Unit 14: Event driven programming

By the end of this unit you should:

1. Understand the features of event driven programming
2. Be able to use the tools and techniques of an event driven language
3. Be able to design event driven applications
4. Be able to implement event driven applications

Whether you are in school or college, passing this unit will involve being assessed. As with most BTEC schemes, the successful completion of various assessment criteria demonstrates your evidence of learning and the skills you have developed.

This unit has a mixture of pass, merit and distinction criteria. Generally you will find that merit and distinction criteria require a little more thought and evaluation before they can be completed.

The colour-coded grid below shows you the pass, merit and distinction criteria for this unit.

To achieve a pass grade you need to:	To achieve a merit grade you also need to:	To achieve a distinction grade you also need to:
P1 Explain the key features of event driven programs	**M1** Discuss how an operating system can be viewed as an event driven application	**D1** Evaluate the suitability of event driven programs for non-graphical applications
P2 Demonstrate the use of event driven tools and techniques	**M2** Give reasons for the tools and techniques used in the production of an event driven application	
P3 Design an event driven application to meet defined requirements		
P4 Implement a working event driven application to meet defined requirements		
P5 Test an event driven application	**M3** Analyse actual test results against expected results to identify discrepancies	**D2** Evaluate an event driven application
P6 Create on-screen help to assist the users of a computer program	**M4** Create technical documentation for the support and maintenance of a computer program	

Introduction

Event driven programming (EDP) is a 10-credit unit that explores concepts typically introduced in Unit 6 Software design and development.

Programs developed using an event driven approach differ from traditional algorithm-oriented solutions. In EDP, the key is to understand what kinds of events can be triggered and how the program can best respond to them.

For some learners, EDP can be much more rewarding as it allows them to build simple applications that offer rich functionality with minimum fuss.

In this unit, we will explore EDP using Microsoft's popular Visual Basic .NET® programming language.

How to read this chapter

This chapter is organised to match the content of the BTEC unit it represents. The following diagram shows the grading criteria that relate to each learning outcome.

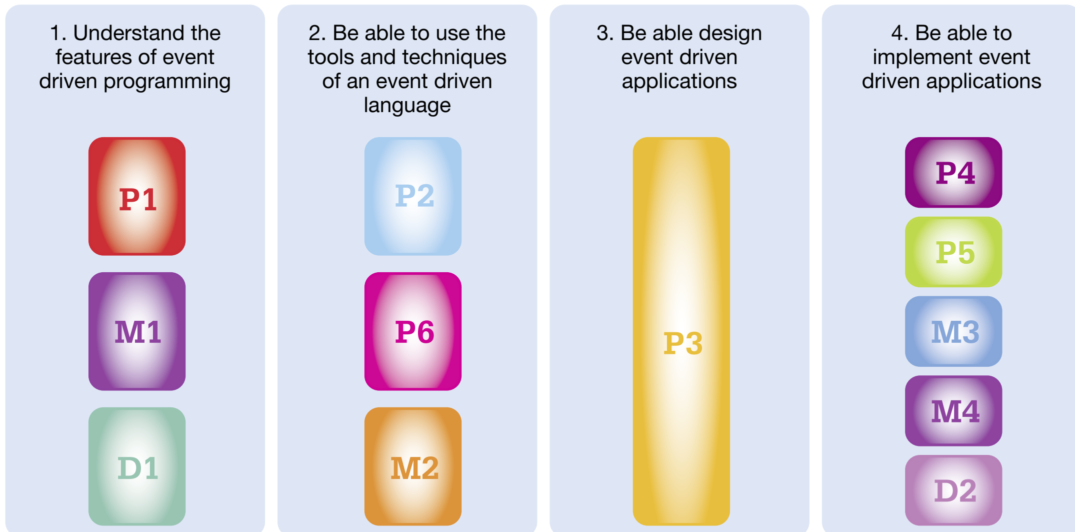


Figure 14.00

You'll find colour-matching notes in each chapter about completing each grading criterion.

14.1 Understand the features of event driven programming

This section will cover the following grading criterion:

P1

Make the Grade — P1

This criterion requires you to be able to **explain** the **key features** of event driven programs.

This is designed to test your knowledge and understanding of the various features that typify EDP.

Assessment is likely through a quiz, discussion, leaflet, poster, report or presentation.

14.1.1 Key features

Event driven programming can contribute to both service oriented and time driven approaches to processing. What *does* this mean?

Service oriented processing is a term used to describe the concept of multiple services (processes written in different programming languages) advertised and accessed as and when needed by an organisation.

Time driven is the term used to describe processing that occurs when tasks are regulated by a clock, that is, actions are occurring at set intervals and typically need a real-time response. In this case, an event could be generated by a 'timer' trigger.

In addition there are a number of other terms that are used when discussing event driven programming. The most common ones are shown in the Key terms box.

Figure 14.01 gives a visual representation of the key EDP features.

Key terms

A **form** is a visual container used to group together user interface components such as text boxes, buttons, labels, checkboxes and so on. It is used to provide an input mechanism for a user that is both approachable and functional.

Event loops are processing cycles that continually look for events to happen (e.g. a button click, file deletion or arrival of a data packet over a network).

Trigger functions are used by event loops to identify and launch a response to an event that has happened in an event loop.

Event handlers are the actual program code modules that are executed when a particular trigger has occurred. For example, if a user clicked a button this would trigger an event handler for the code actions associated with the button.

Event driven programming uses all four of these elements to form an effective **software development solution**.

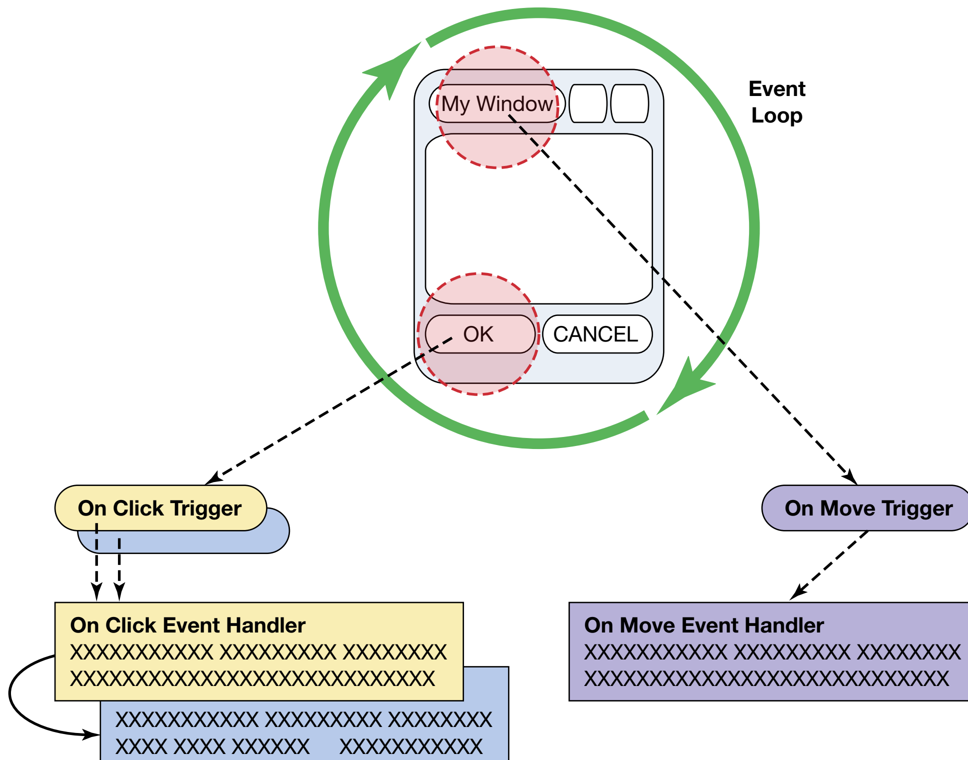


Figure 14.01 Event driven programming

EDP Scorecard

There are clear advantages to using EDP:

+ Flexibility.
+ Suitability for graphical user interfaces.
+ Simplicity of programming (visual components auto generate code for the programmer).
+ Ease of development, particularly for **R**apid **A**pplication **D**evelopment (**RAD**).

In the example shown in Figure 14.01, a form is enclosed in an event loop.

Two trigger functions are specifically available: Form Move and Button Click.

When the 'Form Move' trigger is detected (by the user manually repositioning the form on screen), the 'On Move' event handler is called and executed.

When the 'Button Click' trigger is detected in response to the 'OK' button being clicked, an 'On Click' event handler is called and executed.

Please note that the 'On-Click' event handler could perform an action that would trigger another event to occur. This is called an **event cascade**; in theory a number of events could be cascaded in an event driven solution.

14.1.2 Examples

This section will cover the following grading criteria:

Make the Grade M1 D1

M1 requires you to be able to discuss how an operating system can be seen as an event driven program.

This is designed to test your knowledge and understanding of the various features that typify EDP and be able to equate those to the GUI functionality that is present in modern operating systems.

Assessment is likely through a quiz, discussion, leaflet, poster, report, presentation or screencast.

D1 requires you to be able to evaluate the suitability of event driven programs for non-graphical applications.

In terms of non-graphical user interfaces, a typical command line interface (see Unit 2) normally processes just keyboard input. The scope for varied input mechanisms is therefore very limited.

An EDP application could be written to process system events, programmed to listen for certain events, for example, a system error occurring, informing the user or running a specific utility to fix the problem automatically.

An operating system is a good example of an event driven system as its graphical user interface (GUI) has to accommodate users **freely interacting** with many different on-screen components **in any order**.

In addition, the aspect of the operating system that controls and reports on hardware usage and problems also reacts to system events in order to inform the user.

Some typical operating system occurrences that show examples of EDP include:

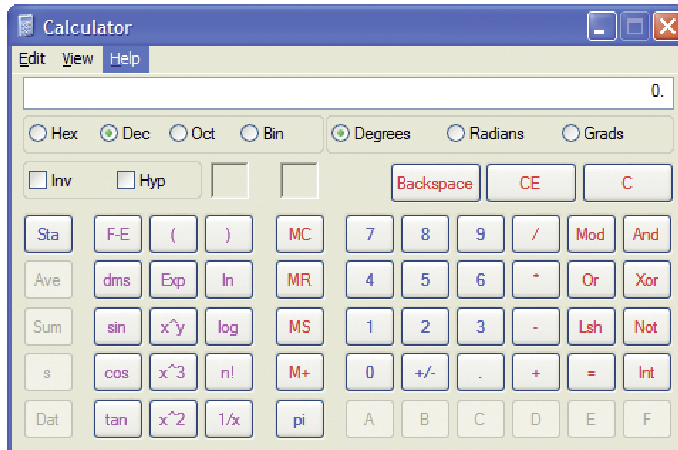


Figure 14.02 Microsoft Windows® calculator applet

1. Events generated by the user via the GUI, such as
   - clicking an icon
   - moving an icon
   - dragging a window
   - resizing a window
   - dragging and dropping a selected item
   - clicking on a menu item
   - minimising or maximising a window.
2. Events generated by the operating system. For a specific example, let us use a Microsoft Windows® **calculator applet**, which has a surprising number of potential **triggers** (Figure 14.02).

This applet has to deal with triggers for button clicks, key presses, menu selection, radio buttons, check boxes, minimising and window movement – and that's before the functional part (i.e. doing the actual calculations) is considered.

As you can see, any simple program that awaits unpredictable user input has to be ready to handle any event, depending on the trigger.

14.1.3 Programming languages

There are a number of programming languages which adopt the **event driven paradigm**.

The most common ones are detailed below.

Microsoft Visual Basic®

Visual Basic®, often abbreviated by developers to just 'VB', is an event driven programming language that was created by Microsoft and released in 1991.

The language itself is heavily derived from **BASIC** (**B**eginner's **A**ll-purpose **S**ymbolic **I**nstruction **C**ode) and shares many of its **keywords** and **constructs**. In addition, Visual Basic® provides opportunities for RAD applications, which often rely on a graphical user interface (GUI).

It also has simple integration with Microsoft Access® and Excel® and provides relatively straightforward use of Window libraries (in the form of .DLL files and ActiveX® controls) to tap into pre-written routines such as the Common Dialog Box for Open/Save/Font/Colour components.

Visual Basic® programs are only intended to run on a Microsoft Windows® operating system. The Visual Basic® language and **Integrated Development Environment (IDE)** have the same limitation.

Visual Basic® is slowly being replaced by its successor (**Visual Basic.NET®**) although, as there is a wealth of legacy, Visual Basic® code is still in use today and is unlikely to disappear that soon.

The last version of traditional Visual Basic® is VB6 (Figure 14.03).

Its successor is **Microsoft Visual Basic.NET®**.

Microsoft Visual Basic.NET®

Visual Basic.NET® is effectively Microsoft's version 7 of Visual Basic®. However, more than being a simple update, VB.NET® is a true **object oriented version** of the language, built on Microsoft's **.NET framework**. As a consequence, the language syntax and features available in Visual Basic® no longer worked as they once did, causing developers much pain during their initial period of transition.

The newest version is part of **Microsoft Visual Studio® 2010 Professional** (Figure 14.04).

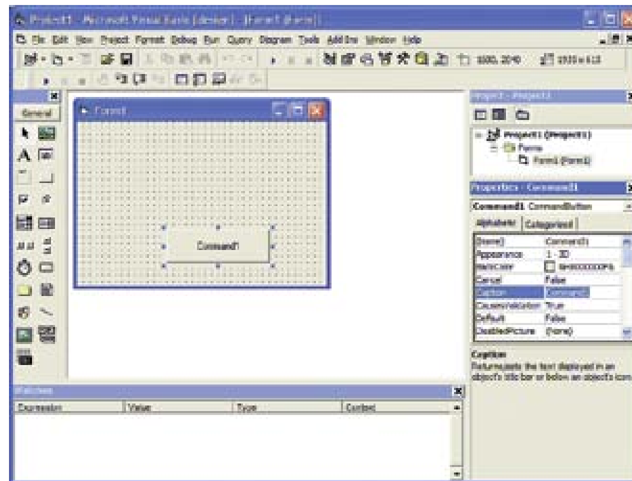


Figure 14.03 Visual Basic® 6 IDE

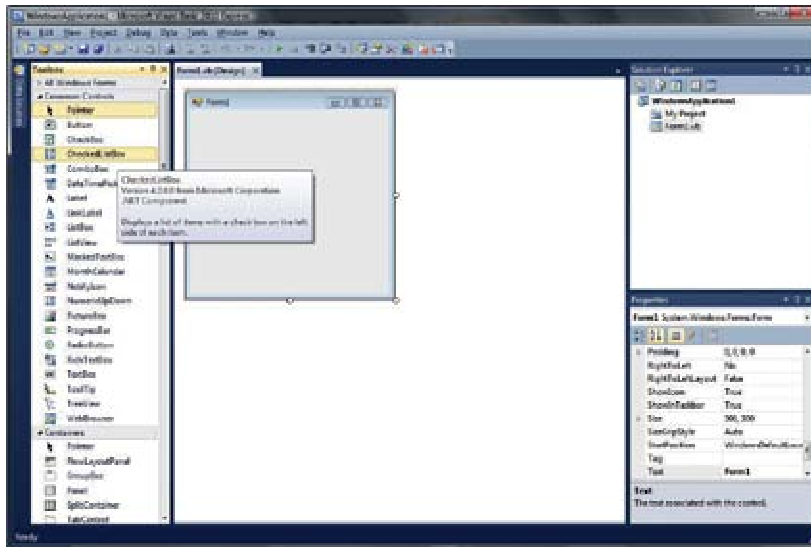


Figure 14.04 Visual Basic.NET® IDE

Activity 1

Downloading Visual Basic.Net® 2010 Express Edition

Following the provided link, download, install and register the free version of Microsoft's Visual Basic.Net® 2010 Express Edition.

Visual Basic.NET® Express edition can be freely downloaded from:

www.microsoft.com/expressDownloads/#2010-Visual-Basic

VBA

VBA (**V**isual **B**asic for **A**pplications) is a dialect of Microsoft's Visual Basic® that is built into Microsoft Office® applications such as Microsoft Word® and Microsoft Excel®.

VBA gives developers a way to add functionality to Microsoft applications and to customise them to form tailored solutions. In addition, VBA also allows developers to automate these packages using macros which represent a quick, low-cost, low-risk approach to designing bespoke solutions.

```
Sub Test()
' Test Macro
' Macro recorded 01/03/2010 by Mark
'    Options.DefaultHighlightColorIndex = wdYellow
    Selection.Range.HighlightColorIndex = wdYellow
End Sub
```

This example, accessed from a **toolbar button click event**, highlights the selected text in yellow.

VBA has a poor reputation for security; many **macro viruses** have been successfully written which infect Microsoft Office® documents, often causing annoying side effects and disabling an application's key features. As a result, security features have been added to these applications to screen malicious ('unsafe') VBA code.

VBA has been superseded by Microsoft Visual Studio® 2005 Tools for Applications, a .NET implementation.

Adobe ColdFusion® (CF)

ColdFusion® is a RAD environment, running on multiple operating system platforms, that can create rich internet applications that often contain a large amount of event driven components.

ColdFusion® uses its own proprietary scripting language (**C**old**F**usion® **M**arkup **L**anguage or '**CFML**') to create interactive and dynamic web pages in a similar way to PHP and ASP .Net®. This requires the use of a ColdFusion® Application Server, which processes the script before sending it to the receiving web browser client for rendering.

ColdFusion®'s ability to expand on traditional HTML limitations enhanced its ability to create web pages with a number of interactive and event driven elements.

Key terms

An **API** or **Application Programming Interface** forms the library or interface that gives an application access to common operating system functions.

DirectX is a form of API designed by Microsoft that is used specifically in the creation of Windows®-based multimedia applications. It includes support for 2D and 3D graphics, network communication, input devices, music and sound.

14.2 Be able to use the tools and techniques of an event driven language

This section will cover the following grading criteria:

P2 M2

Activity 2

Starting with Adobe® ColdFusion®

Following the provided link, download, install and try the trial version of Adobe®'s ColdFusion® 9.

www.adobe.com/eeurope/products/coldfusion/

In addition, the following websites have good CF tutorials, live demonstrations and code downloads:

LearnCF learncf.com/home

EasyCFM www.easycfm.com/

Make the Grade P2 M2

P2 asks you to be able to **demonstrate** event driven tools and techniques; these are listed throughout sections 14.2.1, 14.2.2 and 14.2.3. Any event driven solution you design and implement for P3 and P4 is likely to provide evidence for your selection of these tools and techniques, for example, your choice of variables, use of selections and loops, the controls used on your forms.

For M2, however, you need to explain why you used certain tools in your solution, for example, why use a drop-box rather than a textbox? Why use radio buttons rather than a checkbox? Why is the menu system designed like that?

14.2.1 Triggers

Common triggers are typically either: a **user event** or a **system event**.

It is important that you can use an event driven programming language to respond to such event triggers. The following Visual Basic® .NET example in Figure 14.05 demonstrates handling both user events (key presses, mouse click etc.) and system generated events.

This simple example is created by selecting a 'Button Control' (actually a .NET component) from the VB .NET® Toolbox. This is then drawn on the form.

The Button Control's Text property is changed to 'What time is it?'

The Button is then double-clicked to display the VB .NET® code window.

It is here that an empty 'stub' event handler is shown. As a software developer, you need to add the appropriate actions to this event handler.

In this case we will use a VB .NET® function called 'MsgBox' which displays customised Message Boxes on the Windows® screen (Figure 14.06). In addition, we'll use the 'Now' function which returns the current date and time (as stored in the PC's **RTC** – **r**eal **t**ime **c**lock).

In the nice, simple example in Figure 14.07, the VB .NET® application simply executes the 'Button_Click' event handler when the user triggers the event by clicking on the 'What time is it?' button with the mouse.

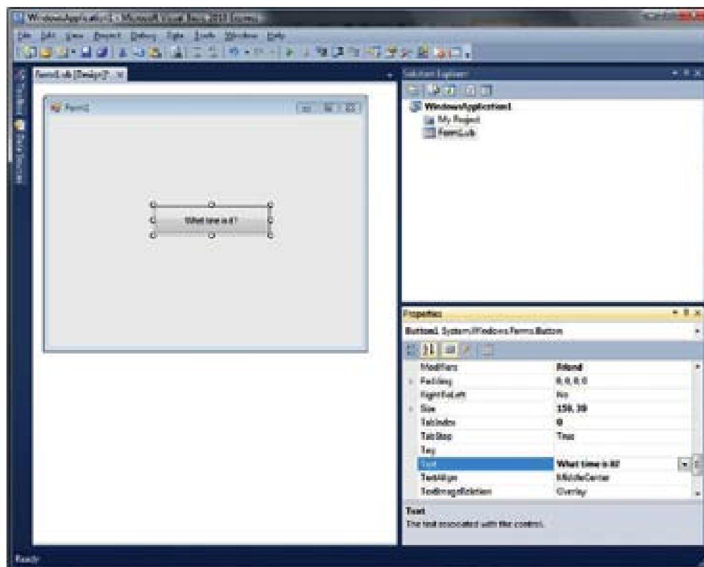


Figure 14.05 Creating a simple user event in VB.NET®

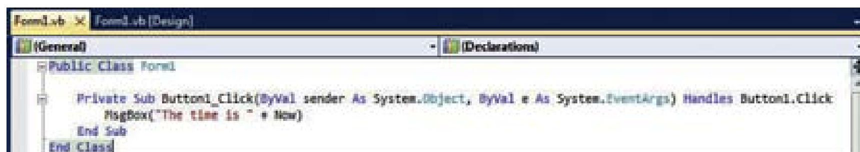


Figure 14.06 Coding the event handler in VB.NET®

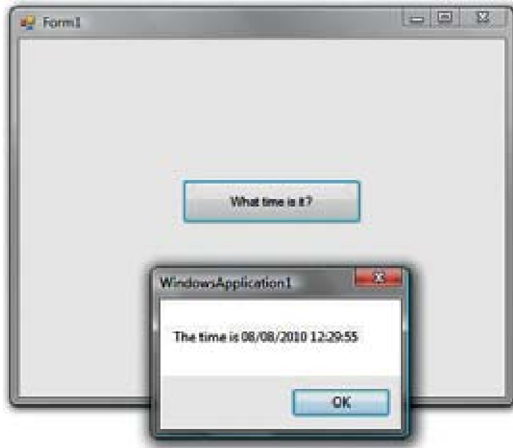


Figure 14.07 Running the VB.NET® application and triggering the user event

The 'Button_Click' event handler responds to the trigger by displaying a message box that shows the current date and time; displaying the 'OK' button is the default behaviour for a message box.

As you can see, coding a basic event handler is a fairly straightforward practice in VB.NET®. Don't worry, though, the examples will get more challenging as we work through.

Activity 3

Modifying the code

VB.NET® programs are modified by altering the form contents and its associated event handlers.

Try to modify the Button so that the Text property says 'Play Windows Alert!'

Delete the Msgbox line of code and replace with the following:

```
My.Computer.Audio.
PlaySystemSound(System.Media.
SystemSounds.Asterisk)
```

Now re-run the VB.NET® program and test the new user event.

Hopefully, if you have modified the code correctly, your PC should make your Windows® Alert sound when the mouse's 'Button_Click' event is triggered.

System events are a bit trickier!

First we need to select a suitable system event.

System events are directly connected to the operating system on a computer system. As a result, any demonstration created using VB.NET® will be using Microsoft Windows®.

Common Windows® system events include triggers for:

- low RAM
- change of display settings (the video mode)
- change of power mode (e.g. moving to and from suspend mode)
- change of RTC settings (date and time).

Caution!

Because of the complexity of the following example, the process has been broken down into a number of steps.

If you follow each step carefully, everything should work. If you discover a problem, go back a step (or two) and see whether you've missed anything.

Step 1

Create a new project, adding the two labelled buttons (Button1 and Button2) and a label (Label1) shown in Figure 14.08.

Figure 14.08 Creating the form for system events

Step 2

Double click on Button1 and add the following VB.NET® code:

```
AddHandler SystemEvents.
TimeChanged, AddressOf
MyEventHandler
```

Step 3

Double click on Button2 and add the following VB.NET® code:

```
RemoveHandler SystemEvents.
TimeChanged, AddressOf
MyEventHandler Label1.Text = " "
```

This will have added code to both buttons' event handlers to add a system event handler (when Button1 is triggered) and remove the system event handler (when Button2 is triggered).

The handler is for the TimeChanged system event. VB.NET® will need to know what we want to do when the system event is triggered (i.e. when the user changes the RTC). To do this we simply give the RAM address of a new event handler we are going to write (we've called this 'MyEventHandler').

Creating the new event handler comes next.

Step 4

Right click on 'WindowsApplication1' in VB.NET®'s Solution Explorer panel (Figure 14.09).

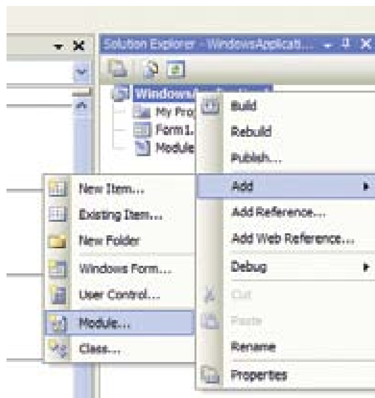


Figure 14.09 Adding a new module to the current application

VB.NET® uses modules as separate .vb files which can store our functions. This new module is where we will put our new program code for the TimeChanged system event handler (i.e. saying what will happen).

Step 5

Select 'Module' which opens up the 'Add New Item' window (Figure 14.10).

Choose 'Module' and ensure that the module is called 'Module1.vb' (it should be).

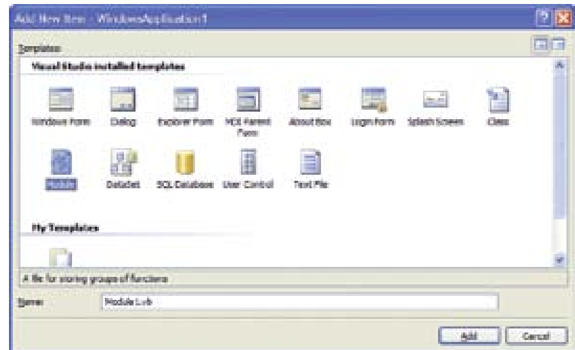


Figure 14.10 Adding a new module to the current application

Step 6

Double click the new 'Module1.vb' entry in the Solution Explorer.

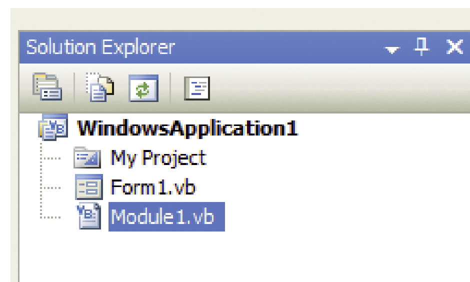


Figure 14.11 Opening the new module

Then add the following VB.NET® code into the empty module:

```
Public Sub MyEventHandler(ByVal
sender As Object, ByVal e As
EventArgs)
    Form1.Label1.Text = "User has
changed the system time"
End Sub
```

Step 7

Double click 'Form1.vb' in the Solution Explorer.

Then, double click the form itself to create the empty 'Form1_Load' event handler.

Add this code to it:

```
Label1.Text = " "
```

Step 8

Move the cursor into the 'General' section of the VB.NET® code window.

Add the following line:

```
Imports Microsoft.Win32
```

Step 9

That should be it!

Let's recap those steps by first looking at the form's ('Form1.vb') code (Figure 14.12).

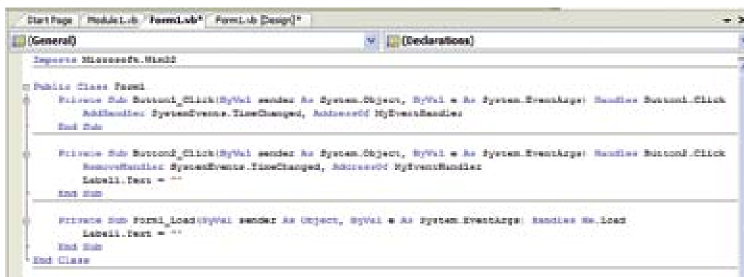


Figure 14.12 VB .NET® code related to the form

Step 10

And then the same, but for the module ('Module1.vb').

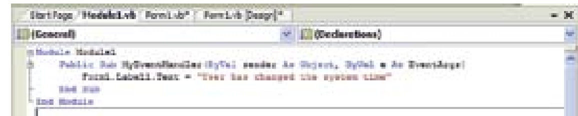


Figure 14.13 VB .NET® code related to the module

Step 11

Save all the files (CTRL+SHIFT+S)!

Step 12

It should now be possible to run the application to test our handling of Windows®' 'TimeChanged' trigger.

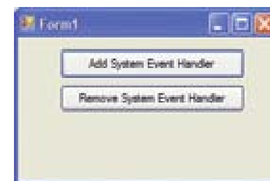


Figure 14.14 The form as it first appears

Step 13

Enable our system event handler by clicking the 'Add System Event Handler' button once (Figure 14.14).

Step 14

Change your PC's system time (Figure 14.15).

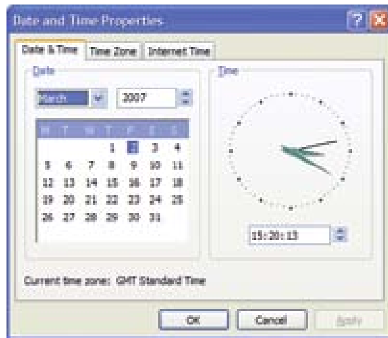


Figure 14.15 Changing the time in Windows®' 'Date and Time' applet

Step 15

Re-examine our form (Figure 14.16).

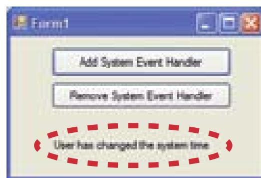


Figure 14.16 The application has changed appearance

As you can hopefully see, our homemade system event handler has been triggered by the operating system. This is shown by the change of the label's text (beneath the buttons). Compare this to Figure 14.14.

Step 16

Click on 'Remove System Event Handler' and change the time back.

Although the 'TimeChanged' trigger will still occur, our system event handler has been removed. As a result, there will be nothing to deal with it.

Activity 4

Modifying the code

Attempt to modify the example application by writing a simple event handler for the 'change of display settings' trigger.

This is known as 'DisplaySettingsChanged'.

As you may have guessed, the '`Imports Microsoft.Win32`' is used to give the developer access to the system events listed in the Win32 API.

Your homemade event handler can simply display a message that says 'Display settings have been changed'.

If you feel really confident, add a Windows® alert sound.

Braincheck 1

1. Name three different event driven programming languages.
2. Common triggers can be caused by the ______ or the ________? Complete this sentence.
3. Name three possible system events.
4. Give three advantages of event driven programming languages.
5. Give three disadvantages of event driven programming languages.

How well did you do? See the answers section!

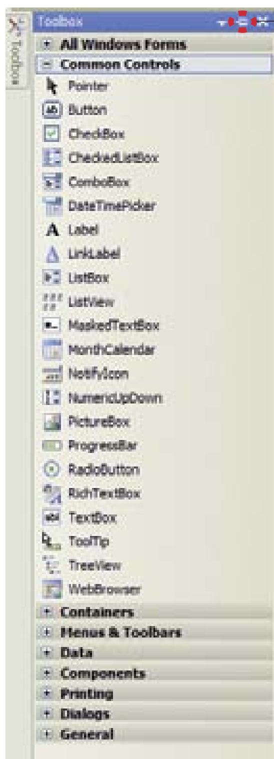
14.2.2 Tools and techniques

An event driven programming language such as VB.NET® has a number of different elements. In this section we'll take a closer look at these elements and look at putting them together to form more complex solutions.

Use of toolbox and controls

Controls are the **basic elements** of a VB.NET® application.

Most of the common controls can be found in the **Toolbox**, a categorised collection of visual elements that can be added to a standard form (Figure 14.17). It is accessed by pressing the button to the left of the form design view.



Clicking the **Auto Hide** icon will **toggle** (stop) the toolbox between **automatically hiding** when not in use and being **permanently pinned** to the **left hand side** of the screen.

Figure 14.17 VB.NET® Toolbox with Common Controls expanded

The Common Controls represent the category that most developers will initially rely on to build their solutions as they form the core of most graphical user interfaces. It is also the category that we will focus on in this book.

When a control is added to a form, it allows the developer to add event handler code to specific triggers and change the control's properties (Figure 14.18).

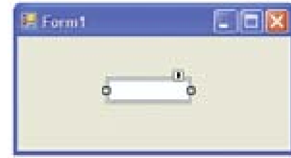


Figure 14.18 VB.NET® TextBox control

In this regard it is similar to an object, having both properties and methods.

> **Unit link**
> Unit 6 – Software design and development section 6.2.2 introduces object oriented programming concepts.

The control's properties can be seen in VB.NET®'s Properties Window (Figure 14.19).

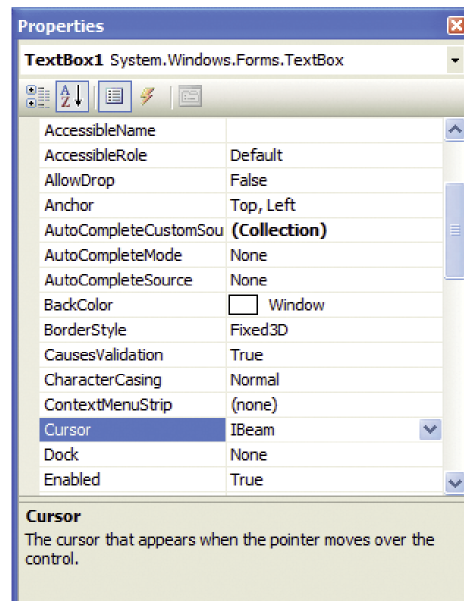


Figure 14.19 VB.NET® TextBox control's properties (alphabetical)

VB.NET® controls can therefore be seen as **classes**, with the actual **instances** of the controls (sat on the form) as the true **objects**.

This is a logical view as changing the **properties** of one TextBox 'object' **does not** change the properties of another (much as in OOP).

As you have seen already, (most but not all) properties can either be changed at **design time** or at **run time** (through the VB.NET® program code itself).

For example, we can use the Properties Window and alter the correct property manually (Figure 14.20).

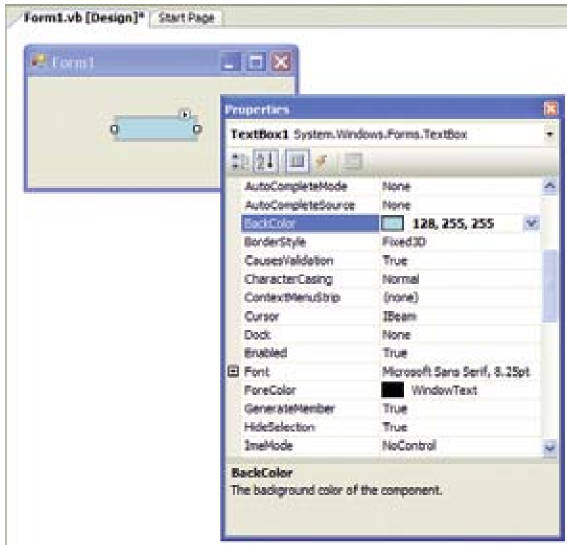


Figure 14.20 Changing the TextBox object's BackColor property to cyan at design time

But we can also achieve the same effect via VB.NET® code, executed at **run time**:

```
TextBox1.BackColor = Color.Cyan
```

VB has a number of pre-defined colours in its Color structure; 'Cyan' is just one of them.

A more complex method would be…

```
TextBox1.BackColor = System.
Drawing.Color.FromArgb(128, 255,
255)
```

The latter example uses the **FromArgb function** (or **method**) to generate a colour based on its **red**, **green** and **blue** values. If you look back at Figure 14.20, you'll see that these are the **same** RGB values as selected in the Properties Window.

Event handlers

VB.NET® has a number of event handlers, some are generic ('Click' being fairly common) but others are specific to certain controls.

Here is a list of some event handlers for the Button control:

- **Click** occurs when user triggers by clicking the button.
- **DoubleClick** occurs when user triggers by double-clicking the button.
- **KeyDown** occurs when button has focus (it's selected) and a key is pressed.
- **MouseHover** occurs when user hovers mouse over the button.

And some different event handlers for a TextBox control:

- **Enter** Occurs when user triggers by moving into the TextBox.
- **TextChanged** Occurs when user triggers by altering the text inside the TextBox.
- **Leave** Occurs when user triggers by moving away from the TextBox (i.e. clicks or tabs to another control).

The full list of event handlers available for a control can be found in VB.NET®'s help system.

Selection

A **selection** is a type of **conditional statement**, allowing the developer to make a choice about which lines of code to execute.

VB.NET® has a number of selection mechanisms but perhaps the most frequently used are 'If…then…else' and 'Select…Case' statements.

Let's take a look at the '**If…then…else**' statement first.

```
Private Sub TextBox1_Leave(ByVal sender As Object, ByVal e As System.
EventArgs) Handles TextBox1.Leave

  If Not IsDate(TextBox1.Text) Then
    MsgBox("Please enter date (dd/mm/yy)", MsgBoxStyle.Information)
    TextBox1.Focus()
  Else
    MsgBox("Date acceptable", MsgBoxStyle.Information)
  End If
End Sub
```

If…then…else

Here is a practical example of using an 'If…then…else' statement.

In this example a sample form (as shown in Figure 14.21) has two TextBoxes.

The first TextBox control (TextBox1) will allow the user to store a date in the format dd/mm/yy, for example, 04/09/07 would be entered for 4 September 2007.

The second TextBox control (TextBox2) will allow the user to store a time.

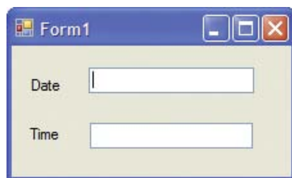


Figure 14.21 The sample form

If an invalid date is entered (tested using the IsDate function in VB.NET®), the first part of the 'If…then….else' statement displays a message box and the user is returned to the Date TextBox (TextBox1) for another try (Figure 14.22).

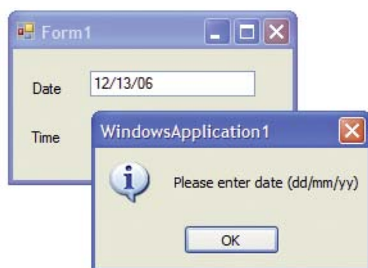


Figure 14.22 Getting the input wrong – an example of data validation

The test for a valid date is performed when the user moves away from the Date TextBox, that is, when they click on the Time TextBox. This is performed by the 'Leave' event handler.

However, if a valid date is entered, the 'else' part of the 'If…then…else' is executed. All this does is to simply pop up another message box, but this time it displays a confirmation that the date was acceptable.

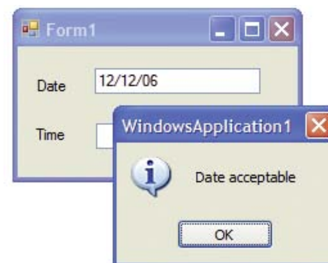


Figure 14.23 Getting the input right

Select…Case statement

Case statements are often used to simplify program code when it is necessary to match a single value (or range of values) from a list of given possibilities.

Here is a practical example of the 'Select…Case' statement at work.

Case Study

Lee Office Supplies is currently running a promotional offer on buying multiple packs of A4 laser printer paper.

Customers who buy two to five packs of 500 sheets will get a 10 per cent discount.

Customers who buy six to eight packs of 500 sheets will get a 20 per cent discount.

No customer may order more than eight packs at any one time.

First we create a simple form with Label, TextBox and Button controls:

Figure 14.24 Lee Office Supplies' quantity check

We then add the following code to the Click event of the 'What's my discount?' button control:

```
Select Case (Val(TextBox1.Text))
  Case 1
    MsgBox("Sorry, no discount!", MsgBoxStyle.Information)

  Case 2 To 5
    MsgBox("10% discount.", MsgBoxStyle.Information)

  Case 6, 7, 8
    MsgBox("20% discount.", MsgBoxStyle.Exclamation)

  Case Else
    MsgBox("Sorry, you can't order more than 8.", MsgBoxStyle.Critical)
    TextBox1.Focus()

End Select
```

If you examine this closely, you will see that the text value entered into the 'Quantity Required' TextBox is converted to a number (with the VB.NET® **Val function**). This is then checked by the 'Select…Case' statement against a list of possible values (and ranges).

Note that the case statements may check a single value, a range (e.g. '2 To 5') or a comma separated list. The else part of the 'Select…Case' statement is used to process any other value that isn't matched by any previous case statement.

Let's test this code with some sample input values that a customer might enter.

Entering a quantity of 1:

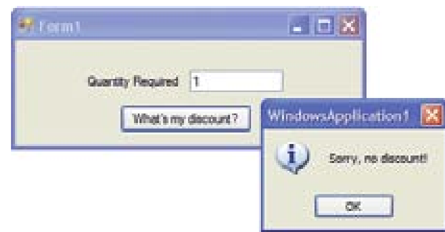


Figure 14.25 No discount awarded

Entering a quantity of 3:

Figure 14.26 10% discount awarded

Entering a quantity of 7:

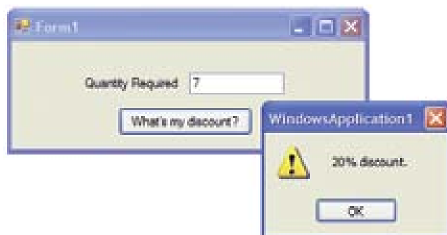


Figure 14.27 20% discount awarded

And finally, entering a quantity of 10:

Figure 14.28 Invalid quantity requested

In most EDP languages, similar types of selections exist; they are fundamental building blocks that let the developer **make choices** in their programs.

Loops

Loops or **iterations** allow the developer to perform a group of actions repeatedly. Unless the loop is infinite (goes on endlessly), it will have some kind of conditional statement that will force it to stop.

VB.NET® has a number of different loops. One of the most commonly used is the **'For…Next' statement**.

The 'For…Next' statement is a common tool in most programming languages; VB.NET® is no exception to this rule. It is used to run a loop a preset number of times, usually controlled by a counter.

Figure 14.29 demonstrates the use of a 'For…Next' loop to generate a child's times table based on two inputs (the table number itself and the number of rows required).

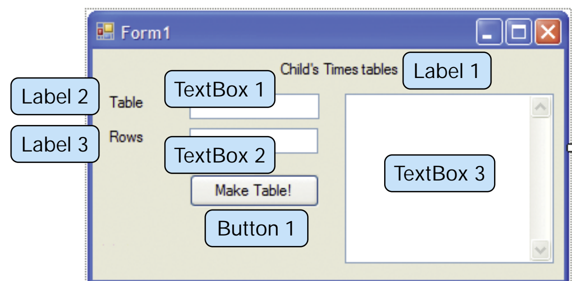


Figure 14.29 Form design for a child's times tables

TextBox3 will be treated slightly differently. Modify the following properties:

Multiline set to **True**

ScrollBars set to **Both**

Next, add the following code to the click event of the 'Make Table!' button control:

Unit link

In Unit 6 – Software design and development, section 6.1.4 also examines selections (or conditional statements) but does so from a C#® perspective. You'll notice that although the C#® syntax is different, 'If' and 'Case' statements are similarly constructed in different languages.

```
Dim counter As Byte
Dim newline As String
TextBox3.Text = TextBox1.Text & " times table" & vbCrLf
For counter = 0 To Val(TextBox2.Text)
   newline = counter & " X " & TextBox1.Text & " = " & _
   Val(TextBox1.Text) * counter & vbCrLf
   TextBox3.AppendText(newline)
Next counter
```

Pre-check condition

Next, run the application and some sample values:

Figure 14.30 The four times table

In this example we've asked for ten rows of the four times table.

If you take a look back at the code, you will see that the 'For…Next' loop is instructed to run from 0 up to the value entered into TextBox2 (the number of rows). If we enter '10' into that TextBox, the 'For' loop will indeed run from 0 to 10 (rows 8, 9 and 10 need to be scrolled down to in Figure 14.30).

Inside the 'For…Next' loop a new string (see section 14.2.3) is created which is used to assemble a line of output that includes the counter, 'X' and '=' symbols, the table number and the calculated result (which is also stored in a new variable (again, see section 14.2.3 for details). This is then appended into the multiline TextBox3.

It is because of the fixed-length nature of the 'For…Next' loop that it is often used to repeat code where the developer knows how many times it is to be repeated **before it starts**.

The second type of loop we will look at is the **Do… loop**.

In VB.NET® the 'Do…' loop can either be pre- or post-check conditioned. What this means is that the controlling condition is put either before (pre-check) or after (post-check) the lines of code being repeated. The following example (a vowel counter) is written twice, using two forms of the 'Do…' loop.

First, create the form as indicated:

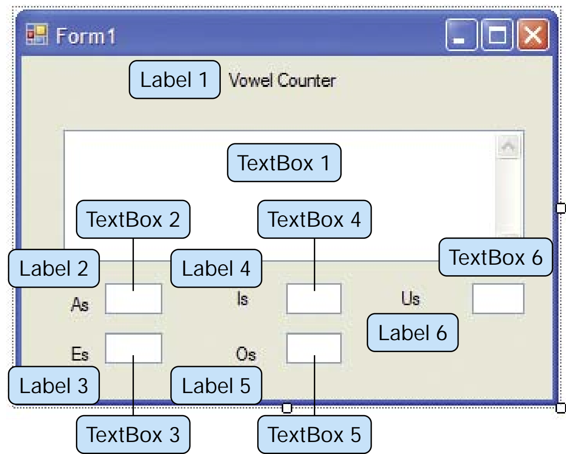


Figure 14.30 The vowel counter form design

Again, in preparation the following properties must be set on the TextBox1 control:

Multiline set to **True**

ScrollBars set to **Both**

Unit link

Unit 6 – Software design and development, section 6.1.4 also examines loops but does so from a C#® perspective. You'll notice that although the C#® syntax is different, loops tend to be similarly constructed in different languages.

```
Dim counter As Byte
Dim nextchar As Char
Dim acount As Byte
Dim ecount As Byte
Dim icount As Byte
Dim ocount As Byte
Dim ucount As Byte
counter = 1
Do
   nextchar = Mid(TextBox1.Text, counter, 1)
   Select Case (nextchar)
      Case "a", "A"
         acount = acount + 1
         TextBox2.Text = acount
      Case "e", "E"
         ecount = ecount + 1
         TextBox3.Text = ecount
      Case "i", "I"
         icount = icount + 1
         TextBox4.Text = icount
      Case "o", "O"
         ocount = ocount + 1
         TextBox5.Text = ocount
      Case "u", "U"
         ucount = ucount + 1
         TextBox6.Text = ucount
  End Select
   counter = counter + 1
Loop Until counter > TextBox1.TextLength
```

Post-check condition

The code above is added to the TextChanged event handler of TextBox1:

When this application runs, the event handler is triggered when a new keystroke is made. The application then runs a post-check conditioned 'Do…' loop, which repeats until the counter is greater than the length of the current text stored in TextBox1. Inside the loop, a 'Select…Case' is used to examine **each character** (by using the **MID function**) and increment the appropriate counter when a vowel is matched (Figure 14.32).

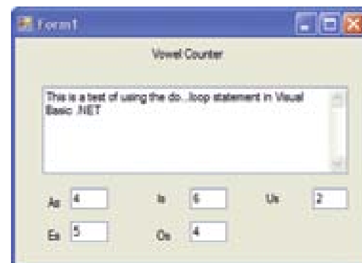


Figure 14.32 The vowel counter application running

The same code can be re-written as:

```
Dim counter As Byte
Dim nextchar As Char
Dim acount As Byte
Dim ecount As Byte
Dim icount As Byte
Dim ocount As Byte
Dim ucount As Byte
counter = 1
Do While counter <= TextBox1.
TextLength
  nextchar = Mid(TextBox1.Text,
counter, 1)
  Select Case (nextchar)
    Case "a", "A"
      acount = acount + 1
      TextBox2.Text = acount
    Case "e", "E"
      ecount = ecount + 1
      TextBox3.Text = ecount
    Case "i", "I"
      icount = icount + 1
      TextBox4.Text = icount
    Case "o", "O"
      ocount = ocount + 1
      TextBox5.Text = ocount
    Case "u", "U"
      ucount = ucount + 1
      TextBox6.Text = ucount
  End Select
  counter = counter + 1
Loop
```

Pre-check condition

In this example, a pre-check condition 'Do…' loop is used, moving the condition to the beginning of the loop. Also notice that the condition statement has changed to 'while counter <=' to reflect the fact that it is working somewhat differently. Now, the loop will only work while the counter is still less than or equal to the length of the text in TextBox1.

In VB.NET® it is possible to use the 'Until' and 'While' forms in both pre- and post-check forms.

The final VB.NET® loop type is the '**While…End While**'.

This works in a similar fashion to the pre-check 'Do...' loop as shown above, so we won't dwell too much on it here!

Its VB.NET® syntax is:

```
Dim counter As Byte
counter = 0
While counter < 10
  counter = counter + 1
  MsgBox("Loop has run " &
  counter & " time(s)!")
End While
```

Menu

Perhaps one of the most important aspects of any EDP application is its ability to respond to triggers generated by menu systems. Drop-down menus are a popular component of modern software applications, so you have to be able to create them and program their event handlers.

Step 1

Have a good idea of the menu functions you require and how you want the menu organised. This can be achieved by drawing up a quick plan of the options and how they are grouped.

For example:

File	Edit	View	Help
Open file	Copy	Orders	Help about this program
Save file	Paste	Sales	About this program
Exit			

Where possible, and to improve Human Computer Interaction (HCI), it is advisable to try to keep menu options **similar** to those seen in **other** commercial programs (or even the operating system's GUI). This gives the user an instant familiarity with your software and makes it **more approachable** and **user-friendly**.

Step 2

The VB.NET® approach to creating menus is different from that of its predecessor (VB6®).

In VB.NET® a new category of controls is present in the ToolBox: Menus & Toolbars:

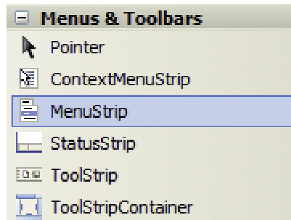


Figure 14.33 VB.NET® ToolBox category Menus & Toolbars

The control we want to use is the **MenuStrip**. Select this and draw it on your empty form.

Don't worry about where you draw it; it will automatically default to the top of the form. This is shown in Figure 14.34.

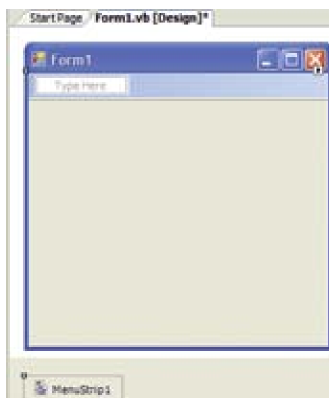


Figure 14.34 VB.NET® form with empty MenuStrip

Step 3

You can now start to add in the text labels (as planned in Step 1) that represent your menu options. Click each 'Type Here' box and enter the appropriate label (moving down or across):

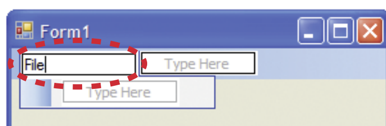


Figure 14.35 Adding the MenuStrip labels

Step 4

When you have finished, you should have a MenuStrip that looks like the plan from Step 1:

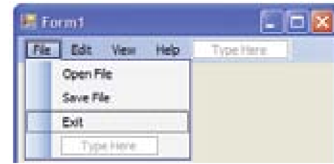


Figure 14.36 Completed MenuStrip

Any unwanted menu items can be selected and removed with the **Delete key**.

Step 5

Running the EDP application will result in a partially working menu system; it will appear and you will be able to navigate through the options but it will not respond to any triggers as we have not yet coded any event handlers.

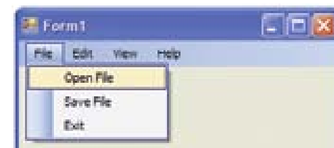


Figure 14.37 Menu in action

The next step is therefore to add an event handler.

Step 6

In design mode, double click the 'Open File' menu option and add the following code to the 'OpenFileToolStripMenuItem' click event handler:

```
MsgBox("This would open a new
file", MsgBoxStyle.Information)
```

This will simply pop up a message box until we are ready to code the full actions of this menu option. It will, however, let us check to see that our menu is working when the program is run.

From here, you can simply repeat Step 6 for each menu option and add program code appropriate to each event handler.

Making improvements to the menu

VB.NET® allows us to make functional improvements to our menu system.

What follows is a basic overview of three possible improvements to the existing menu that you could consider.

1. Use of shortcut keys

Shortcut keys can be used to access menu systems without relying on the mouse. For users who remember **keyboard shortcuts** this can be a useful time saver.

It is recommended that you select shortcuts commonly used in commercial software so that you keep your application consistent.

For example, 'File open' is usually given the keyboard shortcut **CTRL + O**.

This would mean the user holding down the CTRL (Control) key and pressing the 'O' key.

To add this, select the required menu label in the design view and modify the appropriate properties of this menu option in the Properties Window as shown in Figure 14.38.

When the application is run again, the menu option can now be accessed using the CTRL + O shortcut key.

2. Using a separator

As the name suggests, a **separator** is a horizontal line that is used to distinguish different parts of a menu system.

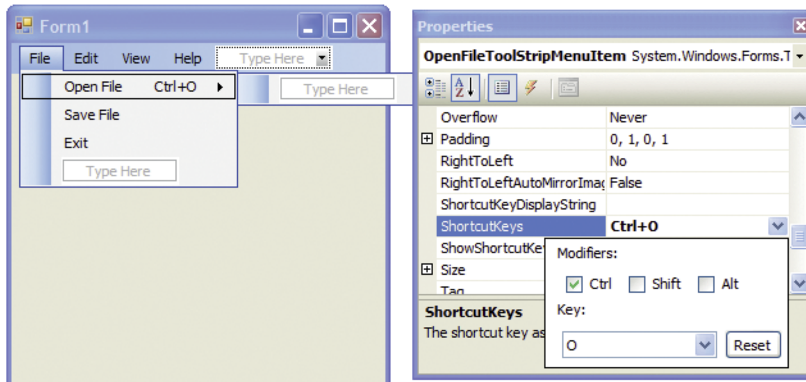


Figure 14.38 Adding a shortcut

A common usage is to separate loading, saving, printing and exit options underneath the File menu option.

We can add a separator to our File menu option by selecting the Exit option and choosing the 'Insert a separator' option from the right-click context menu as shown in Figure 14.39.

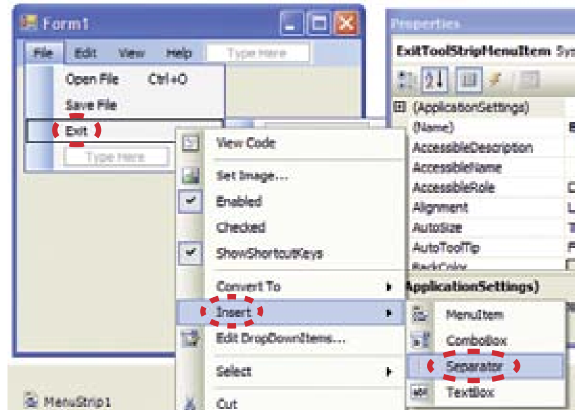


Figure 14.39 Adding a separator

When the application is run again, the separator is clearly visible (Figure 14.40).

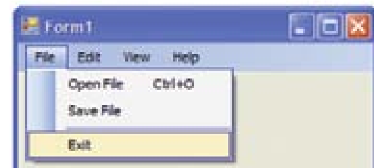


Figure 14.40 Separator between 'Save File' and 'Exit' options

3. Adding a single letter key press for each menu item

You may have seen a Microsoft Windows® application where menu items have certain letters **underlined**. For example, in Microsoft Word®:

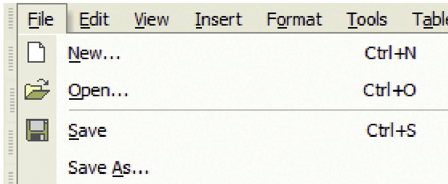


Figure 14.41 Single-letter keyboard shortcuts

This is achieved by use of the **ampersand symbol** (&) in the menu item's Text property. Simply place the ampersand before the letter you wish to use as a key press alternative for the menu item (Figure 14.42).

Many other options for enhancing (including ToolTips – see 14.4.3) exist in VB.NET®.

It is recommended that you experiment with the settings and facilities available in order to create intuitive and functional menu systems for your own EDP applications.

Activity 5

Creating a menu system

Use VB.NET®'s MenuStrip control to create the following menu system:

File		Reports	Help
Open	CTRL + O	Customers	About this program
Save	CTRL + S	Orders	
__________		Sales	
Exit			

Remember to create shortcuts, separators and key press options as indicated!

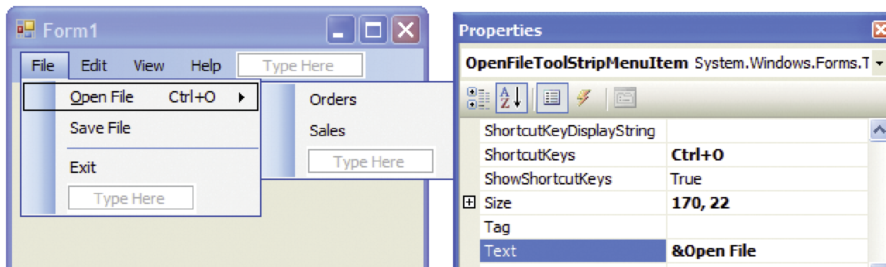


Figure 14.42 Adding single-letter keyboard shortcuts

Online help

This section will cover the following grading criterion:

P6

Make the Grade **P6**

For this criterion you have to create online help to assist the users of your EDP applications.

This should be easy to evidence either through screen captures of the application or writing the user documentation (see section 14.4.7).

The definition of 'online help' is quite varied. The list on this page gives you ideas of some possible elements you could include and demonstrates how to build a simple web-based solution.

Online help can be incorporated into an EDP application in a number of different ways, including:

- **colour-coded** forms
- **logical layout** of form elements to aid user navigation with suitable labels and grouping of related inputs
- use of **tab key** indexing on form elements to assist user navigation
- **menu system** with single-letter keyboard shortcuts (as demonstrated)
- a separate **help option** in the menu system
- use of **tooltips** (see section 14.4.3)
- **validation** of input with pop-up message boxes (see section 14.4.2)
- **launch a web browser** to display a pre-written HTML 'help' web page (see below).

Creating web-based online help

1. Create a sample web page that contains help text and images for your application. You can either code this by using Microsoft Notepad® or use a recognised web design tool, for example, Adobe Dreamweaver®.
2. Create a new windows application in Visual Basic.NET®.
3. On Form1, add a new MenuStrip with a Help -> View Help option (typically on far right).
4. Add a new Windows Form to the Project.
5. Select the new Form (Form2), rename it to 'Help' (text property).
6. Using the toolbox, add a WebBrowser common control to Form2.
7. Edit the WebBrowser control's URL property to the name of your web page.
   Note: You must include the file's full pathname, e.g. C:\Users\you\Desktop\help.html.
8. Select Form1, double-click the Help -> View Help option, then add the following code:

```
Form2.Show()
```

9. Run the application. You should find that clicking on the menu option will open a second form which, acting as a web browser, will display your customised help web page!

Debugging tools

Like most programming languages with an IDE (Integrated Development Environment), VB.NET® has extensive **debugging tools**.

Perhaps the most common debugging tools are:

- breakpoint
- step into (similar to a Trace)
- watch
- immediate Window.

Unit link

Unit 6 – Software design and development, section 6.2.1 also examines debugging tools but does so from a C#® perspective. You'll notice that debugging tools tend to function similarly from language to language. Therefore to avoid unnecessary duplication, the tool definitions are not repeated here and instead the focus is on how these tools are used in VB.NET®.

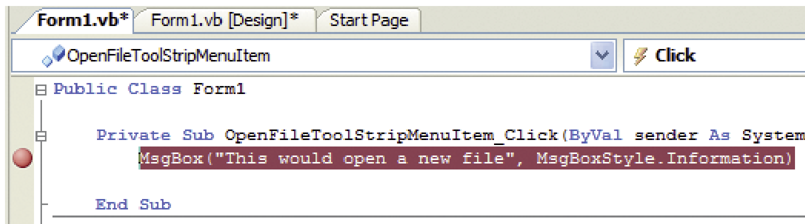


Figure 14.43 A breakpoint is marked on a specific line of code

Let's examine each of these in turn:

Breakpoint

Breakpoints can be placed into a VB.NET® program by either clicking in the left-hand margin (alongside the code window), by using the Debug menu or by pressing the F9 key which toggles the breakpoint on and off.

A breakpoint is signified by a red ball and inverse colouring of the line of code (see Figure 14.43).

When the EDP application is run, it will temporarily halt at the breakpoint, so:

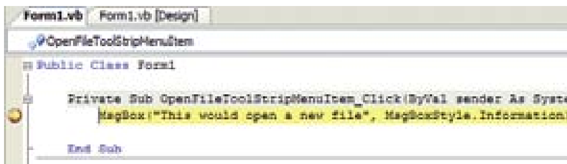


Figure 14.44 An active breakpoint on a line of code

The developer can then decide how to proceed; this might mean examining variables (see section 14.2.3) or tracing the remainder of the code line-by-line.

Step Into

Step Into is activated either by using the Debug menu or by pressing the F8 key, which will let the developer trace the execution of each line of code separately.

Another feature called **Step Over** (Shift+F8 key or the Debug Menu) can be used in a similar fashion but it executes all the statements in a block of code at once; this is useful if you are moving between sections of the program that need to be traced line by line (Step Into) and sections which you are confident are OK (Step Over).

Step Out can be used to move from Step Into to Step Over functionality.

Watch

A **Watch** may be added to a variable (see 14.2.3) in a number of ways.

The most straightforward way to add a Watch is to debug the program (F5), pause the execution (Break All toolbar button) and right-click on the variable you wish to watch. This will display a shortcut menu which has an option 'Add Watch'.

Adding a Watch will result in the chosen variable appearing in a separate Watch window.

Once the program is paused it can be traced using F8 (Step Into). This allows the developer to see the values in the Watch window changing as the variables are processed by each active line of program code. See Figure 14.45.

Right-clicking on the variable's name in the Watch window will display a further shortcut menu. One of the options is to delete the Watch, allowing the developer to manage active Watches effectively.

Immediate Window

This is a separate window which can be used to output debugging information whilst the VB.NET® application is running. If it disappears, it can be made visible again with **CTRL+G**.

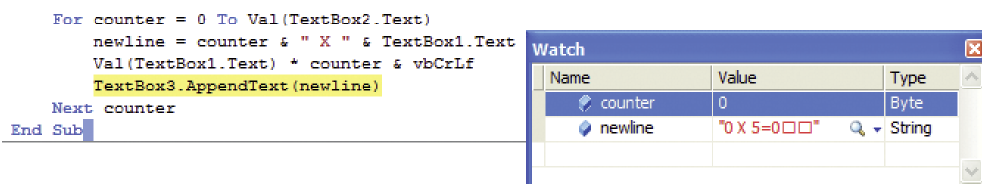


Figure 14.45 Debug window showing contents of variables being processed

Usually located at the bottom left of the design screen, it can be made to float as required or added as a tab (alongside the Form Design and Code windows).

A simple use of the Immediate Window is to output calculated values as the application runs, without affecting the appearance of the form as shown in Figure 14.46.

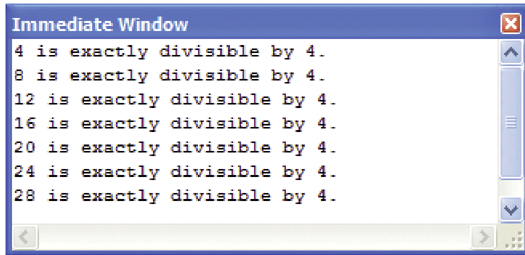


Figure 14.46 Floating Immediate Window displays debug information

This can be achieved using **Debug.Print** like so:

```
Dim counter As Byte
counter = 1
Do While counter < 30
   Dim remainder As Byte
   remainder = counter Mod 4
   If remainder = 0 Then
     Debug.Print(counter & " is
exactly divisible by 4.")
   End If
   counter = counter + 1
Loop
```

In this example, the **Mod** (Modulus) operator is used to discover whether or not a number is exactly divisible by four (it will have a 0 remainder if this is the case).

The output is sent directly to the Immediate Window by use of the Debug.Print statement.

14.2.3 Variables

Identifiers – variables and constants

An **identifier** is simply a name that represents a value. The name is used as an alternative to referring to a value's **memory address** in **RAM** (names are **friendlier** and **easier** to **remember**). Identifier names should **not** be the same as any existing control's property.

VB.NET® uses two kinds of identifier: **variables** and **constants**.

> **Key terms**
>
> A **variable** is an identifier whose value can change while the program runs. Variables can only store one value at a time; if a second value is assigned, the older value is overwritten.
>
> A **constant** is an identifier whose value cannot change while the program is running. Constants represent fixed values that may be used in program code instead of using text or numeric values, for example, 'Pi' instead of 3.14 to improve both its readability and maintainability.

Variables and constants are absolutely critical to writing program code; without these there could be no true processing (see section 14.3.1).

Data types and declaration

In order to create a variable or constant, it is necessary to write a declaration. In VB.NET® a variable is created through the use of the '**Dim**' keyword (standing for **Dimension**). In addition, the programmer must select the correct data type for the variable or constant.

> **Key terms**
>
> A **data type** is an essential building block for programming; data types are used to specify the kind of value that a programmer needs to store, for example, numeric, text, date, Boolean.
>
> A **declaration** in VB.NET® is a statement which, for a variable, states the variable's name and data type.
>
> In addition, a **constant declaration** will also assign the constant's value.

The following example demonstrates declarations of variables and constants in VB.NET®:

```
Dim bAge As Byte
Dim iMatchAttendance As Integer
Dim sUsername As String
Dim dItemPrice As Decimal

Const PI As Double = 3.141592
Const PASSWORD As String =
"LETMEIN"
```

In VB.NET®, constants are created by using the '**Const**' keyword.

Table 14.01 lists popular VB.NET® data types.

When creating variables and constants it is usually considered **good practice** to select a data type which can store expected values in the **most efficient** manner. For example, selecting a 'Long' data type to store a user's age in years would not be appropriate; it's **wasteful** of RAM.

Scope of variables

Scope refers to the visibility of a variable; whereabouts in the program it can be seen (and therefore used).

For example, a variable declared inside an event handler (e.g. 'Click') can only be used inside that event handler; it doesn't exist outside it. This is called a **local variable**.

Table 14.01

Data type	RAM requirement	Range of values
Boolean	4 bytes	True or False
Byte	1 byte	0 to 255 (unsigned)
Char	2 bytes	0 to 65535 (unsigned)
Date	8 bytes	January 1, 1 Common Era to December 31, 9999 CE
Decimal	12 bytes	+/–79,228,162,514,264,337,593,543,950,335 with no decimal point. +/–7.9228162514264337593543950335 with 28 places to the right of the decimal. The smallest non-zero number is +/–0.0000000000000000000000000001
Double	8 bytes	–1.79769313486231E308 to –4.94065645841247E-324 +4.94065645841247E-324 to +1.79769313486232E308
Integer	4 bytes	–2,147,483,648 to 2,147,483,647
Long	8 bytes	–9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
Object	4 bytes	Any type can be stored in a variable of type Object; it's often referred to as a universal data type. Similar to a 'variant' in Visual Basic 6®.
Short	2 bytes	–32,768 to 32,767
Single	4 bytes	–3.402823E38 to –1.401298E-45; +1.401298E-45 to +3.402823E38
String	10 bytes + (2 * string length)	0 to 2 billion Unicode characters (approximately)
User-defined type (structure)	Sum of the sizes of its members	Each member of the structure has a range determined by its data type and independent of the ranges of the other members.

A variable declared inside the Form Class can be used in any of the event handlers; it has wider scope.

In addition the keywords 'public' and 'private' are used to mark variables (and event handlers) as accessible from only within (private) their class or from outside (public). These are called **access modifiers**. Unless made public, variables are private by default.

In VB.NET® variables can belong to one of four basic scopes:

1. **Block** – new in VB.NET®, these are variables declared within a construct, for example, a loop or selection, and cannot be used outside it. For example:

```
Dim counter As Byte
counter = 1
Do While counter < 30
  Dim remainder As Byte
  remainder = counter Mod 4
  If remainder = 0 Then
    Debug.Print(counter & " is
    exactly divisible by 4.")
  End If
  counter = counter + 1
Loop
```

   In this example the variable called '**remainder**' can only be used within the 'Do…' loop block.

2. **Procedure** – variables declared within an event handler are available throughout that event handler. For example:

```
Private Sub Button2_Click(ByVal sender As System.Object, _
ByVal e As System.EventArgs) Handles Button2.Click

   Dim counter As Byte
   counter = 1
   Do While counter < 30
      Dim remainder As Byte
      remainder = counter Mod 4
      If remainder = 0 Then
        Debug.Print(counter & " is exactly divisible by 4.")
      End If
      counter = counter + 1
   Loop
End Sub
```

3. **Module** – variables declared within a separate module are limited to use within that module.

   The term can also apply when a variable is declared within a class (e.g. a Form class), effectively limiting the use of the variable to that class, but any event handler inside it. For example:

```
Public Class Form1
  Dim Username As String
End Class
```

4. **Namespace** – this is a more modern programming idea where defined namespaces are used to collect together identifiers (variables and constants). When a namespace is used, its set of variables and constants is active.

Where possible, scope should be kept as **local as possible** – it **reduces memory consumption** and reduces the opportunity for variables to be used incorrectly; this also **helps debugging** by limiting the opportunities where variables can be given 'bad' values.

14.3 Be able to design event driven applications

This section will cover the following grading criterion:

P3

Make the Grade P3

For this criterion you have to design your event driven program. This is likely to be in response to a problem set by your tutor unless you have been given a 'free' project to work on.

Any design method is acceptable as long as it clearly demonstrates the **appearance** of the application (e.g. the form's design), how the various processes are **triggered** (e.g. a storyboard or action/event table) and what the actual **inputs**, **processes** and **outputs** are.

A **data dictionary** can also be used to detail all the variables and so on that need to be declared.

The sign of a good design is that it should be relatively easy for another developer to implement a working program from it.

14.3.1 Specification

The specification of an event driven application relies on the developer assembling the following:

Purpose What is the program for?
What is it supposed to achieve?

User needs What are the user's needs?
What do they want the program to achieve?
How do they want it to work?
Who is going to use it?
How accurate does it need to be?
How reliable and robust does it need to be?

Inputs What are the inputs?
What type and quantity are they?
How frequently are they required?

Processes What processes are required to calculate the outputs required by the user?
How do we get from the input to the output?

Outputs What outputs are required?
How are they to be presented?
How should the information be formatted?
How often is the output to be generated?

Backing storage
What kind of data needs to be stored?
How is the data to be organised?
How long does the data need to be kept?
How will this data be accessed?

Unit link
Unit 11 – Systems Analysis and Design deals with this topic in far greater detail than is possible here.

14.3.2 Design

Due to its event driven approach, VB.NET® does not really lend itself to traditional design techniques such as pseudocode, flowcharts and structure diagrams.

Other techniques such as form design, storyboards and action/event charts are preferable.

Form design

Good form design is critical to the success of a graphical user interface-based event driven program. As such there are a number of simple rules we can apply to ensure the best user interaction experience possible.

Forms should:

- be **sensibly** laid out – with related information grouped together
- not have any **spelling** or **grammatical** errors (users will lose faith in the software)
- use a **logical** tab order (Tab key) to move between form elements
- use **consistent** formatting (colours, font etc.)
- use helpful labelling
- have **error detection** built in
- ideally support users with **disabilities**, e.g. have font size changes and audible prompts to **improve accessibility**
- have **online help** available.

Forms may be drawn manually (i.e. by hand) or developed using a graphical package on a computer system.

Case Study

The management at Frankoni T-Shirts have asked you to design a new application that they can install on PCs in their retail outlets that will allow users to select the design and features of their personalised T-shirt.

The program will then produce a rough image showing the final product and generate its cost based on size, quantity and so on.

Figure 14.47 shows a hand-drawn form design for the case study described above.

As you can see, the designer has taken the initial needs as identified by the user (Frankoni T-Shirts) and has sketched a suggested user interface. No functionality has been specified yet. The sketch should be presented to the user for feedback and (if necessary) revision.

The final form design will be implemented using a suitable EDP language such as VB.NET® (see section 14.4.1).

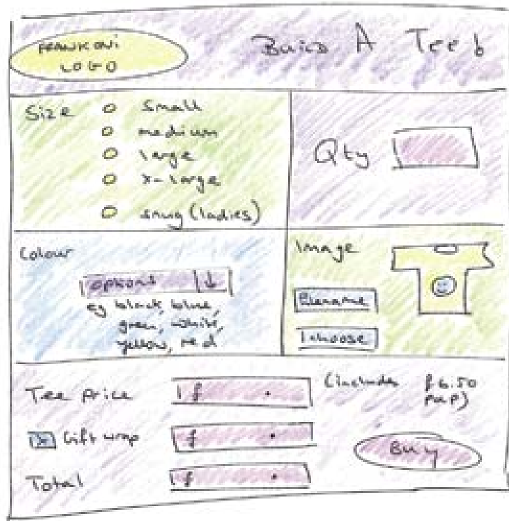


Figure 14.47 Frankoni T-Shirts form design sketch

Storyboard

Storyboards can be created as a way of seeing an event driven solution in an outline fashion.

There is no need to specify functionality (how the event handlers work); all that is necessary is to identify what needs to be done and how multiple forms and components link together and are triggered.

Creating a simple storyboard for this is the logical solution.

Scorecard – Storyboard

- \+ Easy to show to others and discuss as it's a visual tool.
- \+ Can get quick feedback.
- \+ Not linked to any programming language.
- \+ Can be redrawn quickly; more efficient than building prototypes.
- – Cannot be tested interactively.
- – Can become outdated quickly as designs can change rapidly.

Figure 14.48 shows a typical storyboard for a password-controlled stock control system.

In this figure, the designer has drawn a few forms and given basic linkage through the use of arrows. Additional notes are included to help explain the linkage of the forms.

Event procedures and descriptions

This is a useful reference table that lists each form, trigger and event handler for an application. In addition, it also describes the processes performed by each event handler.

Table 14.02 is a suggested Action/Event table for the storyboard shown in Figure 14.48.

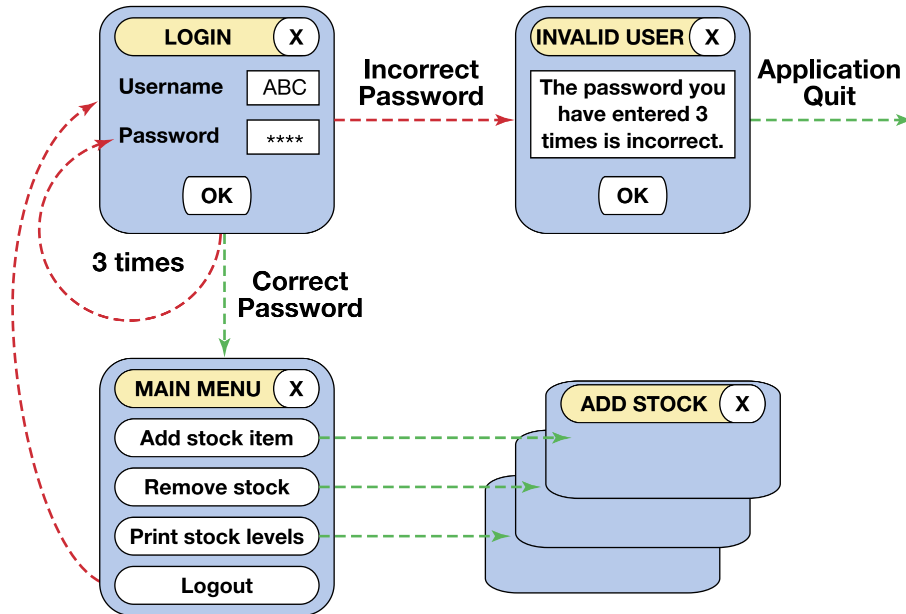


Figure 14.48 Storyboard for a password-controlled stock control system

Table 14.02

Form name	Trigger	Event handler	Event handler description
Login	OK button clicked	Okbutton_ Click()	Checks to see whether the password is correct for the given user name. Also keeps a track of the number of attempts the user has made. An incorrect attempt adds one to the number of attempts and clears the TextBoxes. If three attempts are made unsuccessfully, the Invalid User form is opened. A correct attempt opens the Main Menu form.
Login	Close window button clicked	'Windows Close' System event handler	A System Event that closes the application.
Invalid User	OK button clicked	Okbutton_ Click()	Closes the application after the warning has been displayed.
Invalid User	Close window button clicked	'Windows Close' System event handler	A System Event that closes the application.
Main Menu	Add stock item button clicked	Addstock_ Click()	Opens the Add Stock item form when clicked.
Main Menu	Remove stock item button clicked	Removestock_ Click()	Opens the Remove Stock form when clicked.

Form name	Trigger	Event handler	Event handler description
Main Menu	Print stock levels button clicked	Printstocklevel_Click()	Opens the Print stock levels form when clicked.
Main Menu	Logout button clicked	Logout_Click()	Closes the current form and reopens the Login form.
Main Menu	Close window button clicked	'Windows Close' System event handler	A System Event that closes the application

As you can see, between them the **form design, storyboard** and **action/event table** start to build the mechanics and appearance of the required EDP solution. From here, these designs are implemented using a suitable EDP language such as VB.NET®

14.4 Be able to implement event driven applications

This section will cover the following grading criterion:

P4

Make the Grade **P4**

This criterion is the key aspect of the unit, testing your ability to **use** an event driven programming language to **build** a working application **from** a design.

It is likely that your tutor will ask you to take the design elements **you built** for P3 and implement them using a language such as Visual Basic.Net®.

Pay attention to the IDE, debug facilities, language syntax and required programming standards to build a solid, reliable and usable application that meets the needs defined by your tutor.

14.4.1 Creation of application

14.4.2 Programming language syntax

14.4.3 Constructs

This section will integrate these learning outcomes and walk you through the creation of a new EDP application. We will use the Frankoni T-Shirt company case study and form design as introduced in section 14.3.2 as our active problem.

Caution!

Once again the following example has been broken down into a number of steps due to its complexity.

If you follow each step carefully, everything should work. If you discover a problem, go back a step (or two) to see if you've missed anything.

Step 1

Create a new VB.NET® project. Use the development environment to add these controls based on the original form design's elements and layout (see Figure 14.49).

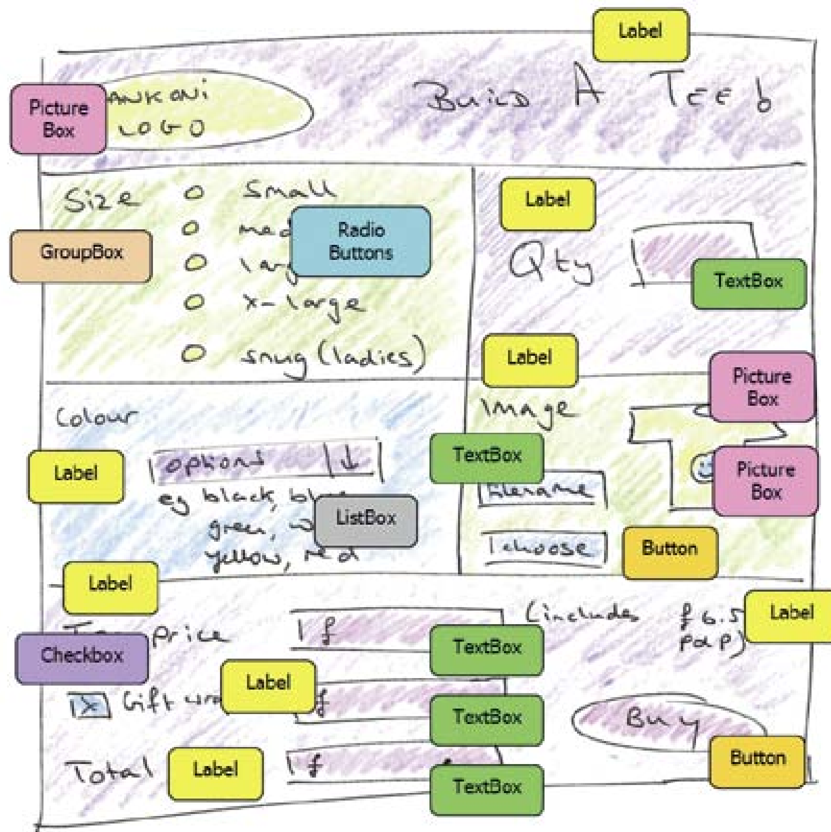


Figure 14.49 Identifying the VB.NET® controls

Step 2

All of these controls can be found in the various sub-categories in VB.NET®'s ToolBox (see section 14.2.2).

Figure 14.50 represents an attempt at recreating the form in VB.NET®.

Figure 14.50 VB.NET® recreation

Although functional and as close to the suggested form design as possible, it looks a bit unbalanced; particularly with the 'Qty' TextBox seemingly floating alone in the middle of the form. In reality, this would be an **iterative process** with the design being shown to the end users to make suggestions until they are happy with the final product.

Some minor changes have been made; two Panel controls have been added to the 'Colour' and 'Image' parts of the form. In addition, there are now two PictureBoxes in the Image panel – one inside the other. The inner one will show our image, the outer one will display the correct colour T-shirt (as selected from the Colour ListBox to the left). The colours in the ListBox (stored in the Items property) have also been put into ascending alphabetical order, that is A to Z.

All of the text is formatted to Tahoma, 12pt regular.

> Activity 6
>
> *Recreating the Frankoni 'Build A Tee!' form*
>
> Now it's your turn! Use VB.NET®'s Form Designer to recreate the manual form.
>
> For now, it is recommended that you do not change any of the default control names (although in reality this would be considered to be good practice).

Step 3

The next step is to code the functionality of the form, identifying the triggers and necessary event handlers that are needed.

Perhaps the best way to do this is to produce the storyboard (Figure 14.51).

As you can see, there are a number of calculations to perform.

Upon investigation we find that the current T-shirt prices are:

Small	£7.50
Medium	£9.50
Large	£10.50
X-Large	£12.50
Snug (Ladies)	£11.00

In addition, while gift wrapping is charged per tee (£2.00), the £6.50 postage and package charge is considered to be a flat rate (Frankoni absorbs the postal charges on larger shipments as a goodwill gesture and in the hope of future business).

The 'File Selection' dialogue will be a Control that exists as part of the Microsoft Windows® API.

Step 4

The next step is to build a basic action/event table like Table 14.03.

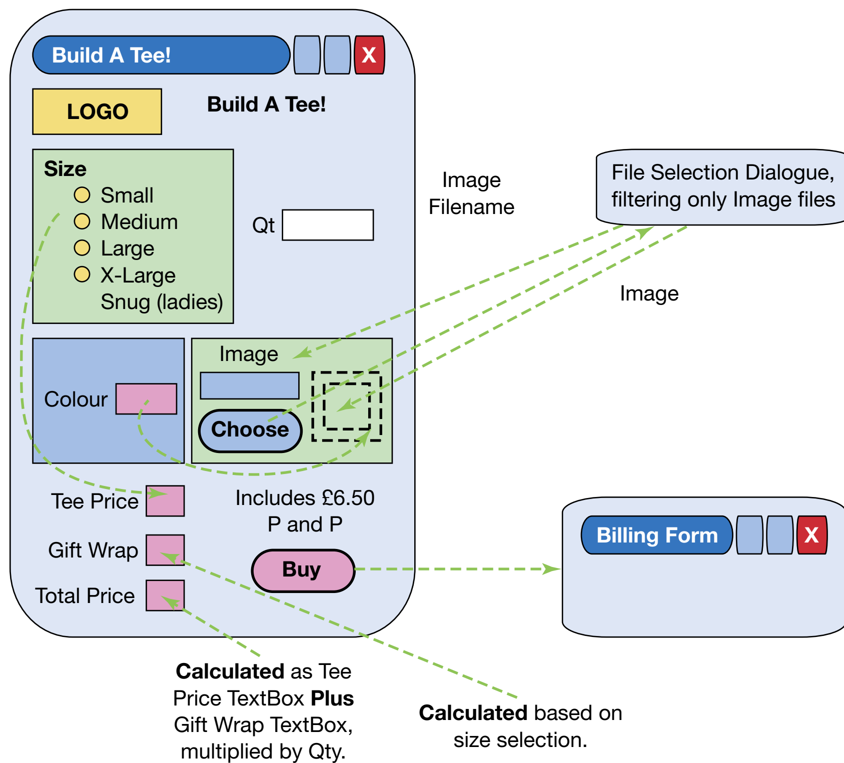


Figure 14.51 Storyboard

Table 14.03

Form name	Trigger	Event handler	Event handler description
BuildATee	'Buy' button clicked	BuyButton_Click()	Opens the Billing Form.
BuildATee	'Choose' button clicked	ChooseButton_Click()	Opens a File Open Dialogue. The selected image's filename is stored in the Filename TextBox. The selected image itself is loaded into the inner PictureBox.
BuildATee	Colour is selected from listbox	ColourList_ SelectedIndexChanged()	The colour is used to select the correct image for the T-shirt background that forms the outer PictureBox.
BuildATee	'Close window' button clicked	'Windows Close' system event handler	A system event that closes the application.
BuildATee	'Gift Wrap' checkbox is checked or unchecked	GiftWrap_ CheckedChanged()	If the GiftWrap option is selected, the associated TextBox is set to £2.00. If the option is unchecked, the TextBox is cleared. TotalPrice TextBox is also updated to show TeePrice + GiftWrap values.
BuildATee	'Small Size' is selected from RadioButtons	SmallSize_ CheckedChanged()	Tee Price TextBox is given the value of: £7.50 x Qty TextBox + £6.50
BuildATee	'Medium Size' is selected from RadioButtons	MediumSize_ CheckedChanged()	Tee Price TextBox is given the value of: £9.50 x Qty TextBox + £6.50
BuildATee	'Large Size' is selected from RadioButtons	LargeSize_ CheckedChanged()	Tee Price TextBox is given the value of: £10.50 x Qty TextBox + £6.50
BuildATee	'X-Large Size' is selected from RadioButtons	XLargeSize_ CheckedChanged()	Tee Price TextBox is given the value of: £12.50 x Qty TextBox + £6.50
BuildATee	'Snug Size' is selected from RadioButtons	SnugSize_ CheckedChanged()	Tee Price TextBox is given the value of: £11.00 x Qty TextBox + £6.50
BuildATee	Value in Qty TextBox is changed	Qtychosen_ TextChanged()	Cascade to appropriate RadioButton event handler to recalculate the total price.

Step 5

You'll notice that the design work has necessitated us changing the object names into something more meaningful. This is achieved by selecting the form object and changing its (Name) property in the Property Window, as shown in Figure 14.52.

Figure 14.52 Changing the object names

As noted, changing the names of form objects is **good practice** as it makes it easy for the developer to remember what an object actually is when reading through a section of code. It also helps to **self-document** the program code.

Various techniques are used to name objects in a professional development environment. For now, we will keep things straightforward and use simple, understandable names.

The recommended changes to this EDP application are shown in Figure 14.53 and Table 14.03.

Note: It is, of course, possible that you may have added objects to the form in a different order. If this has happened, ignore the 'old name' column and just focus on the key letter and the new name.

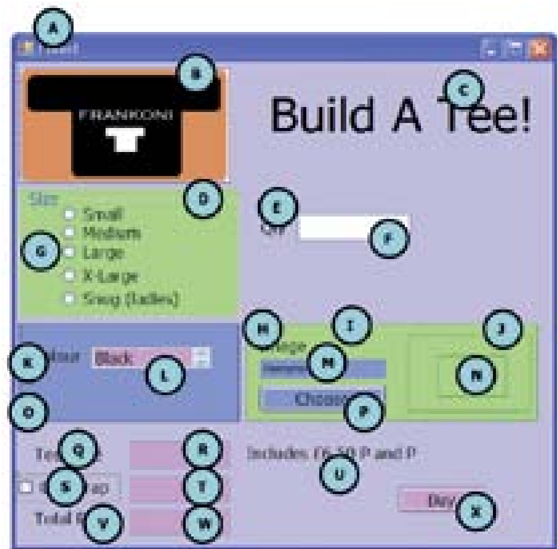


Figure 14.53 Application with renamed objects

Table 14.04

Key	Control	Old name	New name	Additional new properties
A	Form	Form1	BuyATee	**Text** = Buy A Tee!
B	PictureBox	PictureBox1	Logo	**SizeMode** = StretchImage (will help to autoscale the image selected)
C	Label	Label1	Title	
D	GroupBox	GroupBox1	SizeSelector	
E	Label	Label2	Qty	
F	TextBox	TextBox1	Qtychosen	
G	RadioButton	RadioButton1 RadioButton2 RadioButton3 RadioButton4 RadioButton5	SmallSize MediumSize LargeSize XLargeSize SnugSize	Checked = True (SmallSize)
H	Label	Label4	Image	
I	Panel	Panel2	ImagePanel	
J	PictureBox	PictureBox2	TeeImage	
K	Label	Label3	Colour	
L	ListBox	ListBox1	ColourList	Has to have entries in the **Items property**: Black, Blue, Green, Yellow, Red and White.
M	TextBox	TextBox2	Imagename	
N	PictureBox	PictureBox3	Teedesign	**SizeMode** = StretchImage (will help to autoscale the image selected)
O	Panel	Panel1	ColourPanel	
P	Button	Button1	ChooseButton	
Q	Label	Label5	TeePrice	
R	TextBox	TextBox3	TeePriceBox	**TextAlign** = Right
S	CheckBox	CheckBox1	GiftWrap	
T	TextBox	TextBox4	GiftWrapBox	**TextAlign** = Right
U	Label	Label7	PandP	
V	Label	Label6	TotalPrice	
W	TextBox	TextBox5	TotalPriceBox	**TextAlign** = Right
X	Button	Button2	BuyButton	

Table 14.05

Identifier name	Var or Const	Scope	Value	Date type	Description
dpostpack	Const	Module	6.5	Decimal	Current price of postage and packing
dindgiftwrap	Const	Module	2.0	Decimal	Current price of gift wrapping each tee
dsmallprice	Const	Module	7.5	Decimal	Current price for a small tee
dmediumprice	Const	Module	9.5	Decimal	Current price for a medium tee
dlargeprice	Const	Module	10.5	Decimal	Current price for a large tee
dxlargeprice	Const	Module	12.5	Decimal	Current price for an extra large tee
dsnugprice	Const	Module	11.5	Decimal	Current price for a snug (ladies) tee
dqty	Var	Module	–	Byte	Quantity of tees wanted by user
dgiftprice	Var	Module	–	Decimal	Cost of gift wrapping based on quantity of tees wanted
dteeprice	Var	Module	-	Decimal	Price of tees – includes P&P for quantity wanted
dtotalprice	Var	Module	-	Decimal	Total price – includes gift wrapping (if required)

Step 6

Declaring variables and constants for the application is also necessary. These can be recorded in a simplified form of data dictionary which describes each identifier as shown in Table 14.05.

These variables are then added to the program code under the Public Class line as below:

```
Public Class BuyATee

  Const dpostpack As Decimal = 6.5
  Const dindgiftwrap As Decimal = 2.0
  Const dsmallprice As Decimal = 7.5
  Const dmediumprice As Decimal = 9.5
  Const dlargeprice As Decimal = 10.5
  Const dxlargeprice As Decimal = 12.5
  Const dsnugprice As Decimal = 11.5

  Dim dqty As Byte = 0
  Dim dgiftprice As Decimal = 0.0
  Dim dteeprice As Decimal = 0.0
  Dim dtotalprice As Decimal = 0.0
```

Step 7

Now starts the fun part; we have to **code** some of the **event handlers** that match to the **identified triggers** in the action/event table.

We'll start by coding the **Size radio buttons** (SmallSize, MediumSize, LargeSize, XLargeSize and SnugSize).

Let's code the **SmallSize_CheckedChanged() event** handler by double clicking on this radio button.

Add the following code:

```
dqty = Val(Qtychosen.Text)
dteeprice = dsmallprice * dqty +
dpostpack
TeePriceBox.Text = Str(dteeprice)
TeePriceBox.Text =
Format(TeePriceBox.Text,
"Currency")
dtotalprice = dteeprice + dgiftprice
TotalPriceBox.Text =
Str(dtotalprice)
TotalPriceBox.Text =
Format(TotalPriceBox.Text,
"Currency")
```

Similar code will then be added to each radio button's event handler, making modifications to the tee price being charged in the calculation for the different sizes.

Step 8

The next step is to code the Qtychosen TextBox.

This is a little more complex as changing the quantity will result in changes to the Tee Price and the Total Price. In other words, we'll have to recalculate these costs as the quantity changes.

These calculations already exist in each Size radio button's **CheckChanged() event handler** (as seen in Step 7). Because of this, it seems a little silly to simply repeat the code. There must be another way!

One technique involves using the GroupBox (SizeSelector) to determine which radio button has been selected, but this is a little complex to be described here.

Instead, we will manually check (with 'If…then…else' statements) which Size radio button has been selected and trigger its event handler accordingly. This is effectively cascading an event (from the TextChanged() event handler in the Qty TextBox to the CheckChanged() event handler of the currently selected radio button).

Here is the code for the TextChanged() event handler for the Qtychosen TextBox:

```
If SmallSize.Checked Then
  SmallSize_CheckedChanged
  (sender, e)
End If

If MediumSize.Checked Then
  MediumSize_CheckedChanged
  (sender, e)
End If

If LargeSize.Checked Then
  LargeSize_CheckedChanged
  (sender, e)
End If

If XLargeSize.Checked Then
  XLargeSize_CheckedChanged
  (sender, e)
End If

If SnugSize.Checked Then
  SnugSize_CheckedChanged
  (sender, e)
End If
```

Step 9

This step involves adding the GiftWrap price of £2 per tee to the bill if the associated checkbox is selected. Of course, if the checkbox is unselected, the GiftWrap costs must be set to £0.00.

Here is the appropriate code:

```
If GiftWrap.Checked Then
  dgiftprice = dindgiftwrap * dqty
  GiftWrapBox.Text = Str(dgiftprice)
  GiftWrapBox.Text = Format(GiftWrapBox.Text, "Currency")
Else
  dgiftprice = 0.0
  GiftWrapBox.Text = Format("0.0", "Currency")
End If
dtotalprice = dteeprice + dgiftprice
TotalPriceBox.Text = Str(dtotalprice)
TotalPriceBox.Text = Format(TotalPriceBox.Text, "Currency")
```

This code is added to the **GiftWrap_Checked Changed() event handler**.

Step 10

Next let's try coding the change of T-shirt colour – the ListBox we've called ColourList.

This is reasonably straightforward but relies on good preparation. The easiest way to do this (without accessing Microsoft Windows® graphics API) is to use simple bitmaps to represent each plain T-shirt. These will be created using a graphic package such as Microsoft Paint®, Adobe Photoshop® or Corel Paintshop Pro®.

We'll need to create six T-shirt bitmap images, one for each option with uniform size, format and filename (Figure 14.54).

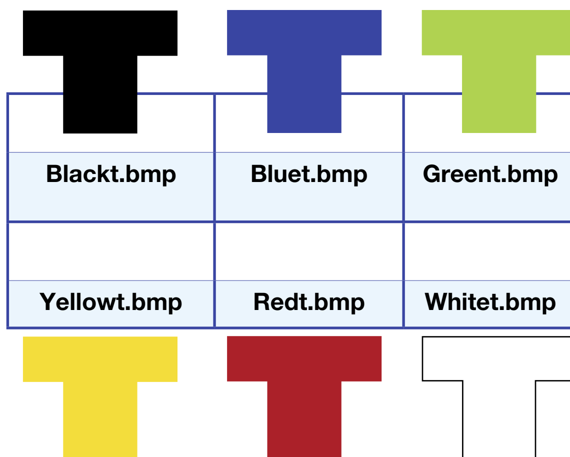


Figure 14.54

Double-click the ColourList to access the code window and add the following VB.NET® code to the **ChooseButton_SelectedIndexChanged handler**:

```
TeeImage.Image = System.Drawing.
Image.FromFile("C:\MyFolder\" +
ColourList.SelectedItem + "t.bmp")
```

This code will load a T-shirt bitmap image into the outer PictureBox object (TeeImage).

The bitmap loaded will be the value picked from the ListBox (e.g. Red) plus the string 't.bmp' giving the full filename of 'Redt.bmp'. The Help box shows an example of this code in action:

Help

Data validation

Validating user input is a key aspect when creating any EDP application. Validation is the process of checking to see whether something is valid.

In this application, the most obvious input value to validate would be the 'Qty' textbox (Qtychosen).

Validation would logically be added to the 'Buy' button's click event; if the 'Buy' button is clicked and the 'Qty' textbox has a value of less than 1, a suitable error message dialogue should be displayed (Figure 14.55).

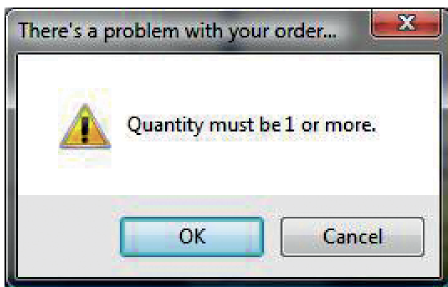


Figure 14.55

```
If Val(Qtychosen.Text) < 1 Then
  MsgBox("Quantity must be 1 or more.", vbExclamation + vbOK, "There's
a problem with your order...")
Else
  MsgBox("Would display the customer information form")
End If
```

Figure 14.56 Selecting a plain red T-shirt

Step 11

Let's move our attention to **ChooseButton's Click()** event.

In order to code this we have to add another object to the form. In this case, it is a special type of dialogue called the OpenFileDialog. It is located in the 'Dialogs' category in the ToolBox (see Figure 14.57).

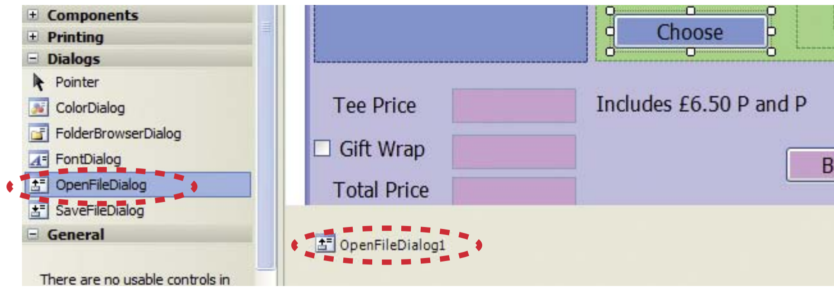


Figure 14.57 Adding an OpenFileDialog

Once selected, it is added to the bottom of your form in a separate area.

Double-click the ChooseButton to access the code window and add the following VB.NET® code to the ChooseButton_Click handler:

```
OpenFileDialog1.Filter = "Image
Files(*.BMP;*.JPG)|*.BMP;*.JPG"
OpenFileDialog1.ShowDialog()
Imagename.Text = OpenFileDialog1.
FileName
Teedesign.Image = System.Drawing.
Image.FromFile(Imagename.Text)
```

This section of code (a) sets a filter for only .JPG and .BMP files, (b) shows the OpenFileDialog, (c) stores the selected filename in the text property of our Imagename TextBox and (d) loads the image selected into the Image property of the inner PictureBox object (Teedesign).

Figure 14.58 is an example OpenFileDialog – I'm sure you will have seen this before while using Microsoft Windows®.

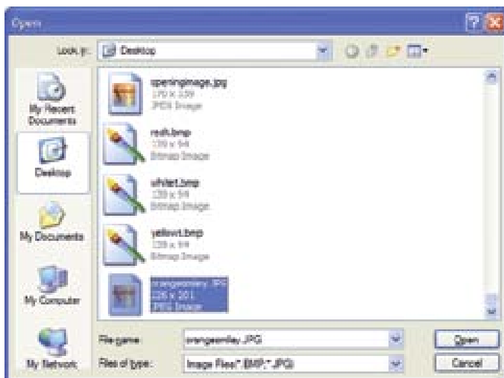


Figure 14.58 Our OpenFileDialog browsing for image files only

Putting this together now means we can select the size, T-shirt colour and image (Figure 14.59).

Figure 14.59 One white, small T-shirt with an orange smiley and no giftwrap

Help

Online help – ToolTips

You may notice that **Imagename** (the TextBox storing the chosen filename) is probably **too short** to store the **whole pathname** of the selected image.

A common solution to this is to use a **ToolTip**. ToolTips are helpful 'tags' that appear on a form object when the mouse is left **hovering** over it, for example:

Figure 14.60 An example ToolTip in Microsoft Word®

We can add a ToolTip to our EDP solution to improve the online help (Figure 14.61). The ToolTip control can be found in the Common Controls category in the ToolBox.

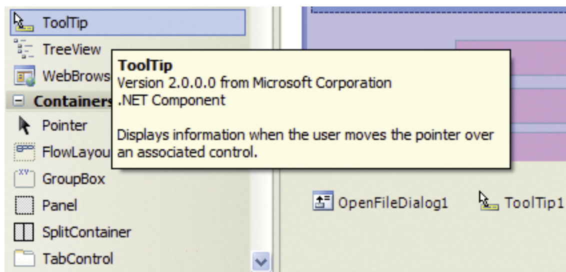


Figure 14.61 Adding a ToolTip to the EDP application

The ToolTip Control will be added to the separate area alongside the OpenFileDialog.

The next step is to add a line of code (ringed below) into the existing ChooseButton_Click handler:

```
OpenFileDialog1.Filter = "Image
Files(*.BMP;*.JPG)|*.BMP;*.JPG"
OpenFileDialog1.ShowDialog()
Imagename.Text = OpenFileDialog1.
FileName
Teedesign.Image = System.Drawing.
Image.FromFile(Imagename.Text)

ToolTip1.SetToolTip(Imagename,
Imagename.Text)
```

Run the application again, select an image file and then hover over the Imagename TextBox (Figure 14.62).

Figure 14.62 An example ToolTip, improving the online help

14.4.4 Programming standards

Organisations very often have programming standards that they ask their developers to adhere to. Typically this involves:

- **Comments** – should describe the code's purpose in the solution, not the syntax of the actual code itself.
- **Code layout** – code is indented to highlight the structure of the program, e.g. what code is part of an 'If…else' statement.
- **Identifier naming** – variables, constants etc. are all given meaningful and sensible names.

Together these aspects form part of the internal documentation of a program. In Visual Basic.NET® all of these can be simply accomplished, e.g. Visual Basic.NET® automatically indents program code to encourage good code layout, comments can be added to lines using a single apostrophe symbol (usually appearing in green text).

The full Frankoni EDP solution (with comments)

```
' Frankoni BuildaTee
'
' Written by M Fishpool
' January 2007
' Version 1
'
Public Class BuyATee
    Const dpostpack As Decimal = 6.5          ' price of packing and postage
    Const dindgiftwrap As Decimal = 2.0       ' price of individually gift wrapping 1 Tee
    Const dsmallprice As Decimal = 7.5        ' price of 1 small Tee
    Const dmediumprice As Decimal = 9.5       ' price of 1 medium Tee
    Const dlargeprice As Decimal = 10.5       ' price of 1 large Tee
    Const dxlargeprice As Decimal = 12.5      ' price of 1 extra large Tee
    Const dsnugprice As Decimal = 11.5        ' price of 1 snug Tee
    Dim dqty As Byte = 0                      ' quantity required as a whole number
    Dim dgiftprice As Decimal = 0.0           ' gift wrapping price as a decimal number
    Dim dteeprice As Decimal = 0.0            ' tee shirt price as a decimal number
    Dim dtotalprice As Decimal = 0.0          ' total price of order as a decimal number

    Private Sub ChooseButton_Click(ByVal sender As System.Object, ByVal e As  _
    System.EventArgs) Handles ChooseButton.Click
        '   Code to select an image file and load it onto the form
        OpenFileDialog1.Filter = "Image Files(*.BMP;*.JPG)|*.BMP;*.JPG"
        OpenFileDialog1.ShowDialog()
        Imagename.Text = OpenFileDialog1.FileName
        Teedesign.Image = System.Drawing.Image.FromFile(Imagename.Text)
        ToolTip1.SetToolTip(Imagename, Imagename.Text)
    End Sub

    Private Sub ColourList_SelectedIndexChanged(ByVal sender As System.Object,ByVal
e As _
    System.EventArgs) Handles ColourList.SelectedIndexChanged
        '   Code to load the appropriate plain Tee image onto the form
      TeeImage.Image = System.Drawing.Image.FromFile("C:\"+ColourList.
SelectedItem+ _
        "t.bmp")
    End Sub
    Private Sub SmallSize_CheckedChanged(ByVal sender As System.Object, ByVal e As _
    System.EventArgs) Handles SmallSize.CheckedChanged
        '  Code to recalculate costs if a small Tee is selected
        dqty = Val(Qtychosen.Text)
        dteeprice = dsmallprice * dqty + dpostpack
        TeePriceBox.Text = Str(dteeprice)
        TeePriceBox.Text = Format(TeePriceBox.Text, "Currency")
        dtotalprice = dteeprice + dgiftprice
        TotalPriceBox.Text = Str(dtotalprice)
        TotalPriceBox.Text = Format(TotalPriceBox.Text, "Currency")
    End Sub
```

```
Private Sub MediumSize_CheckedChanged(ByVal sender As System.Object, ByVal e
As _
System.EventArgs) Handles MediumSize.CheckedChanged
    ' Code to recalculate costs if a medium Tee is selected
    dqty = Val(Qtychosen.Text)
    dteeprice = dmediumprice * dqty + dpostpack
    TeePriceBox.Text = Str(dteeprice)
    TeePriceBox.Text = Format(TeePriceBox.Text, "Currency")
    dtotalprice = dteeprice + dgiftprice
    TotalPriceBox.Text = Str(dtotalprice)
    TotalPriceBox.Text = Format(TotalPriceBox.Text, "Currency")
End Sub

Private Sub LargeSize_CheckedChanged(ByVal sender As System.Object, ByVal e As _
System.EventArgs) Handles LargeSize.CheckedChanged
    ' Code to recalculate costs if a large Tee is selected
    dqty = Val(Qtychosen.Text)
    dteeprice = dlargeprice * dqty + dpostpack
    TeePriceBox.Text = Str(dteeprice)
    TeePriceBox.Text = Format(TeePriceBox.Text, "Currency")
    dtotalprice = dteeprice + dgiftprice
    TotalPriceBox.Text = Str(dtotalprice)
    TotalPriceBox.Text = Format(TotalPriceBox.Text, "Currency")
End Sub

Private Sub XLargeSize_CheckedChanged(ByVal sender As System.Object,ByVal e _
As System.EventArgs) Handles XLargeSize.CheckedChanged
    ' Code to recalculate costs if a large Tee is selected
    dqty = Val(Qtychosen.Text)
    dteeprice = dxlargeprice * dqty + dpostpack
    TeePriceBox.Text = Str(dteeprice)
    TeePriceBox.Text = Format(TeePriceBox.Text, "Currency")
    dtotalprice = dteeprice + dgiftprice
    TotalPriceBox.Text = Str(dtotalprice)
    TotalPriceBox.Text = Format(TotalPriceBox.Text, "Currency")
End Sub

Private Sub SnugSize_CheckedChanged(ByVal sender As System.Object, ByVal e As _
System.EventArgs) Handles SnugSize.CheckedChanged
   ' Code to recalculate costs if a snug Tee is selected
   dqty = Val(Qtychosen.Text)
   dteeprice = dsnugprice * dqty + dpostpack
   TeePriceBox.Text = Str(dteeprice)
   TeePriceBox.Text = Format(TeePriceBox.Text, "Currency")
   dtotalprice = dteeprice + dgiftprice
   TotalPriceBox.Text = Str(dtotalprice)
   TotalPriceBox.Text = Format(TotalPriceBox.Text, "Currency")
End Sub
```

```
    Private Sub Qtychosen_TextChanged(ByVal sender As System.Object, ByVal e As _
    System.EventArgs) Handles Qtychosen.TextChanged
        ‘  Code to recalculate costs by cascading the correct event handler when
        ‘  quantity changes
        If SmallSize.Checked Then
            SmallSize_CheckedChanged(sender, e)
        End If
        If MediumSize.Checked Then
            MediumSize_CheckedChanged(sender, e)
        End If
        If LargeSize.Checked Then
            LargeSize_CheckedChanged(sender, e)
        End If
        If XLargeSize.Checked Then
            XLargeSize_CheckedChanged(sender, e)
        End If
        If SnugSize.Checked Then
            SnugSize_CheckedChanged(sender, e)
        End If
    End Sub

    Private Sub BuyButton_Click(ByVal sender As System.Object, ByVal e As _
    System.EventArgs) Handles BuyButton.Click
        MsgBox(“Would display the customer information form”)
    End Sub

    Private Sub GiftWrap_CheckedChanged(ByVal sender As System.Object,ByVal e As _
    System.EventArgs) Handles GiftWrap.CheckedChanged
        ‘ to recalculate costs based on whether or not the customer has asked for
        ‘ Tees to be gift wrapped
        If GiftWrap.Checked Then
            dgiftprice = dindgiftwrap * dqty
            GiftWrapBox.Text = Str(dgiftprice)
            GiftWrapBox.Text = Format(GiftWrapBox.Text, “Currency”)
        Else
            dgiftprice = 0.0
            GiftWrapBox.Text = Format("0.0", "Currency")
        End If
        dtotalprice = dteeprice + dgiftprice
        TotalPriceBox.Text = Str(dtotalprice)
        TotalPriceBox.Text = Format(TotalPriceBox.Text, "Currency")
    End Sub
End Class
```

14.4.5 Testing

This section will cover the following grading criteria:

P5 M3

Make the Grade P5 M3

Getting your EDP application to run isn't the end of the job – you have to prove that it is working **as expected**.

P5 asks you to test the application.

M3 asks you to analyse the results. This means that you must compare and contrast the expected and actual results to see whether there are any discrepancies.

Thorough testing will make the review exercise required for D2 (see section 14.4.6) much easier!

Although **testing** and **debugging** can be seen as two separate stages of the EDP development path, they are linked to a degree.

Vigorous testing should discover the following types of error in a typical EDP:

- triggers not working or incorrectly identified
- event handlers not working properly
- calculations incorrect or with insufficient accuracy
- calculations not correctly formatted
- some functionality missing or incorrectly implemented.

Testing an EDP relies on the following:

1. Creating a **test strategy** (how you are going to test the application).
2. Creating a **test plan structure** – specifically:
   - What is being tested?
   - When is it being tested?
   - What are the expected results?
   - What are the actual results?
3. Comparing the **expected** and **actual results**:
   - Did it work as expected?
   - If not, what **corrective action** needs to be performed?
   - Which **error messages** occurred?

Putting it all together…

A Frankoni customer tries the program, selecting five large white T-shirts with an orange smiley logo. He would also like them gift wrapped.

Producing this calculation manually will give us our expected result:

Tee price: 5 x £10.50 =	£52.50
Add P&P (£6.50) =	£ 6.50
Plus gift wrap (5 @ £2.00) =	£10.00
Total price =	**£69.00**

We can then input these values and choices into our EDP solution to see if it functions correctly. This is shown in Figure 14.63 on page 48.

Figure 14.63 Testing the full application

From this calculation, it would appear that the program is **functioning correctly**. In addition, all triggers and event handlers seem to be working properly.

Finally, you may wish to check the **tab order** of each form control to ensure that the form can be easily (and logically) navigated, that is, top-to-bottom, left-to-right.

As we've already seen in section 14.2.2, specialist debug tools available in VB.NET® (Breakpoint, Step Into, Watch and the Immediate Window) are invaluable in the quest to identify and correct errors in the program code.

Unit link

Unit 6 – Software design and development, section 6.2.1, stage 6 also examines testing and debugging and while it does so from a C#® perspective, concepts such as **white box testing, black box testing** and **trace tables** are still relevant. It also provides an example of a typical test table.

Refer to these for additional guidance.

14.4.6 Review

This section will cover the following grading criterion:

D2

Make the Grade D2

This criterion follows on from P5 and M3 (which involved testing and analysis of test results).

Reviewing your own work is always difficult as you tend to be emotionally invested in it. This may mean that your tutor may ask you to review one of your peer's applications instead.

A review, produced as a report, video or a presentation, should say:

- how well it meets the **defined requirements**
- how **accurate** the application is
- how **reliably** the application works
- how **easy** the program is **to use**
- how **aesthetically pleasing** the application is
- any aspects which **need improvement**.

Review is a **critical process**, which occurs **after** testing and debugging.

Initially the major concern should be: how does the EDP compare to the user's original specifications?

If the correct level of **fact-finding** was conducted as part of the design (see section 14.3.2), the program should meet the user's needs accurately.

At the risk of finding out at the end of the process that considerable time and money has been wasted in creating something that does not meet the user's needs, the use of interim reviews that are regularly spaced along the development period are a good idea. These will prevent a program getting too far from its original design (something that is called '**feature drift**').

Reviews feed back into the design and implementation stages of an EDP, helping the programmer to understand what is **really wanted** rather than what **they think** is wanted.

Unit link

Unit 6 – Software design and development, Section 6.2.1, stage 8 also examines the review process. Refer to this for additional guidance.

Documentation

This section will cover the following grading criterion:

M4

Make the Grade M4

This criterion requires you to create technical documentation for the EDP application's future support and maintenance.

This is likely to be a written manual or a series of web pages that document how the program was created. A good test for technical documentation is whether it has been written in enough detail for another developer to alter it or add extra functionality post-review.

Documentation is another vital aspect and consists of material written for two different types of audience:

- **User documentation** – written for the end-user, documenting how the application is used.
- **Technical documentation** – written for other developers, documenting how the application works.

Unit link

Unit 6 – Software design and development, section 6.6.1, stage 7 gives more information on user and technical documentation.

Refer to this for additional guidance.

Help

EDP stages of development

1. Understand the user's needs.
2. Design screen layouts, work out data storage etc.
3. Implement forms and controls to reflect approved screen layouts.
4. Identify triggers needed.
5. Code associated event handlers (including documentation).
6. Debug the program.
7. Test the program.
8. Review (may occur during stages if required).
9. Produce user and technical documentation.

Activity 7

More EDP problems

1 Kris Arts and Media Ltd requires an EDP that will calculate a quote for specific marketing and advertising jobs.

Generally they use the formula below when creating a sales brochure.

One-off costs for the one-off sample brochure:

- Black and white printing £300 or
- Greyscale printing £500 or
- Full colour printing £900
- Costs are £3 per page (matt stock paper) and/or
- Costs are £5 per page (gloss stock paper)
- Costs are £25 per illustration/graphic
- Costs are £35 per photograph (black and white)
- Costs are £45 per photograph (colour)
- Duplication of the brochure to PDF format for electronic distribution: £100

Costs per printed brochure:

- Duplication of each brochure is £1.25

2 Kris Arts and Media Ltd would like a simple 'colour picker' that allows them to use three scrollbars (Red, Green, Blue) to generate a true colour shade. Each scrollbar should permit movement between 0 and 255.

The program should display the RGB colour code and generated colour.

Unit links

Unit 14 is a **mandatory unit** for the Edexcel BTEC Level 3 National Extended Diploma in IT (Software Development) pathway and **optional** for all other qualifications and pathways of this Level 3 IT family.

Qualification (pathway)	Mandatory	Optional	Specialist optional
Edexcel BTEC Level 3 National Certificate in Information Technology		✓	
Edexcel BTEC Level 3 National Subsidiary Diploma in Information Technology		✓	
Edexcel BTEC Level 3 National Diploma in Information Technology		✓	
Edexcel BTEC Level 3 National Extended Diploma in Information Technology		✓	
Edexcel BTEC Level 3 National Diploma in IT (Business)		✓	
Edexcel BTEC Level 3 National Extended Diploma in IT (Business)		✓	
Edexcel BTEC Level 3 National Diploma in IT (Networking and System Support)		✓	
Edexcel BTEC Level 3 National Extended Diploma in IT (Networking and System Support)		✓	
Edexcel BTEC Level 3 National Diploma in IT (Software Development)		✓	
Edexcel BTEC Level 3 National Extended Diploma in IT (Software Development)	✓		

There are specific links to the following units in the scheme:

Unit 6 – Software design and development
Unit 15 – Object-oriented programming
Unit 16 – Procedural programming
Unit 22 – Developing computer games

Achieving Success

In order to achieve each unit you will complete a series of coursework activities. Each time you hand in work, your tutor will return this to you with a record of your achievement.

This particular unit has 12 criteria to meet: 6 Pass, 4 Merit and 2 Distinction.

For a Pass:	You must achieve all 6 Pass criteria.
For a Merit:	You must achieve all 6 Pass and all 4 Merit criteria.
For a Distinction:	You must achieve all 6 Pass, all 4 Merit and 2 Distinction criteria.

Further reading

Balena, F. – *Programming Microsoft Visual Basic 6* (Microsoft Press US, 1999) ISBN-10: 0735605580, ISBN-13: 978-0735605589

Bond, M., Law, D., Longshaw, A., Haywood, D. and Roxburgh, P. – *Sams Teach Yourself J2EE in 21 Days, 2nd Edition* (Sams, 2004) ISBN-10: 0672325586, ISBN-13: 978-0672325588

Palmer, G. – *Java Event Handling* (Prentice Hall, 2001) ISBN-10: 0130418021, ISBN-13: 978-0130418029

Longshaw, J. and Sharp, J. – *Visual J#.NET Core Reference* (Microsoft Press US, 2002) ISBN-10: 0735615500, ISBN-13: 978-0735615502

Suddeth, J. – *Programming with Visual Studio.NET 2005* (Lulu.com, 2006) ISBN-10: 1411664477, ISBN-13: 978-1411664470

Troelsen, A. – *Pro C# 2005 and the.NET 2.0 Platform, 3rd Edition* (Apress US, 2004) ISBN-10: 1590594193, ISBN-13: 978-1590594193

Websites

eventdrivenpgm.sourceforge.net

www.vbwm.com

www.vbexplorer.com/VBExplorer/VBExplorer.asp