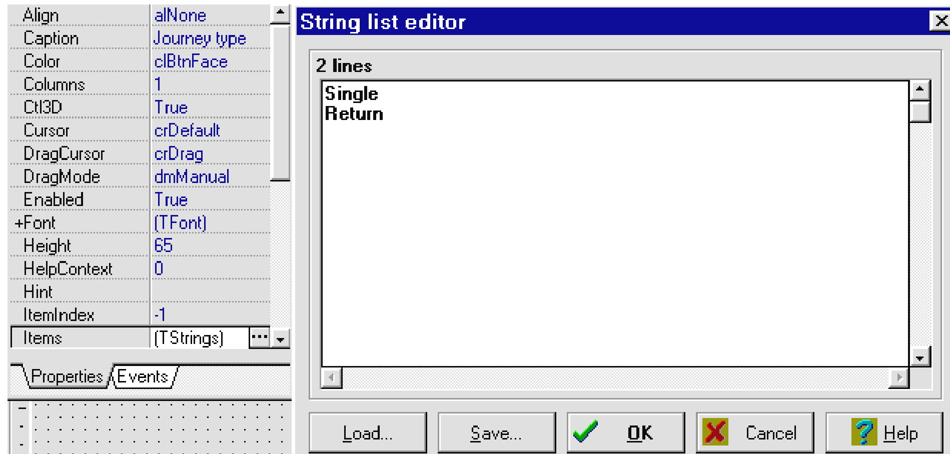
The next step is to input the type of journey: single or return. A convenient way of doing this is to use a **radio button group**. On the STANDARD component menu select *radio group*:



Place a radio group box on the form grid. Press ENTER to bring up the Object Inspector and add the caption 'journey type':

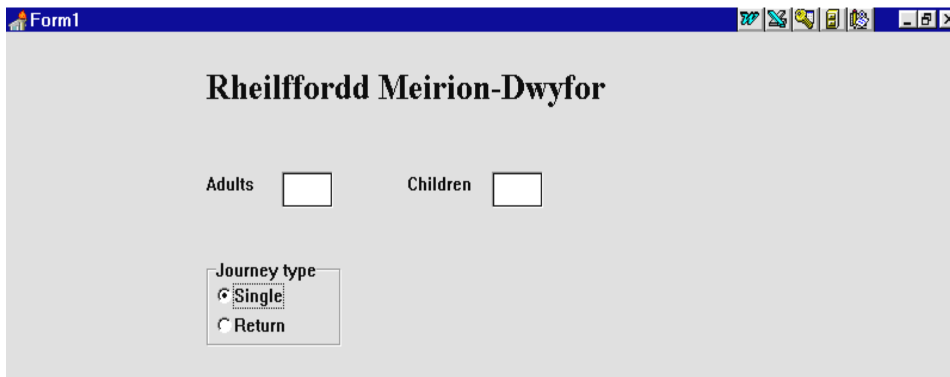


Now go to the **Items** property and double-click the right hand column to bring up the **String list editor** window:

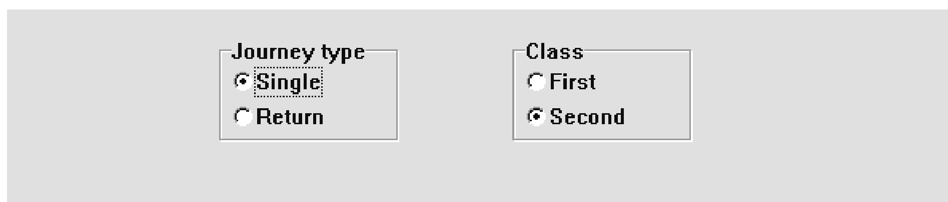


Type in the entries '**Single**' and '**Return**'. Click **OK** and two small round buttons labelled 'Single' and 'Return' should appear inside the border of the radio group box. Adjust the box side if nesessary so that the labels are displayed neatly.

Compile and run the program to test the radio group. It should be possible to select **either** the **'Single' or** the **'Return'** option, but not both at once:



Return to the Delphi editing screen and set up a similar radio group box alongside to input the choice of **first class** or **second class**. Run the program to test this:



It is now necessary to add **event handler** procedures for the inputs. Double-click on the edit box for 'Adults' and add lines of program to the **Edit1Change** procedure:

```
procedure TForm1.Edit1Change(Sender: TObject);
begin
  if edit1.text='' then
    adult:=0
  else
    adult:=strtoint(edit1.text);
end;
```

This will store the number of adults as an integer variable called '**adult**'. Produce a similar event handler for the 'Children' edit box:

```
procedure TForm1.Edit2Change(Sender: TObject);
begin
  if edit2.text='' then
    child:=0
  else
    child:=strtoint(edit2.text);
end;
```

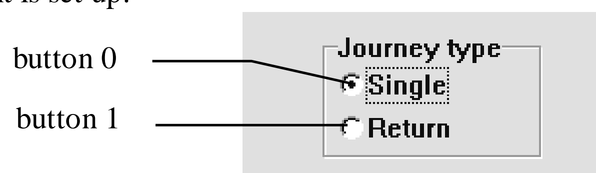
Add the variables to the **public declarations** list:

```
public
   { Public declarations }
   adult,child:integer;
```

We can now create an event handler for the '**Journey type**' radio group. Double-click the radio group box to set up the procedure then add the following lines of program:

```
procedure TForm1.RadioGroup1Click(Sender: TObject);
begin
  if radiogroup1.itemindex=1 then
    return:=true
  else
    return:=false;
end;
```

This is quite complicated and needs some explanation. The buttons of the radio group are given reference numbers by the computer when the component is set up:



We can find which button has been selected by checking the **itemindex** value for the radio group: if this is 0 then the journey type is 'Single', if it is 1 then the journey type is 'Return'.

A convenient way to record the journey type is to use a new kind of variable called a **Boolean** (named after George Boole who developed the mathematical logic used by computers). A Boolean variable can have only the values 'TRUE' or 'FALSE'.

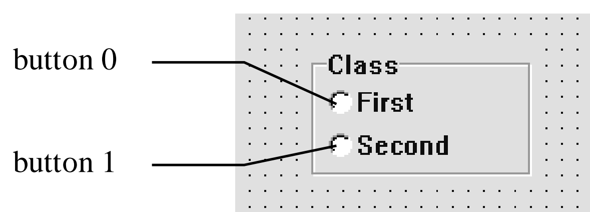
We are going to use a Boolean variable with the name '**return**' to have the following meanings:

return = TRUE	return ticket required
return = FALSE	single ticket required.

Create a similar event handler for the 'Class' radio group:

```
procedure TForm1.RadioGroup2Click(Sender: TObject);
begin
  if radiogroup2.itemindex=0 then
    firstclass:=true
  else
    firstclass:=false;
end;
```

The Boolean variable this time is called '**firstclass**'. It will be set to TRUE if a first class ticket is required, or to FALSE for a second class ticket. Notice that the 'First' button is at the top of the radio group so will be numbered 0:



Add the Boolean variables to the **public declarations** list:

```
public
   { Public declarations }
   adult,child:integer;
   firstclass,return:boolean;
 end;
```

This completes the handling of inputs and we can turn our attention to processing.
Before starting a calculation procedure in the program, let's use an **algorithm progressive refinement sequence** to find exactly what is required. (The term '**algorithm**' means '*the sequence of instructions needed to carry out a task*').

Begin with the overall task. We can number this 2 since it is the second stage of the program:

2. calculate ticket cost

We might now devise a strategy for carrying out the calculation:

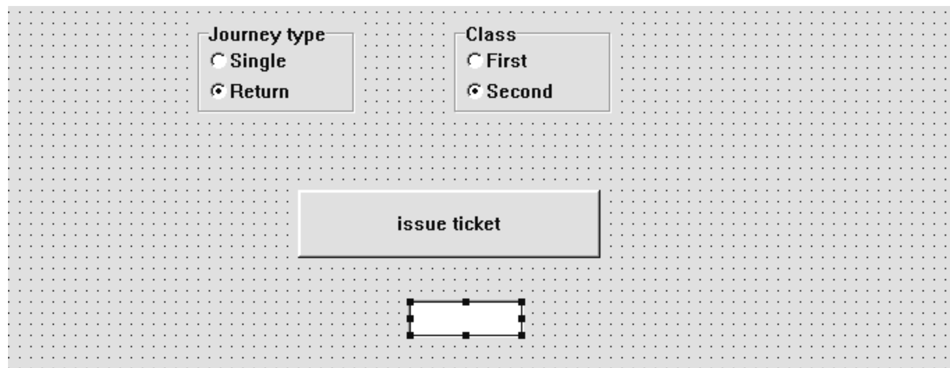
2.1 calculate the single journey cost for the passengers
2.2 **if** a return ticket is required **then**
2.3 add the extra for a return fare
2.4 **end if** *{end of the 'if' condition}*
2.5 **if** the passengers wish to travel first class **then**
2.6 add the supplement for first class
2.7 **end if**
2.8 **if** there are four or more passengers **then**
2.9 deduct the group discount
2.10 **end if**

Step 2.1 can be further refined:

2.1.1 find the single journey cost for the adults
2.1.2 add the single journey cost for the children

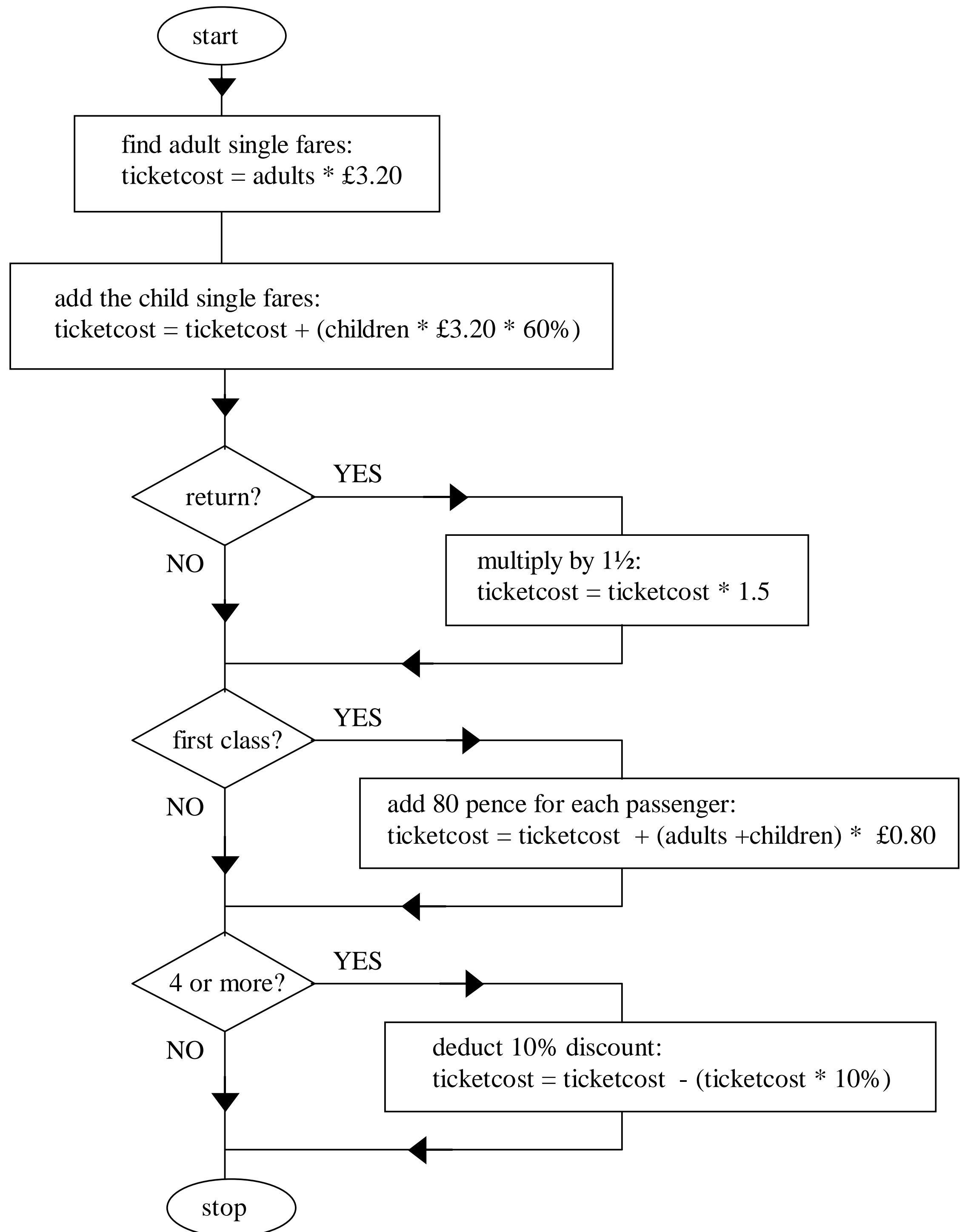
Once a detailed algorithm has been produced, the clearest way of displaying this is to draw a flow chart, as shown on the next page. The hard work of planning the calculation is now completed and it will be relatively simple to write the program.

Add a button and label this 'issue ticket' :



An edit box will also be needed to display the ticket total during the testing of the calculation procedure. Put this below the 'issue ticket' button.

Flow chart for the ticket calculation



Double-click the '**issue ticket**' button to create an event handler and add program lines to calculate the ticket cost:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
   ticketcost:=adult*3.20;
   ticketcost:=ticketcost + child*3.20*0.60;
   if return=true then
      ticketcost:=ticketcost*1.5;
   if firstclass=true then
      ticketcost:=ticketcost + (adult+child)*0.80;
   if (adult+child) >=4 then
      ticketcost:=ticketcost - (ticketcost*0.10);
   edit3.text:=floattostrf(ticketcost,ffFixed,8,2);
end;
```

Make sure that you can relate these lines of program to the steps on the flow chart.

It will be necessary to add '**ticketcost**' as a real number variable:

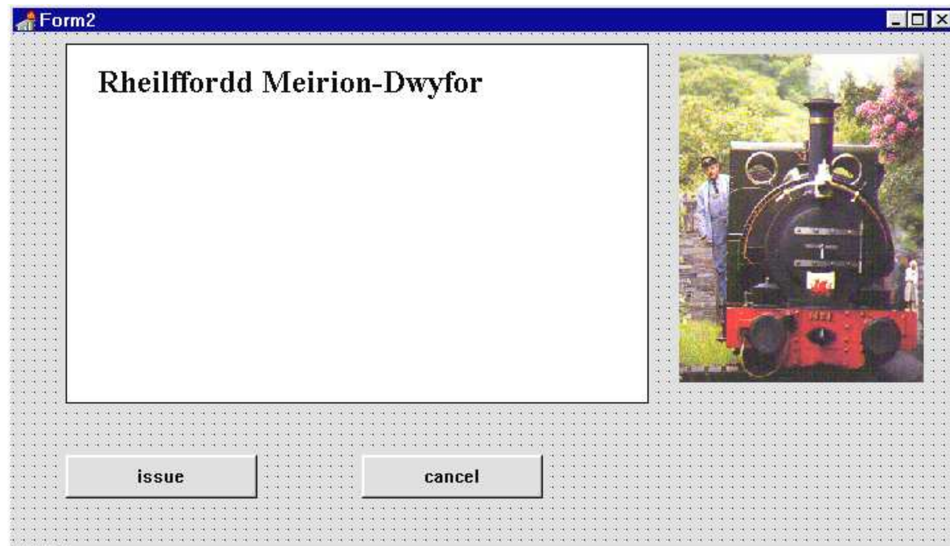
```
      public
         { Public declarations }
         adult,child:integer;
         ticketcost:real;
         firstclass,return:boolean;
       end;
```

Compile and run the program to see if it works correctly. Try out test data for different groups of passengers - some travelling first class and some second class, some making single journeys and some return. Remember that groups of 4 or more will have a 10% discount. Check the results with a calculator.

If the calculation procedure works correctly we can now turn our attention to displaying a ticket - in a real system this would be printed out on a printer. The ticket should show the name of the railway and details of the passengers' journey, as well as the fare to be paid. It will be best to display this as a separate window.

Use the '*new form*' short cut button to create a **Form2** screen grid. Use the '*save project*' button to save the accompanying program **Unit2.pas** into your project sub-directory.

From the ADDITIONAL component menu select the '*shape*' component, and place a white rectangle in the form window. Use a '*label*' component to show the name of the railway. Alongside the rectangle add an *image* box, and load the bitmap file TRAIN.BMP. Complete the window with two buttons with the captions: 'issue' and 'cancel':



Click on the dotted grid, press RETURN to bring up the Object Inspector for Form 2, then set the **Border Style** property to **Dialog**. This will ensure that the size of the form cannot be changed while the program is running.

Use the project manager to select the Form1 window. Double-click the 'issue ticket' button to bring up the event handler. Replace the

    'edit3.text:=...'

line with

    'form2.visible:=true'

as shown below:

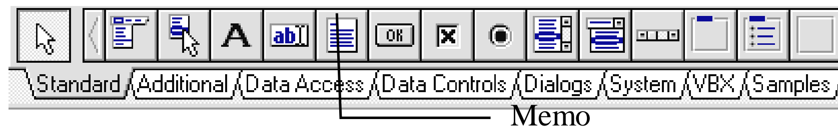
```
procedure TForm1.Button1Click(Sender: TObject);
begin
   ticketcost:=adult*3.20;
   ticketcost:=ticketcost + child*3.20*0.60;
   if return=true then
      ticketcost:=ticketcost*1.5;
   if firstclass=true then
      ticketcost:=ticketcost + (adult+child)*0.80;
   if (adult+child) >=4 then
      ticketcost:=ticketcost - (ticketcost*0.10);
   form2.visible:=true;
end;
```

Go to the 'Uses' line near the top of the program and add 'Unit2' to the list:

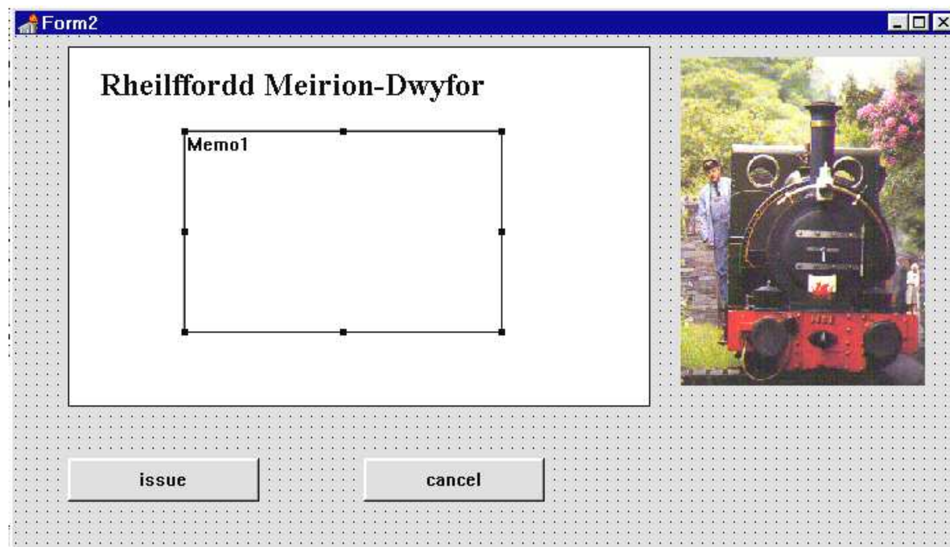
```
uses
  SysUtils,WinTypes,WinProcs,Messages,Classes,Graphics,
Controls,Forms,Dialogs,StdCtrls,ExtCtrls, unit2;
```

Compile the project using '**Build All**'. Run the program and enter some ticket data. Click the 'issue ticket' button on Form1. The Form2 window should now appear, ready to display the ticket. Exit from the program by clicking the cross in the top right hand corner of the Form1 window.

Use the Project Manager to bring the Form2 grid to the front. Select the '*memo*' component from the STANDARD menu:



Drag the mouse to position a memo box on the form:



A memo box is similar to an edit box component, except that it allows multiple lines of text to be displayed and/or edited. We will use the memo to show the ticket details.

Click on the Form2 grid and press ENTER to bring up the Object Inspector window.

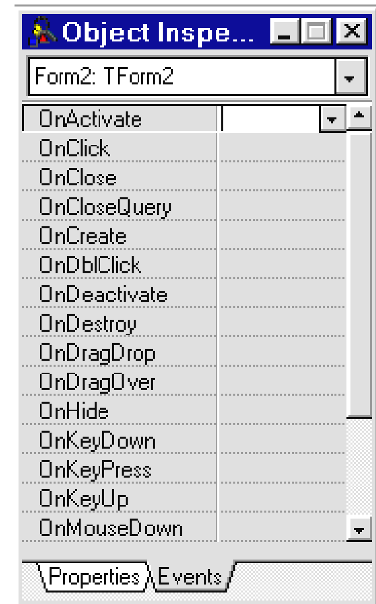
Click the 'events' tab at the bottom of the Object Inspector. This brings up a list of events which can occur while the form is in use - the most familiar will be:

OnClick

meaning that the user has clicked the mouse on the form, and

OnKeyPress

which means that the user has pressed a key on the keyboard while using the form. We can link *event handler* procedures to any of these events.



For the present program we are going to create an *event handler* for the

On Activate

event. This procedure will be used to display ticket details as soon as Form2 is activated and appears on the screen.

Double-click the right hand column alongside 'On Activate' and the event handler will appear. Add lines of program as shown below:

```
procedure TForm2.FormActivate(Sender: TObject);
var
  textline:string;
begin
  memo1.clear;
  if form1.adult>0 then
  begin
    textline:=inttostr(form1.adult)+ ' adults';
    memo1.lines.add(textline);
  end;
end;
```

The purpose of this is to display the number of adults travelling.

The line

memo1.clear;

blanks out the memo box.

The '**if**' condition will operate only if the number of adults is greater than zero. The program builds up a line of text by converting the integer variable '**adult**' from Form1 and then adding to this the caption ' *adults*' . The whole line is then added to the memo box and displayed on the screen.

We must warn the computer that it will need a variable from Form1 - we do this by adding lines of program under the 'implementation' heading:

```
implementation

{$R *.DFM}

uses
  unit1;
```

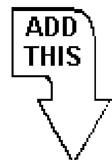
Before testing the program, double-click the 'cancel' button of Form2 to create an event handler, and add a line of program to close the window:

```
procedure TForm2.Button2Click(Sender: TObject);
begin
  form2.visible:=false;
end;
```

We can now use the 'Build All' option to compile the project. Run the program and test it by entering a numbers of adults. Press the 'issue ticket' button and the correct number should be displayed. Press' cancel' to close the window. Repeat this a few times with different numbers, then return to the Delphi editing screen.

A similar set of lines will be needed to display the number of children. Add this and test the program:

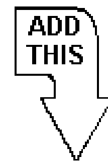
```
procedure TForm2.FormActivate(Sender: TObject);
var
  textline:string;
begin
  memo1.clear;
  if form1.adult>0 then
  begin
    textline:=inttostr(form1.adult)+ ' adults';
    memo1.lines.add(textline);
  end;
  if form1.child>0 then
  begin
    textline:=inttostr(form1.child)+ ' children';
    memo1.lines.add(textline);
  end;
end;
```



We now wish to show whether the ticket is *first* or *second* class, *single* or *return*. The boolean variables '**firstclass**' and '**return**' can be used to do this.

Go to the **FormActivate** event handler again and add the extra lines of program:

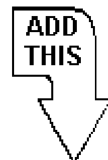
```
procedure TForm2.FormActivate(Sender: TObject);
var
  textline:string;
begin
  memo1.clear;
  if form1.adult>0 then
  begin
    textline:=inttostr(form1.adult)+ ' adults';
    memo1.lines.add(textline);
  end;
  if form1.child>0 then
  begin
    textline:=inttostr(form1.child)+ ' children';
    memo1.lines.add(textline);
  end;
  if form1.firstclass=true then
    memo1.lines.add('First class ')
  else
    memo1.lines.add('Second class ');
  if form1.return=true then
    memo1.lines.add('Return')
  else
    memo1.lines.add('Single');
end;
```



Compile and test the program using a variety of ticket data.

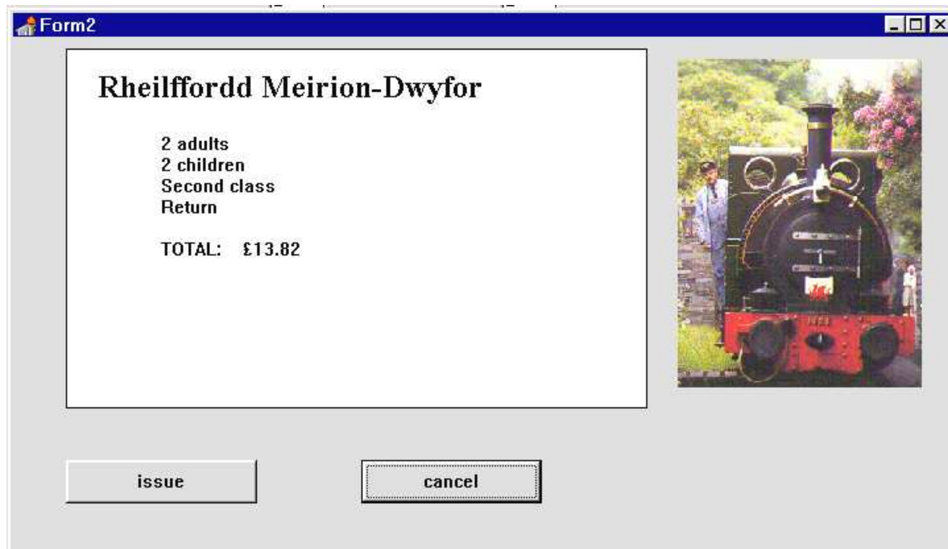
The last step is to display the ticket total which was calculated earlier on Form1. Do this by adding further lines at the end of the **FormActivate** procedure:

```
          ........
          ........
  if form1.return=true then
    memo1.lines.add('Return')
  else
    memo1.lines.add('Single');
  memo1.lines.add('');
  textline:='TOTAL:    £';
  textline:=textline +
        floattostrf(form1.ticketcost,fffixed,8,2);
  memo1.lines.add(textline);
end;
```



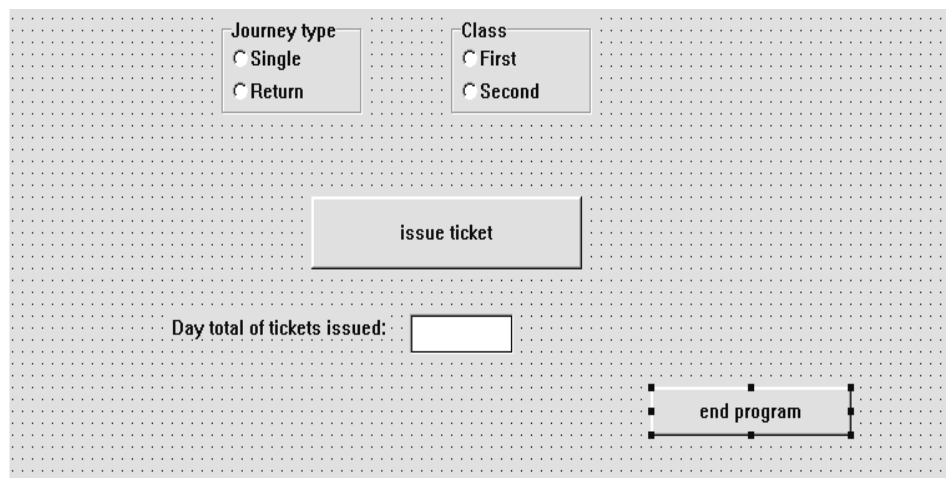
Compile and run the program to test the ticket display, then return to the Delphi editing screen. Bring the Form2 window to the front.

The display will look neater if the frame around the memo box is removed. To do this, click on the memo box and press ENTER to bring up the Object Inspector. Set the **BorderStyle** property to '**None**', and the '**Ctl3D**' property to '**false**' :



Display of tickets is now completed. The last stage of the program is to keep a total of the tickets issued.

Go to the Form1 grid and add a label alongside the edit box. Also add another button and give this the caption 'end program':



Create an event handler for the 'end program' button:

```
procedure TForm1.Button3Click(Sender: TObject);
begin
  halt;
end;
```

Now double-click on the Form1 grid to set up an event handler procedure called **FormCreate.** This procedure will operate only once - at the time when the program first starts. We can use it to initialise the day total to zero. Add the line:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  daytotal:=0;
end;
```

'Daytotal' needs to be added to the variable list:

```
  { Public declarations }
  adult,child:integer;
  ticketcost,daytotal:real;
  firstclass,return:boolean;
end;
```

Go now to Form2 and double-click the'issue' button to produce an event handler and add the lines:

```
procedure TForm2.Button1Click(Sender: TObject);
begin
  form1.daytotal:=form1.daytotal+form1.ticketcost;
  form1.edit3.text:=
           floattostrf(form1.daytotal,ffFixed,8,2);
  form2.visible:=false;
end;
```

The purpose of this procedure is to:

- add the current ticket price to the day total
- display the day total in the edit box on Form1
- close the window for the ticket

Compile and try out the finished program.

SUMMARY

In this chapter you have:

- Examined three methods used in program design: a *program schematic diagram*, *algorithm progressive refinement sequences* and a *flowchart*
- used *radio group* components
- used Boolean variables
- written sections of program containing **if..then..else** conditions
- used a *memo* component to display lines of text
- transferred values of variables between the different Forms in the project
- produced a *FormActivate* event handler, which operates each time the form is opened
- produced a *FormCreate* event handler, which operates only once at the start of the program.