NEAR EAST UNIVERSITY

FACULTY OF ENGINEERING

DEPARTMENT OF COMPUTER ENGINEERING

Graduation Project

COM - 400

TIMETABLE SCHEDULING PROGRAM WITH DELPHI

STUDENT NO : 20010600

STUDENT NAME : Muhammed Akün

SUPERVISOR : Mr. Ümit İlhan

Nicosia-2006

ACKNOWLEDGEMENTS

I would like to acknowledge my parents and siblings first who has gave me all efforts and opportunities. They are never give-up supporting and encouraging me for years. The truth of the matter is that they deserve more than this little thanks.

Second, I would like to thank to my supervisor Mr. Ümit İLHAN for his great guidance, support and encouragements to accomplish this mission. Under the guidance of him, I have been able to overcome all difficulties, and accomplished all requirements.

I also would like to state appreciates and thanksgiving to the all academic and nonacademic Near East University personnel, for supplying all opportunities to us without grudging anything that they can do.

And I would like to thank to all instructors who spend their all efforts to teach something to us without being tired of. Their contribution cannot be omitted on the academic knowledge that I reached.

And I also would like to thank my friends Caner Çakır and Aykut Danışman too. They forced me to do my best and they have been helped me all my study of under graduate at this University, their helps on my graduation project also cannot be despised.

At last my special thanks for my home mates who were never loose the understanding to me. They do also all housework instead of me in my turn, and they become a companion to me for four years.

ABSTRACT

The main objective of this project can be explained as analyzing and producing solutions for complex activities which require special algorithmic design. In this project, complexity of creating timetables is tried to be reduced by combining specially designed algorithm with software application.

Preparing timetable takes long time. Actually spending lots of time does not solve the problem at all. Everything begins after listing the timetable; someone asks for a change, other has a course clash, but changing is not seem to be possible because changing any course will cause another clash or may be not seem to be good for whom course is already changed.

To put over a solution for this problem, an application program designed with using Delphi. By computer made work, the clashes will appear no more, and this process takes a few minute instead of hours, and also people have a block of idea in their mind that if any job is done with using computerized program, then the result is accepted as it is. And because it is automatically formed problem, it is the most appropriate solution for concerned problem.

Before writing codes, the requirements are designated first, and according to these requirements an appropriate relational database designed and then passed to the program design. While developing the program two things are held always up. The first one is that the program should be effective and should create a reliable result to the user, and the other thing is about appearance of the interface. Well designed user friendly interfaces make people use the programs willingly.

TABLE OF CONTENTS

TABLE OF TABLES

TABLE OF FIGURES

INTRODUCTION

Automation programs have been take palace in every aspect of life by the improvement of the computer technology. They become the most considerable applications at all. This project is an example of the automation programs with its general usage area, and it is also an example of developing windows based applications with specially designed algorithm embedded in it.

The problem underlying in this topic's background is very complicated. There are lots of courses offered, may be more than hundred, but on the other hand there are lots of limitations such as courses of same department can not be replaced to the same period, or courses of same year and term must be replaced to different periods. Another limitation is Lecturer cannot offer more than one course at one period as usual. These requirements can be increased by thinking on the problem just for a while. The complexity is tried to be reduced by developing related algorithms.

There may be lots of solution created for this problem. This is just an example to improve myself just before graduating from university. If it is looked at, everything is valuable for its creator and if lots of time spent on it much, point of view then this program also worth discussing for me although there are many things to do on it.

The technology 'which is Borland Delphi 7' is explained in the first chapter. Its IDE introduced to the readers. Basic rules of creating an application and projects are also mentioned. There is sufficient enough information about how to compile build or debug a project. And also general information, properties, and events of all components which are used in this project can be found in this document.

In the second chapter what is explained can be divided into two categories. One of these categories is brief information about designing a database and transaction between tables. The main operations applicable while dealing with database tables and the relationships also included to the document. Key feature of database and basic operators are also explained briefly. On the other hand second category of this chapter is Microsoft Access. This relational database management system is used for designing database, and writing the designed feature on flat tables. If entire chapter is dealt, then database concept and the Microsoft Access will be clearly understood.

The third chapter is about database concept of Delphi. The connection types BDE, dBase, and ADO are described. ADO connection is studied more, because the program uses this connection type.

Last chapter is about the program. It describes the program step by step. This chapter explains the solutions for the problem. The illustrations of code cores and interfaces are added to the text when necessary.

CHAPTER ONE

BASICS OF BORLAND DELPHI 7

1.1. Developing applications with Delphi environment

Borland Delphi is an object-oriented, visual programming environment to develop 32-bit applications for deployment on Windows and Linux. Using Delphi, you can create highly efficient applications with a minimum of manual coding.

Delphi provides a suite of Rapid Application Development (RAD) design tools, including programming wizards and application and form templates, and supports object-oriented programming with a comprehensive class library that includes:

- The *Visual Component Library* (VCL), which includes objects that encapsulate the Windows API as well as other useful programming techniques (Windows).
- The *Borland Component Library for Cross-Platform* (CLX), which includes objects that encapsulate the Qt library (Windows or Linux).

1.2. Editions of Delphi

Before delving into the details of the Delphi programming environment, let's take a side step to underline two key ideas. First, there isn't a single edition of Delphi; there are many of them. Second, any Delphi environment can be customized. For these reasons, Delphi screens may differ. Here are the current editions of Delphi:

- The "Personal" edition is aimed at Delphi newcomers and casual programmers and has support for neither database programming nor any of the other advanced features of Delphi.
- The "Professional Studio" edition is aimed at professional developers. It includes all the basic features, plus database programming support (including ADO support), basic web server support (WebBroker), and some of the external

tools, including ModelMaker and IntraWeb. This book generally assumes you are working with at least the Professional edition.

- The "Enterprise Studio" edition is aimed at developers building enterprise applications. It includes all the XML and advanced web services technologies, CORBA support, internationalization, three-tier architecture, and many other tools. Some chapters of this book cover features included only in Delphi Enterprise; these sections are specifically identified.
- The "Architect Studio" edition adds to the Enterprise edition support for Bold, an environment for building applications that are driven at run time by a UML model and capable of mapping their objects both to a database and to the user interface.

1.3. An Overview of the Delphi's IDE

When you start Delphi, you are immediately placed within the integrated development environment, also called the IDE. This IDE provides all the tools you need to design, develop, test, debug, and deploy applications, allowing rapid prototyping and a shorter development time. The general appearance of the Delphi's IDE is as shown in figure1.1 The IDE includes all the tools necessary to start designing applications, such as the:

- Form Designer, or *form*, a blank window on which to design the user interface (UI) for your application.
- Component palette for displaying visual and non-visual components you can use to design your user interface.
- Object Inspector for examining and changing an object's properties and events.
- Object TreeView for displaying and changing components' logical relationships.
- Code editor for writing and editing the underlying program logic.
- Project Manager for managing the files that makes up one or more projects.
- Integrated debugger for finding and fixing errors in your code.
- Many other tools such as property editors to change the values for an object's property.
- Command-line tools including compilers, linkers, and other utilities.

- Extensive class libraries with many reusable objects. Many of the objects provided in the class library are accessible in the IDE from the Component palette. By convention, the names of objects in the class library begin with a T, such as *TStatusBar*.

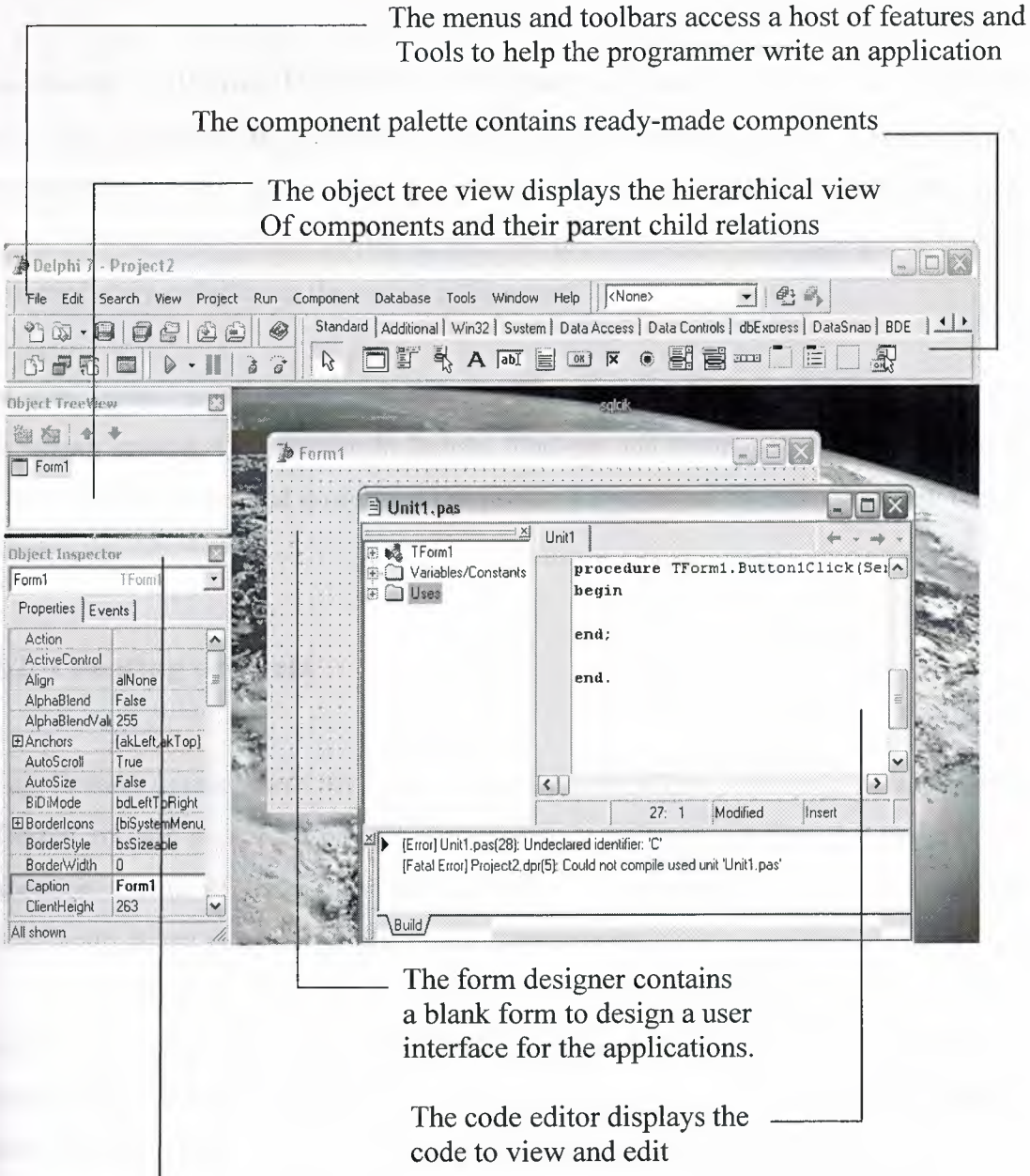


Figure 1.1 The view of IDE with simple form

1.3.1. Designing applications

You can design any kind of 32-bit application from general purpose utilities to sophisticated data access programs or distributed applications.

As you visually design the user interface for your application, the Form Designer generates the underlying Delphi code to support the application. As you select and modify the properties of components and forms, the results of those changes appear automatically in the source code, and vice versa. You can modify the source files directly with any text editor, including the built-in Code editor. The changes you make are immediately reflected in the visual environment.

You can create your own components using the Delphi language. Most of the components provided are written in Delphi. You can add components that you write to the Component palette and customize the palette for your use by including new tabs if needed.

1.3.2. Creating projects

All application development revolves around projects. When you create an application in Delphi you are creating a project. A project is a collection of files that make up an application. Some of these files are created at design time. Others are generated automatically when you compile the project source code.

You can view the contents of a project in a project management tool called the Project Manager. The Project Manager lists, in a hierarchical view, the unit names, and the forms contained in the units, and show the paths to the files in the project.

Project files, which describe individual projects, files, and associated options, have a .dpr extension. Project files contain directions for building an application or shared object. When you add and remove files using the Project Manager, the project file is updated.

Units and forms are the basic building blocks of an application. A project can share any existing form and unit file including those that reside outside the Project directory tree. If you add a shared file to a project, realize that the file is not copied into the current project directory; it remains in its current location. Adding the shared file to the current project registers the file name and path in the **uses** clause of the project file. Delphi automatically handles this as you add units to a project.

1.3.3. Editing code

The Code editor is a full-featured ASCII editor. If using the visual programming environment, a form is automatically displayed as part of a new project. You can start designing your application interface by placing objects on the form and modifying how they work in the Object Inspector. But other programming tasks, such as writing event handlers for objects, must be done by typing the code.

The contents of the form, all of its properties, its components, and their properties can be viewed and edited as text in the Code editor. You can adjust the generated code in the Code editor and add more components within the editor by typing code. As you type code into the editor, the compiler is constantly scanning for changes and updating the form with the new layout. You can then go back to the form, view and test the changes you made in the editor, and continue adjusting the form from there.

1.3.4. Compiling applications

When you have finished designing your application interface on the form and writing additional code so it does what you want, you can compile the project from the IDE or from the command line.

All projects have as a target a single distributable executable file. You can view or test your application at various stages of development by compiling, building, or running it:

- When you compile, only units that have changed since the last compile are recompiled.
- When you build, all units in the project are compiled, regardless of whether they have changed since the last compile. This technique is useful when you are unsure of exactly which files have or have not been changed, or when you simply want to ensure that all files are current and synchronized.
- When you run, you compile and then execute your application. If you modified the source code since the last compilation, the compiler recompiles those changed modules and re-links your application.

1.3.5. Debugging applications

With the integrated debugger, you can find and fix errors in your applications. The integrated debugger lets you control program execution, monitor variable values and items in data structures, and modify data values while debugging. The integrated debugger can track down both runtime errors and logic errors.

1.4. The Component Library

1.4.1. What is a component?

Components are the building blocks of Delphi applications. Although most components represent visible parts of a user interface, components can also represent non-visual elements in a program, such as timers and databases.

There are three different levels at which to think about components: a functional definition, a technical definition, and a practical definition.

If we deal the functional definition of 'component' from the ordinary user's perspective, a component is something to choose from the palette and use in an application by manipulating it in the Forms Designer or in code. From the expert writer's perspective, however, a component is an object in code.

The technical definition of 'component' can be explained as; a component is any object descended from the type *TComponent*. *TComponent* defines the most basic behavior that all components must have, such as the ability to appear on the Component palette and operate in the Forms Designer.

On the other hand when we look at the component on practical definition, a component is any element that can plug into the Delphi development environment.

1.4.2. Understanding the Component Library

The component library includes the Visual Component Library (VCL) and the Borland Component Library for Cross-Platform (CLX). The VCL is for Windows-only development and CLX is for cross-platform development on both Windows and Linux. The component library is extensive, containing both components that you can work with in the IDE and classes that you create and use in runtime code. Some of the classes can be used in any application, while others can only appear in certain types of applications. The component library is made up of objects separated into several sub-libraries, each of which serves different purposes. (These sub-libraries are BaseCLX, DataCLX, NetCLX, VisualCLX, and WinCLX).

Use the VCL when you want to use native Windows controls, Windows-specific features, or extend an existing VCL application. Use CLX when you want to write a cross-platform application or use controls that are available in CLX applications.
All classes descend from *TObject*. *TObject* introduces methods that implement fundamental behavior like construction, destruction, and message handling.
Components are a subset of the component library that descends from the class *TComponent*. You can place components on a form or data module and manipulate them at design time. Using the Object Inspector, you can assign property values without writing code (see the figure 1.2). Most components are either visual or non-visual, depending on whether they are visible at runtime.

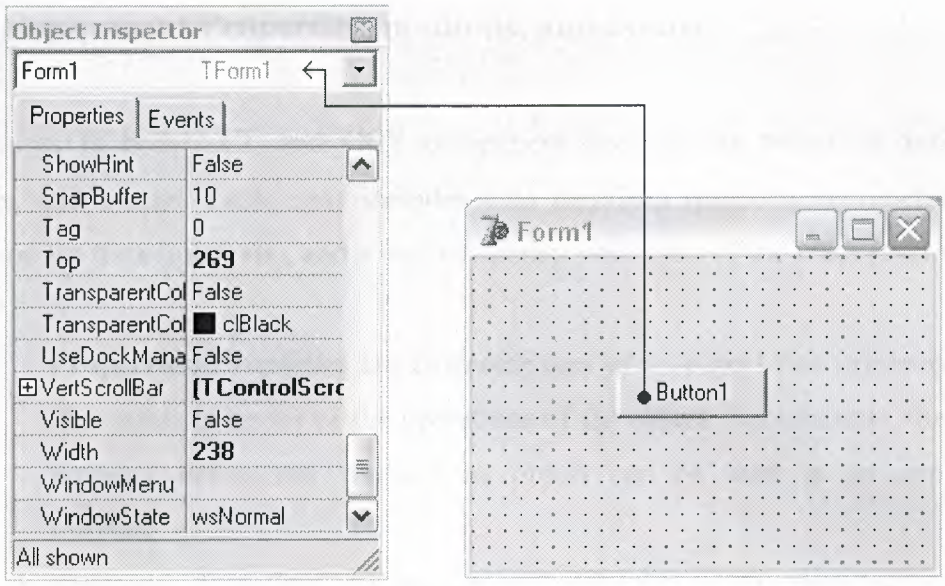


Figure 1.2 Tcomponent properties & object inspector

Visual components, such as *TForm* and *TSpeedButton*, are called *controls* and descend from *TControl*. Controls are used in GUI (Graphical User Interface) applications, and appear to the user at runtime. *TControl* provides properties that specify the visual attributes of controls, such as their height and width.

Non-visual components are used for a variety of tasks. For example, if you are writing an application that connects to a database, you can place a *TDataSource* component on a form to connect a control and a dataset used by the control. This connection is not visible to the user, so *TDataSource* is non-visual. At design time, non-visual components are represented by an icon. This allows you to manipulate their properties and events just as you would a visual control. Some components appear on the Component palette. (See the figure1.3 below).

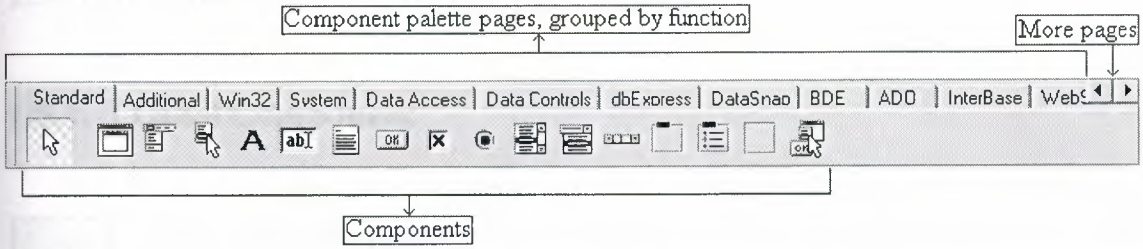


Figure 1.3 Component Palettes.

1.4.3. Component Properties, methods, and events

The classes in both VCL and CLX component libraries are based on properties, methods, and events. Each class includes data members (properties), functions that operate on the data (methods), and a way to interact with users of the class (events).

i. **Properties:** *Properties* are characteristics of an object that influence either the visible behavior or the operations of the object. For example, the *visible* property determines whether an object can be seen in an application interface.

ii. **Methods:** A *method* is a procedure that is always associated with a class. Methods define the behavior of an object. Class methods can access all the *public*, *protected,* and *private* properties and fields of the class and are commonly referred to as member functions.

iii. **Events:** An *event is* an action or occurrence detected by a program. Most modern applications are said to be event-driven, because they are designed to respond to events. You can write code to handle the events in which you are interested, rather than writing code that always executes in the same restricted order. The kinds of events that can occur can be divided into three main categories: a. User events are actions that the user initiates. Examples of user events are *OnClick.* b. System events are events that the operating system fires for you. For example, the *OnTimer* event. c. Internal events are events that are generated by the objects in your application. An example of an internal event is the *OnPost* event.

1.5. Common Components in Detail

1.5.1. TLabel Component

The *TLabel* component is a non-windowed control that displays text on a form. Usually this text labels some other control. The text of a label is value of its *Caption* property. Within caption, you can include an accelerator key.

How the text of the caption aligns within the label is determined by the value of the *Alignment* property. You can have the label resize automatically to fit a changing caption if you set the *AutoSize* property to *True*. If you prefer to have the text wrap, set *WordWrap* to *True*.

In addition to these properties, methods, and events, this component also has the properties and methods that apply to all controls.

TLabel Component Properties

Align	ComponentIndex	Font	Parent	ShowHint
Alignment	Cursor	Height	ParentColor	Tag
AutoSize	DragCursor	Hint	ParentFont	Top
BoundsRect	DragMode	Left	ParentShowHint	Transparent
Caption	Enabled	Name	PopupMenu	Visible
Color	FocusControl	Owner	ShowAccelChar	Width

TLabel Component Methods

BeginDrag	EndDrag	Invalidate	ScreenToClient	Show
BringToFront	GetTextBuf	Refresh	SendToBack	Update
ClientToScreen	GetTextLen	Repaint	SetBounds	
Dragging	Hide	ScaleBy	SetTextBuf	

TLabel Component Events

OnClick	OnDragDrop	OnDragOver	OnMouseMove	OnEndDrag
OnDblClick	OnMouseUp	OnMouseDown		

1.5.2. TEdit component

Edit boxes are used to retrieve information from the user, because the user can type data into an edit box. Edit boxes can also display information to the user. When users enter data into an edit box or the application displays information

to the user in the edit box, the value of the edit box's *Text* property changes. To limit the number of characters users can enter into the edit box, use the *MaxLength* property.

If you want to prevent the user from changing the value of the *Text* property, set the *ReadOnly* property to *True*.

You can cut, copy, and paste text to and from an edit box using the *CutToClipboard*, *CopyToClipboard*, and *PasteFromClipboard* methods.

In addition to these properties, methods, and events, this component also has the Properties, methods, and events that apply to all windowed controls.

TEdit component Properties

Align	DragCursor	MaxLength	ParentShowHint	TabOrder
AutoSelect	DragMode	Modified	PasswordChar	TabStop
AutoSize	Enabled	Name	PopupMenu	Tag
BorderStyle	Font	OEMConvert	ReadOnly	Text
CharCase	Height	Owner	SelLength	Top
Color	HelpContext	Parent	SelStart	Visible
ComponentIndex	HideSelection	ParentColor	SelText	Width
Ctl3D	Hint	ParentCtl3D	ShowHint	
Cursor	Left	ParentFont	Showing	

TEdit component Methods

BeginDrag	Dragging	GetTextBuf	Repaint	ScreenToClient
BringToFront	EndDrag	GetTextLen	ScaleBy	SetFocus
ClientToScreen	Hide	GetSelTextBuf	SetBounds	SetSelTextBuf
Clear	Free	Invalidate	ScrollBy	SetTextBuf
ClearSelection		Refresh	SelectAll	Show
SendToBack	Update	FindComponent		

TEdit component Event

OnChange	OnDragOver	OnExit	OnKeyUp	OnMouseMove
OnDblClick	OnEndDrag	OnKeyDown	OnMouseDown	OnMouseUp
OnDragDrop	OnEnter	OnKeyPress		

1.5.3. TButton component

A *TButton* is a push button control. Users choose button controls to initiate actions. Buttons are most commonly used in dialog boxes.

A default button is the button whose *OnClick* event handler runs whenever the user presses the Enter key while using the dialog box. To make a button a default button, set the button's *Default* property to *True*.

In addition to these properties, methods, and events, this component also has the properties, methods, and events that apply to all windowed controls.

TButton component Properties

Enabled	Caption	HelpContext	Parent	TabOrder
DragMode	Cancel	Hint	ParentFont	TabStop
DragCursor	BoundsRect	Left	ParentShowHint	Tag
Default	Align	ModalResult	PopupMenu	Top
Cursor	Font	Name	ShowHint	Visible
ComponentIndex	Height	Owner	Showing	Width

TButton component Methods

Focused	BringToFront	Hide	ScreenToClient	SetFocus
EndDrag	BeginDrag	Refresh	ScrollBy	SetTextBuf
Dragging	GetTextBuf	Repaint	SendToBack	Show
CanFocus	GetTextLen	ScaleBy	SetBounds	Update

TButton component Events

OnEndDrag	OnClick	OnKeyDown	OnKeyUp
OnDragOver	OnEnter	OnKeyPress	OnMouseDown
OnDragDrop	OnExit	OnMouseUp	OnMouseMove

1.5.4. TComboBox component

A *TComboBox* component is a control that combines an edit box with a list, much like that of a list box. Users can either type text in the edit box or select an item from the list.

When users enter data into the combo box, either by typing text or selecting an item from the list, the value of the *Text* property changes. Your application can also change the *Text* property by displaying text for the user in the edit box of the combo box.

The list of items in the list is the value of the *Items* property. The *ItemIndex* property indicates which item in the list is selected.

You can add, delete, insert, and move items in the list using the *Add*, *Delete*, and *Insert* methods of the *Items* object. For example, to add a string to the list, you could write this line of code: ComboBox1.Items.Add('New item');
You can change the style of the combo box or make it an owner-draw control by changing the value of the *Style* property.

In addition to these properties, methods, and events, this component also has the properties, methods, and events that apply to all windowed controls.

TComboBox component Properties

Font	DragCursor	MaxLength	SelLength	Style
Enabled	Cursor	Items	SelStart	TabOrder
DropDownCoun	ComponentIndex	ItemIndex	SelTSelStartext	TabStop
DragMode	Parent	ItemHeight	Sorted	Visible

TComboBox component Methods

BeginDrag	Clear	Focused	Invalidate	SelectAll
BringToFront	Dragging	GetTextBuf	Refresh	SendToBack
CanFocus	EndDrag	GetTextLen	Repaint	Update

TComboBox component Events

OnChange	OnDragDrop	OnDropDown	OnDblClick	OnExit
OnClick	OnDragOver	OnEndDrag	OnKeyDown	OnKeyUp
OnEnter	OnDrawItem	OnMeasureItem	OnKeyPress	

1.5.5. TListBox component

The *TListBox* component is a Windows list box. A list box displays a list from which users can select one or more items.

The list of items in the list box is the value of the *Items* property. The *ItemIndex* property indicates which item in the list box is selected.

You can add, delete, and insert items in the list box using the *Add*, *Delete*, and *Insert* methods of the *Items* object. For example, to add a string to a list box, you could write this line of code: ListBox1.Items.Add('New item');

You can allow users to select more than one item at a time by setting the *MultiSelect* property to *True*. The *ExtendedSelect* property determines how multiple items can be selected. To determine whether a particular item is selected and how many items are selected, check the values of the *Selected* and *SelCount* properties, respectively.

You can make the list box an owner-draw list box by changing the *Style* property.
In addition to these properties, methods, and events, this component also has the properties, methods, and events that apply to all windowed controls.

TListBox component Properties

BorderStyle	*Cursor*	*ItemIndex*	*Parent*	*Style*
ComponentIndex	*DragMode*	*ItemHeight*	*SelCount*	*Visible*
ExtendedSelect	*Enabled*	*Item*	*Showing*	*Align*
IntegralHeight	*Canvas*	*TopIndex*	*Columns*	*Name*
MultiSelect	*Color*	*Selected*	*Sorted*	*Font*

TListBox component Methods

BeginDrag	EndDrag	ItemAtPos	ScaleBy	SetFocus
BringToFront	GetTextBuf	Invalidate	ScrollBy	SetTextBuf
Clear	GetTextLen	Refresh	SendToBack	Show
Dragging	Hide	Repaint	SetBounds	Update

TListBox component Events

OnKeyDown	OnDragOver	OnEnter	OnKeyPress	OnMouseDown
OnDblClick	OnDrawItem	OnExit	OnKeyUp	OnMouseMove
OnDragDrop	OnEndDrag	OnClick	OnMouseUp	OnMeasureItem

1.5.6. TMaskEdit component

A mask edit box is an much like an ordinary edit box (*TEdit* component), except you can require the user to enter only valid characters through the use of an *EditMask* property. You can also use the mask to format the display of data.

The text the user enters in the edit box is the value of the *Text* property, just as it is with any edit box. The text of the edit box with the mask specified in the *EditMask* property applied to it is the value of the *EditText* property.

User can cut, copy, and paste text to and from a mask edit box using the *CutToClipboard*, *CopyToClipboard*, and *PasteFromClipboard* methods.
In addition to these properties, methods, and events, this component also has the properties, methods, and events that apply to all windowed controls.

TMaskEdit component Properties

AutoSelect	DragCursor	Height	Owner	SelStart
AutoSize	DragMode	IsMasked	Parent	SelText
BorderStyle	EditMask	MaxLength	PasswordChar	Text
CharCase	EditText	Modified	ReadOnly	Top
Color	Enabled	Name	SelLength	Visible

TMaskEdit component Methods

BeginDrag	Dragging	Repaint	ScaleBy	SetSelTextBuf
BringToFront	EndDrag	Refresh	ScrollBy	SetTextBuf
CanFocus	GetSelTextBuf	Hide	SetFocus	Update
GetTextLen	GetTextBuf	Clear	SelectAll	ValidateEdit
Invalidate	SendToBack	Focused	SetBounds	ClearSelection

TMaskEdit component Events

OnChange	OnDragOver	OnDragDrop	OnExit	OnEndDrag
OnMouseMove	OnKeyPress	OnKeyDown	OnEnter	OnKeyUp
OnMouseDown	OnDblClick	OnMouseUp		

1.5.7. TMainMenu component

The *MainMenu* component encapsulates a menu bar and its accompanying drop-down menus for a form. To begin designing a menu, add a main menu component to your form, and double-click the component.

The items on the menu bar and in its drop-down menus are specified with the *Items* object, a property of a main menu. The *Items* object is of type *TMenuItem*. Your application can use the *Items* property to access a particular command on the menu.

You can choose to have the menus of one form merge with those of another using the *AutoMerge* property and the *Merge* and *Unmerge* methods.

In addition to these properties and methods, this component also has the properties and methods that apply to all components.

TMainMenu component Properties

AutoMerge Owner ComponentIndex Name Tag Items

TMainMenu component Methods

FindItem GetHelpContext Unmerge Free Merge

1.5.8. TDBGrid component

The *TDBGrid* component can access the data in a database table or query and display it in a grid. Your application can use the data grid to insert, delete, or edit data in the database, or simply to display it.

The most convenient way to move through data in a data grid and to insert, delete, and edit data is to use the database navigator (*TDBNavigator*) with the data grid.

The *Fields* property is an array of all the fields in the dataset displayed in the data grid. To determine which field is the currently selected field, use the *SelectedField* property. Use the *FieldCount* property to find out how many fields are in the dataset displayed in the data grid.

You can change the appearance and behavior of a data grid by changing the value of the *Options* property. For example, you can choose to allow the user to use the Tab key to move to a new column, or you can decide to display grid lines between columns, but not between rows.

If you want the user to be able only to view the data and not to edit it, set the *ReadOnly* property to *True*. If you want the user to be able to edit the data, set *ReadOnly* to *False*. Also, the dataset must be in Edit state, and the *ReadOnly* property of the data must be *False*. The user can cancel an edit by pressing Esc.

Users don't really insert or edit the data in a field using the data grid until they move to a different record or close the application.

In addition to these properties, methods, and events, this component also has the Properties, methods, and events that apply to all windowed controls.

TDBGrid component Properties

Fields	SelectedField	ClientOrigin	TopRow	DragCursor
Parent	DefaultDrawing	ClientRect	Options	DragMode
Font	SelectedIndex	ClientWidth	Visible	ParentFont
Enabled	FixedColor	DataSource	Cursor	EditorMode
ReadOnly	ClientHeight	FieldCount	Name	BorderStyle

TDBGrid component Methods

GetTextBuf	BringToFront	Dragging	Refresh	SetFocus
GetTextLen	SendToBack	EndDrag	Update	Focused
SetTextBuf	ScrollBy	Repaint	Invalidate	

TDBGrid component Events

OnColEnter	OnColExit	OnDblClick	OnKeyPress
OnDragOver	OnCell Click	OnEndDrag	OnDragDrop
OnExit	OnKeyDown	OnEnter	OnKeyUp

1.5.9. TDBNavigator component

The *TDBNavigator* component (a database navigator) is used to move through the data in a database table or query, and perform operations on the data, such as inserting a blank record or posting a record. It is used in conjunction with the data-aware controls, such as the data grid, which give you access to the data, either for editing the data, or for simply displaying it.

User link the database navigator with a dataset when a data source component

is specified, (identifies dataset as value of navigator's *DataSource* property.)

The database navigator consists of multiple buttons (see Figure).

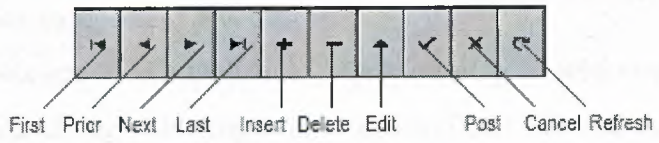


Figure 1.4 DBNavigator

When the user chooses one of the navigator buttons, the appropriate action occurs on the dataset the navigator is linked to. For example, if the user clicks the Insert button, a blank record is inserted in the dataset. This table describes the buttons on the navigator:

Button	Purpose
First	Sets the current record to the first record in the dataset, disables the First and Prior buttons, and enables the Next and last buttons if they are disabled
Prior	Sets the current record to the previous record and enables Last and Next buttons
Next	Sets the current record to the next record and enables the First and Prior buttons
Last	Sets the current record to the last record in the dataset, disables the Last and Next buttons, and enables the First and Prior buttons if they are disabled
Insert	Inserts a new record before the current record, and sets the dataset into Insert and Edit states
Delete	Deletes the current record and makes the next record the current record
Edit	Puts the dataset into Edit state so that the current record can be modified
Post	Writes changes in the current record to the database
Cancel	Cancels edits to the current record, restores the record display to its condition prior to editing, and turns off Insert and Edit states if they are active
Refresh	Redisplays the current record from the dataset, thereby updating the display of the record on the form

Table1.1 DBNavigatorButtnons

In addition to these properties, methods, and events, this component also has the properties, methods, and events that apply to all windowed controls.

TDBNavigator component Properties

ConfirmDelete DataSource Enabled Cursor Parent
PopupMenu DragCursor Height Align Visible
VisibleButtons DragMode Name

TDBNavigator component Methods

BtnClick Invalidate GetTextLen BringToFront Update
Refresh SetTextBuf GetTextBuf SendToBack Repaint

TDBNavigator component Events

OnExit OnResize OnDblClick OnMouseUp OnMouseMove
OnClick OnEndDrag OnDragOver OnDragDrop OnMouseDown
OnEnter

1.5.10. TDataSource component

TDataSource is the interface between a dataset component and data-aware controls on forms. *TDataSource* attaches to a dataset through the *Dataset* property. Data-aware controls, such as database edit boxes and data grids, attach to a *TDataSource* through their *DataSource* properties. Usually there is only one data source for each dataset component, but there can be as many data source components connected to a dataset as programmer needs.

The *Dataset* property identifies the dataset from which the data is obtained. Set the *AutoEdit* property to *False* to prevent the dataset from going into edit mode Automatically when the value of an attached data-aware control is modified (programmer can still call the *Edit* method to permit modifications). Set the *Enabled* property to *False* to clear and disable the data-aware controls. Check the current status of the dataset with the *State* property. To monitor changes to both the dataset and attached data-aware controls, assign a method to the *OnDataChange* event. To monitor changes in the dataset's state, assign a method to the *OnStateChange* event. To update the dataset prior to a post, assign a method to the *OnUpdateData* event.

TDataSource component Properties

AutoEdit Dataset Owner Enabled State
Name Tag

TDataSource component Methods

There is only a single method for TDataSourse is : Edit

TDataSource component Events

OnDataChange OnStateChange OnUpdateData

1.5.11. TADOTable Componet

TADOTable Component is an Encapsulation of a table. ADOTable is implemented almost entirely by its immediate ancestor class, TCustom-ADODataSet. This component provides the majority of dataset functionality, and its descendants are mostly thin wrappers that expose different features of the same component. As such, the component has a lot in common. In general, however, ADOTable is viewed as "compatibility" component and is used to aid the transition of knowledge and code from its BDE counterparts. Be warned, though: this compatibility component is similar to its counterparts but not identical.

TADOTable Componet Properties

Tag	MasterFields	AutoCalcFields	CacheSize	CursorLocation
Name	MasterSource	CommandTimer	CursorType	ExecuteOptions
Filtered	MaxRecords	ConnectionString	LockType	TableDirect
Fitler	IndexName	IndexFieldNames	ReadOnly	EnableBCD
Active	IndexFields	MaeshalOptions	TableName	

TADOTable Componet Events

AfterCancel	AfterOpen	BeforeClose	BeforePost	OnClick
AfterClose	AfterPost	BeforeDelete	BeforeRefresh	OnCloseQuery
AfterDelete	AfterRefresh	BeforeEdit	BeforeScroll	OnDestroy
AfterEdit	AfterScroll	BeforeInsert	OnCreate	OnKeyPress
AfterInsert	BeforeCancel	BeforeOpen	OnClose	

1.5.12. TADOQuery Component

The TADOQuery component provides Delphi developers the ability to fetch data from one or multiple tables from an ADO database using SQL. These SQL statements can either be DDL (Data Definition Language) statements such as CREATE TABLE, ALTER and so, or they can be DML (Data Manipulation Language) statements, such as SELECT, UPDATE, and DELETE. The most common statement, however, is the SELECT statement, which produces a view similar to that available using a Table component.

The SQL used in a ADOQuery component must be acceptable to the ADO driver in use. In other words you should be familiar with the SQL writing differences between, for example, MS Access and MS SQL.

As when working with the ADOTable component, the data in a database is accessed using a data store connection established by the ADOQuery component using its *ConnectionString* property or through a separate ADOConnection component specified in the *Connection* property.

To make a Delphi form capable of retrieving the data from an Access database with the ADOQuery component simply drop all the related data-access and data-aware components on it and make necessary links.

The TADOQuery component doesn't have a *TableName* property as the TADOTable does. TADOQuery has a property (TStrings) called *SQL* which is used to store the SQL statement. You can set the SQL property's value with the Object Inspector at design time or through code at runtime.

The (ordinary) SQL statements are executed by setting the *TADOQuery.Active* property to *True* or by calling the *Open* method (essentialy the same).

TADOQuery Component Properties

Active	ConnectionString	Tag	Filtered	MaxRecords
AutoCalcFields	CursorLocation	SQL	Prepared	ParamCheck
CacheSize	ExecuteOptions	Name	Parameters	DataSource
CursorType	MarshalOptions	Fitler	lockType	EnableBCD
Connection	CommandTimer			

TADOQuery Componet Events

AfterCancel	AfterOpen	BeforeClose	OnCloseQuery	OnCreate
AfterClose	AfterRefresh	BeforeDelete	OnDestroy	OnClose
AfterDelete	AfterScroll	BeforeInsert	OnKeyPress	OnClick
AfterEdit	BeforePost	BeforeOpen	BeforeCancel	AfterPost
AfterInsert	BeforeEdit	BeforeScroll	BeforeRefresh	

1.6. Classes and Objects in Delphi

Delphi is based on OOP concepts, and in particular on the definition of new class types. The use of OOP is partially enforced by the visual development environment, because for every new form defined at design time, Delphi automatically defines a new class. In addition, every component visually placed on a form is an object of a class type available in or added to the system library.

A *class*, or *class type*, defines a structure consisting of *fields*, *methods*, and *properties*. Instances of a class type are called *objects*. The fields, methods, and properties of a class are called its *components* or *members*.

- A field is essentially a variable that is part of an object. Like the fields of a record, a class's fields represent data items that exist in each instance of the class.
- A method is a procedure or function associated with a class. Most methods operate on objects that is, instances of a class. Some methods (called *class methods*) operate on class types themselves.

- A property is an interface to data associated with an object (often stored in a field). Properties have *Access specifiers*, which determine how their data are read and modified. From other parts of a program outside of the object itself a property appears in most respects like a field.

Objects are dynamically allocated blocks of memory whose structure is determined by their class type. Each object has a unique copy of every field defined in the class, but all instances of a class share the same methods. Objects are created and destroyed by special methods called *constructors* and *destructors*.

As in most other modern OOP languages (including Java and C#), in Delphi a class-type variable doesn't provide the storage for the object, but is only a pointer or reference to the object in memory.

A class type must be declared and given a name before it can be instantiated. (You cannot define a class type within a variable declaration.) Declare classes only in the outermost scope of a program or unit, not in a procedure or function declaration. A class type declaration has the form

```
type className = class (ancestorClass)
memberList
end;
```

Where *className* is any valid identifier, (*ancestorClass*) is optional, and *memberList* declares members that is, fields, methods, and properties of the class. If you omit (*ancestorClass*), then the new class inherits directly from the predefined *TObject* class. Methods appear in a class declaration as function or procedure headings, with no body. Defining declarations for each method occur elsewhere in the program.

As an alternative to class types, *object types* can be declared using the syntax

```
type objectTypeName = object (ancestorObjectType)
memberList
end;
```

where *objectTypeName* is any valid identifier, (*ancestorObjectType*) is optional, and *memberList* declares fields, methods, and properties. If (*ancestorObjectType*) is omitted, then the new type has no ancestor. Object types cannot have published members.

Since object types do not descend from *TObject*, they provide no built-in constructors, destructors, or other methods. Instances of an object type can be created using the *New* procedure and destroy them with the *Dispose* procedure, it can be simply declared variables of an object type.

CHAPTER TWO

MICROSOFT ACCESS DATABASE

2.1. What is a Database?

The term "database" has been applied in a number of different ways, many specific to the development context in which the word is mentioned. Ignoring the marketing driven terminology for the moment, a *database* is best described as simply a collection of related data. The relationship is defined by some natural or forced affinity between the items, or records, that make up the collection.

A computerized database is an electronic store (representation) of data. It is a repository for electronically storing data records, each record being a set of the individual data elements that describe each item that the records are modeled upon.

2.2. Structure of Database

- A person, place, event, or item is called **entity**.
- The facts describing an entity are known as **data**. (For example a registrar in a college would like to have all the information about the students. Each student is an entity in such a scenario.).
- Each entity can be described by its characteristics, which are known as **attributes.** (For example some of the attributes in a college student database are name, number, phone etc.).
- All the related entities are collected together to form an **entity set**. An entity set given singular name.
- Collection of entity sets is called a **database.**
- The entities in a database likely to interact with other entities. The interactions between entity sets are called **relationships.** Relationships can

be grouped into three categories; a) one-to-one relationship, b) one-to-many relationship, c) many-to-many relationship.

2.3. The Relational Database Model

The database is everywhere in modern information processing; data is stored in collections of some type in nearly every application of note. It was the automation of huge collections of data and the commensurate ability to sort, search, and maintain those records that led to the advancement, at breakneck speed, of the powerful computer hardware that has become common today. Data is power, and organizations have come to recognize that the ability to harness these information resources is not only a competitive edge, but critical to their survival.

The need for a data is always present. In the computer age, the need to represent data in an easy-o understand, logical form has led to many models, such as the relational model, the hierarchical model, the network model, and the object model. Because of its simplicity in design and ease in retrieval of data, the relational database model has been very popular, especially in the personal computer environment.

The relational model which is developed by the E.F.Codd in 1970 is based on mathematical set theory; it uses relation as the building block of the database. The **relation** is represented by a two-dimensional, flat structure known as **table.** Users do not have to know about mathematical details or physical aspects of the data. Microsoft Access 2003 is one of the examples of the Relational Database Management Systems.

Let us have looked at the basic relational database terminology;

- A row referred to as a **tuple.**
- The number of columns in a table is called the **degree** of the relation.
- The set of all possible values that a column may have is called the **domain** of that column.
- A **key** is a minimal set of columns used to uniquely define any row in a table.
- When a single column is used as a unique identifier, it is known as a **primary key**.

- When a combinations of columns is used as a unique identifier, it is known as a **composite primary key** or, simply, as a **composite key.**
- In a relational database, tables are related to each other through a common column. A column in a tablet hat references a column in another table is known as a **foreign key**.

Microsoft Access 2003 follows that the data underlying the tables be consistent. If consistency is compromised, the data are not usable. For this reason Microsoft Access 2003 strictly follows the Entity integrity rule which means that no column in a primary key be null. The primary key provides the means of uniquely identifying a row or an entity. A null value means a value that is not known, not entered not defined, or not applicable. A zero or a space is not considered to be a null value. That is why Microsoft Access 2003 does not allow users to enter a row without unique value in the primary key. Microsoft Access 2003 also support the referential integrity rule hat means a foreign key value ay be null value, or it must be exist as a value of primary key in the referenced table.

2.4. Data Manipulation

Another aspect of the relational database model concerns itself with the manipulation of data. The model defines two categories of operations that can be performed using the Relations:

i. Assignment of relations to other relations
ii. Manipulation of the data using eight defined operators

Both categories are in reality intertwined. Data manipulation operations such as Select result in the selected data being placed into a new table.

The eight relational database operators (stored, Select, Project, Product, Join, Union, Intersection, Difference, Division) share two characteristics. First, the relational operators are set processing commands; they apply to and result in relational tables. The second characteristic is that the operators are unaffected by how the data is physically

- **Select:** The *select* operation retrieves a set of rows into a new relation. This set is composed of rows in the base relation in which the column values match the criteria provided in the query.
- **Project:** The *project* operator retrieves a subset of columns from a relational table, placing them into a new relation. In the process, it also removes duplicate rows from the result.
- **Product:** The *product* operation puts two rows from separate tables together in the resultant table. The new relation is now twice the column width of the original base tables.
- **Join:** The *join* operation combines the Product and Select operations to produce the new relation.
- **Union:** The *union* operation vertically combines the data in the rows of one relation with the rows in another table, removing duplicate rows in the resulting table.
- **Intersection:** The *intersection* of two tables is a relation containing those rows that are common to both tables. The intersection operator evaluates the contents of matching columns in each table to determine if the criteria are a match.
- **Difference:** The *difference* of two tables is a relation that contains those rows that exist in one of the two tables but not the other.
- **Division:** The *division* operator results in a relation that contains column values from one table for which there are other matching column values corresponding to every row in another table. In other words, a relation is going to be divided by another relation with the quotient being a new relation.

2.5. Database Design

As with any programming effort, taking the time to plan and create a proper design pays off when the rubber hits the road and the time comes to develop a database application. The application creation process is simplified, as numerous problems and issues have already been addressed up front, before the first Begin statement is coded. If it is not carefully considered while spending time on designing database up front, then the price

will be paid and programming effort needed to make it right in stability and accuracy. While designing a database basic three steps are followed;

First, the design for the database should be mapped through the creation of entity-relationship diagrams, a standard tool that makes the database members clear and puts the information into a format that can be quickly converted into relations.

Secondly, the process of normalizing the relations described in the ER diagrams should be examined in detail. *Normalization* is the process by which the relations that have designed are tested against the rules of the relational database, and through multiple design iterations they are manipulated into place.

Finally, the process of mapping the design to a physical data structure should be explored, setting up the design to be converted to a physical implementation.

2.5.1. Diagramming the Database

To arrive at any destination, the most efficient process is to follow a map. Developing a database is no different. The mapping used for this type of development effort is called an *entity-relationship (ER) diagram*; it's a graphical representation of all of the items that will be contained in the database. The diagram completely describes the database to the level of detail necessary to transfer the logical design directly to a physical implementation.

In nearly every instance, the diagram will be composed of representations of the following items:

- **Entities:** Finding entities or in other words finding the players are those people, places, or things about which are wanted to record facts in the database. Then the positive attribute of these entities should be found described.
- **Relationships:** After determining and identifying the entities and their attributes there should also be some idea of forming about how they relate to one another.

The relationship describes transactions, communications, and ownership between the entities.

- **Primary keys:** The singular nature of items in a relational database system is the key concept that drives the paradigm. The main activity taken in this part of the design process is to identify an attribute or a minimal set of attributes that can be used to uniquely identify a record to become the primary key. Sometimes it may become necessary to introduce a new attribute to the relation if a single attribute or a set of attributes (a composite key) cannot be identified.
- **Alternate keys:** The candidate keys that were not selected as the primary keys become *alternate keys.* The purpose of identifying the alternate keys is to provide substitute access paths.
- **Foreign keys:** A *foreign key* serves a critical purpose in a relationship. It is an attribute or set of attributes that identifies the parent record. In other words, the foreign key is the attribute that links the child occurrences to the parent entity occurrence through matching key values. The foreign key is artificially present in the child entity and is the primary key of the parent.

In reviewing the ER diagram, it is easy to see that the two most critical components of the database model are the entities and the relationships between them. Before applying these diagramming tools, establishing some definitions is in order.

2.5.2. Normalization Designed Database

Normalization is the process of decomposing relations to ensure maximum stability and minimal data redundancy. The process is one in which relations and their structures are refined in such a way that no data is lost and no artificial structures are introduced. A fully normalized relation is one that most closely matches guidelines of the relational model and exhibits correctness, consistency, stability, and non-redundancy.

A table is said to be in normal form when its structure and data meet the requirements of one of the stages of normalization. There are stages labeled first through fifth normal form, Boyce/Codd normal form, and domain-key normal form.

First three steps are sufficient for most applications but there may be still the possibility of some specific anomalies being found in the database. If it is so then the 4NF, 5NF, Boyce/Codd normal form, and domain-key normal form can be applicable.

i. First Normal Form (1NF): The table is said to be in first normal form, or can be labeled 1NF, if the primary keys, or composite keys are defined and all non-key columns show functional dependency on the primary key components. In other words to be in first normal form, a relation must have no repeating groups or multi-valued attributes.

ii. Second Normal Form (2NF): Second normal form further refines the database structures. To be in 2NF a table must in 1NF and all of the attributes must be fully dependent upon the whole primary key. Every non-key attribute must be fully dependent upon the primary key.

iii. Third Normal Form (3NF): Third normal form is achieved by first massaging the relation into second normal form. In 3NF, each non-key attribute must now be fully dependent upon the whole primary key.

2.5.3. Translating the Logical Design into a Physical Design

The final step in all of this work is to create the tables that will make up the database. The entities and their attributes will directly map to a table structure and, if the design and normalization steps were carefully applied it will be end up with a stable, simple, optimal database. There are two steps remaining in the process: determining the data types of the structure of the table and creating the field and file names.

Each attribute that is identified will become a field in the table. In creating a field in a table, some new aspects of the attribute must be considered. The name, data type, and size of the field are the most critical elements of any field.

2.6. Introduction to Microsoft Access 2003

Microsoft Access 2003 is a Relational Database Management System (RDMS) that allows a user to store, organize, and manipulate collections of information in an

electronic format. A database is a collection of related information or data. The basic file type in Access is database, which uses the extension *.mdb.

The components that make up the Microsoft Access 2003 are as follows;

- The D*atabase Engine* is the (generally invisible) software that actually stores, indexes, and retrieves data. When standalone database is created, Access uses the jet engine to manage data.
- *Database objects* provide users the interface to view, enter, and extract information from a database.
- Access includes a full set of *design tools* that the users to create objects.
- Access includes a rich set of *programming tools* that users can automate routine task.

The basic building blocks of Access applications are known as database objects. There are six basic database objects in Access tables, queries, forms, reports, macros and modules. All Access database applications are built from these database objects. Tables, queries, forms and reports have properties that govern their behaviors and their general appearance.

2.7. Microsoft Access 2003 Tables

A table is the basic unit for storing a collection of data in an Access database. A table's definition consists of list of *fields*, each of which stores a discrete piece of information for a single *record.*

When the database design process is applied to Microsoft Access 2003, it follows the hierarchy illustrated on figure 2.1 below;

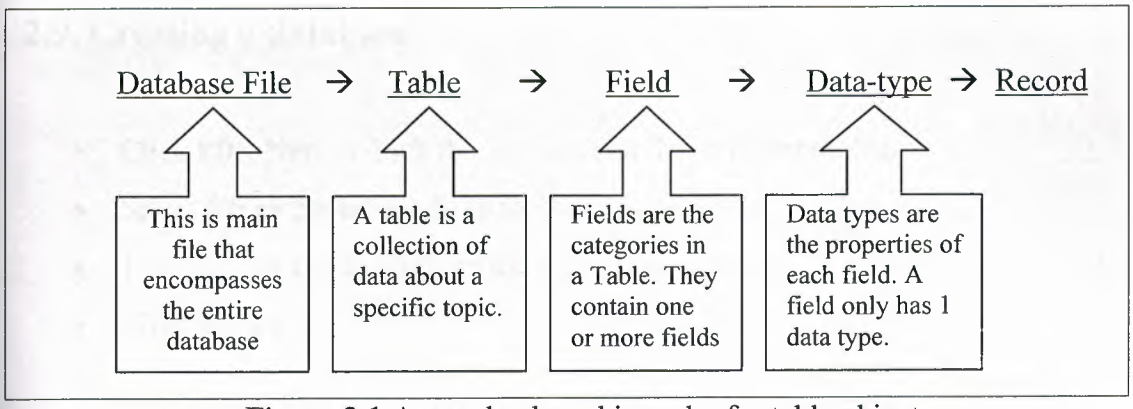


Figure 2.1 Acces database hierarchy for table objects

2.8. Opening Access

- Double click on the Microsoft Access 2003 shortcut icon on the desktop.

Figure 2.3 Opening Access from Shortcut

- Click **Start**, select **All Programs** and click on **Microsoft Access**

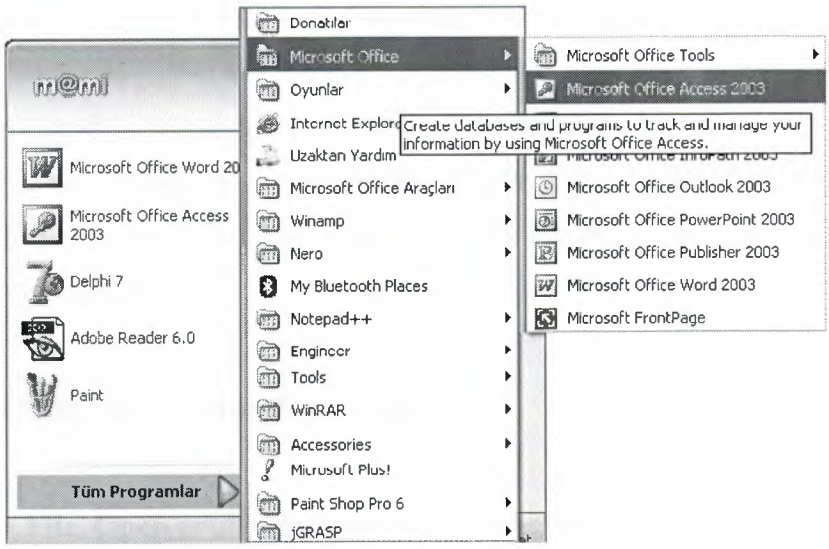


Figure 2.2 Opening Access from start menu

2.9. Creating a database

- Click **File**, **New** or click the *new icon* on the standard toolbar
- Select **Blank Database** from the *Task Pane* menu
- Type a name for database in the *File Name* window
- Click **Create**

The figure 2.4 below illustrates the creating database application;

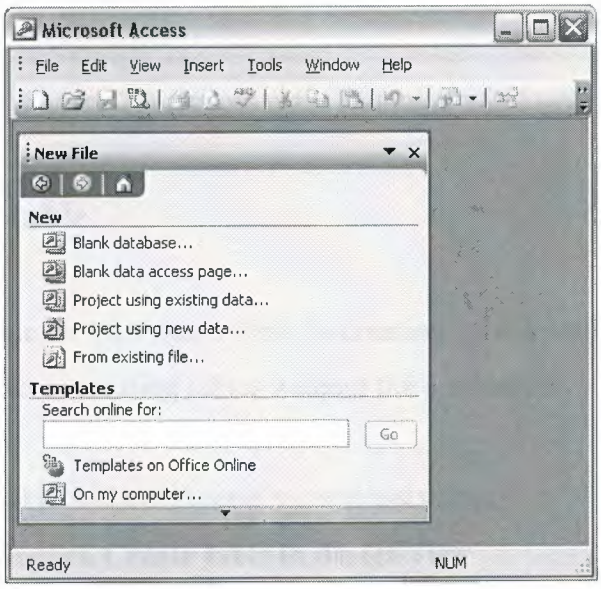


Figure 2.4 Creating a Database

Database is already created to specified address location by the user, as far as the database is saved Access is automatically displays the dialog box for the related database object. The user can either close the newly created database or create an object (example: Table object).

2.10. Opening an existing Database

- Click **File**, **Open** or click the *open icon* on the standard toolbar
- Browse to where the database is saved
- Click the name of the database
- Click **Open**

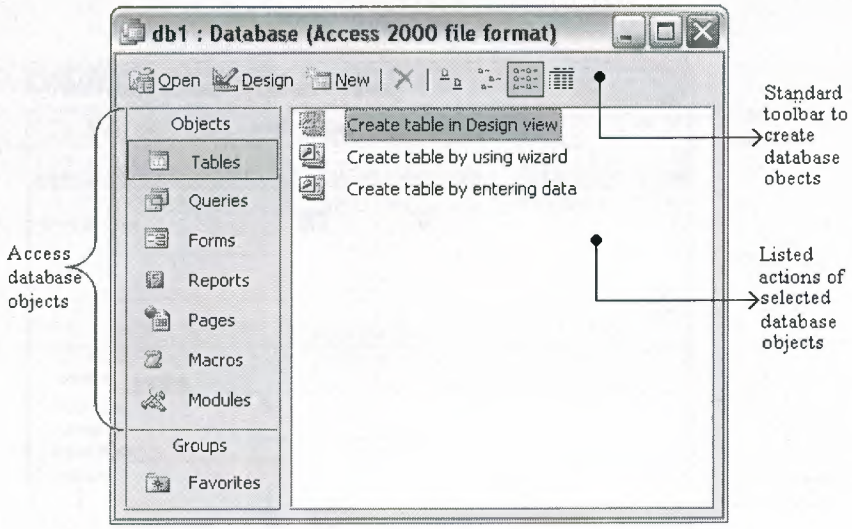


Figure 2.5 Access main Database windows.

2.11. Creating a table

Users can create table in two ways first is creating tables with the Access wizard assistance and the other is creating tables without the assistance.

i. Creating table without Access wizard assistance

- Double-click **Create table in design view**
- In the *Field Name* column type the name of data field (i.e. FirstName)
- In the *Data Type* column select the type of data to be entered in the field (i.e. Text)
- Complete steps second and third for all other data fields (i.e. LastName, Address, etc.)
- Save the table. Click **File**, **Save** or click the *save icon* on the Standard Toolbar.
- Type the name for the table.
- Click **Ok**.
- If you do not want a primary key, click no.
- Click **View**, **Datasheet View** or click the *view icon* on the Standard Toolbar to begin entering data in the table.

The following figure shows the Access design view window;

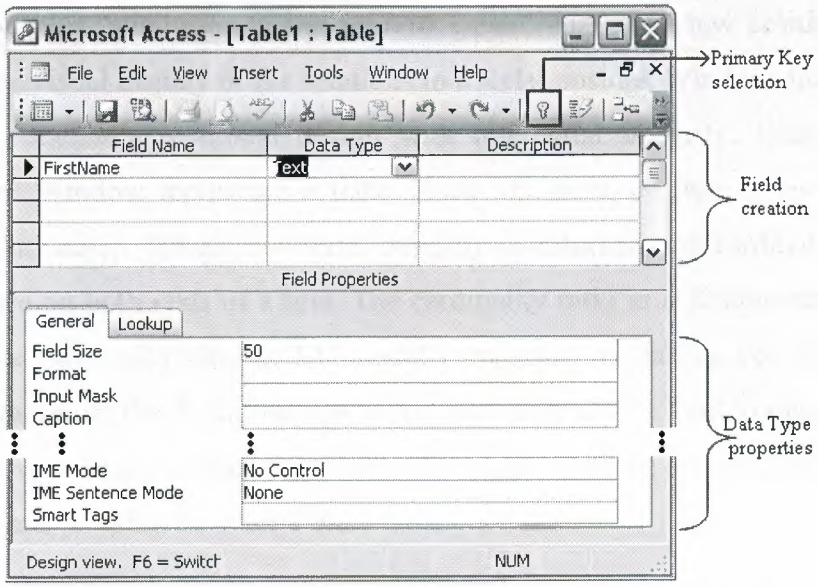


Figure 2.6 Microsoft Access 2003 create table design view window

ii. Creating table with Access wizard assistance

- Double-click **Create table by using wizard**
- Select the type of table (Business or Personal)
- Choose a table from the **Sample Tables** list
- Select a data field to include in the table
- Click the single arrow
- Repeat steps d and e for all other data fields you wish to include in the table
- Click **Finish**
- Enter data into table

2.12. Relationships between Tables

Access supports the definition of relationships between tables. A relationship specifies the fields that can be equijoined together to derive related data from the tables specified by the relationship. Access uses relationships for various purposes. The most important use of relationships is to enforce referential integrity.

- Select Relationships from the Tools menu to open the Relationships Window.

The Relationships Window lets the user to create, edit and view relationships in a database. The visual display of the relations in a Relationships Window also lets user to visualize a database's relational design with referential integrity. Each box in the Relationships Window represents a table. Each line between two tables represents a reference relationship. When referential integrity is enforced, the cardinality ratio will also be shown on both ends of a line. The cardinality ratio in a Relationships Window represents the cardinality after an ER model is mapped into tables. For this reason the cardinality ratios in the Relationships Windows can only be one-to-one and one-to-many, as many-to-many relationships in the ER model must be mapped into two one-to-many relationships using an intermediate table.

- Double click the line between two tables to view the relationship in a Relationships Dialog box.

In the Relationships Dialog box, users can edit and view the related fields and also get information about whether referential integrity is enforced. If referential integrity is enforced, user can edit and view its cardinality ratio and its *cascade update* and *cascade Delete* options. Cascade update means that if the key of a primary table record is changed, the corresponding foreign key of the related records in the related table will be updated as well. Cascade delete means that if a primary table record is deleted, all related records in the related table will also be deleted. From the Relationships Dialog box, users can open a Join Properties Dialog box to edit and view the join type (equijoin or outer join) of the relationship.

- Click the Join Type button in the Relationships Dialog box to view the join type in a Join Properties Dialog box.
- Click Cancel to close the Join Properties Dialog.
- Click Cancel to close the Relationships Dialog.
- Close the Relationships Window.

2.13. Structured Query Language (SQL)

SQL, Structured Query Language, is the driving force behind nearly every database product available to developers and end users today. Depending on the audience for which the product is designed, the SQL may be hidden behind the scenes, doing its work through a point and click interface rather than the typing of a long SQL statement. One of the major benefits of SQL is its portability between vendors. Though there are numerous, vendor-specific extensions and dialects that color the language, the core concepts and principles of SQL remain constant.

SQL is an evolving standard language that closely parallels the evolution of the relational database model. Early implementations were developed in the 1970s and an ANSI committee developed a first standard around 1986 (also accepted by the ISO).

The structure of the language is based on the relational concept of all data being in table form rather than a flat file. Files have a specific structure and order while tables are unordered sets of items. Languages that address files must explicitly rely on the structure and the sequence of the file itself and thus become inextricably linked to the data structure. One of the most important precepts of the relational model is the separation of the logical and the physical implementation of the database. The SQL language is not directly connected to the underlying storage mechanism and is therefore not affected by the data structure or the storage mechanisms.

2.14. SQL Subsets

The SQL language is subdivided into four subsets that represent separate, functionally related tasks:

- **Data retrieval:** retrieves data from the database (e.g., SELECT).
- **Data Manipulation Language (DML):** inserts, removes and changes the rows (e.g., INSERT, UPDATE, and DELETE).
- **Data Definition Language (DDL):** creates, changes, and removes a table's structure (e.g., CREATE, ALTER, DROP, RENAME, and TRUNCATE).

- **Data Control Language (DCL):** gives and removes the rights to DBMS objects (e.g., GRANT, and REVOKE).

2.15. Data Manipulation Language Statements

The DML subset is the core that drives the SQL-based components consist of the following statements:

- SELECT - Retrieves data from the database
- DELETE - Deletes records from a database
- INSERT - Inserts new records into the database
- UPDATE - Modifies the existing records in the database

The operations that are used in SQL commands and aggregate functions are listed in the Table 2.1 and Table 2.2correspondingly;

Operators used in SQL	
Arithmetic operations	
*	Multiplication
/	Division
+	Addition
-	Subtraction
Logical operators	
AND	Returns true if only if both conditions are true
OR	Returns true f one or both conditions are true
NOT	Returns true if the condition is false
Predicate operations	
BETWEEN	Compares a value to a range of values
EXISTS	Compares a value to a lookup list
IN	Determines if a value exists in values
LIKE	Compares one value with another
IS NULL	Compares a value with Null
SOME/ANY	Performs a quantified comparison
Relational operations	
<	Less than
<=	Less than or equal to
=	Equal to
>=	Grater than or equal to
>	Grater than

Table2.1 Operators used in SQL statements

Aggregate Functions	
AVG	Averages all non-null numeric values in a column
COUNT	Counts the number of rows in a result set
MAX	Determines the maximum value in a column
MIN	Determines the minimum value in a column
SUM	Totals all numeric values in a column
COUNT	Averages all non-null numeric values in a column

Table2.2 Aggregate functions.

Note1: Variables will be appearing in <angle brackets>.

Note2: The commands between [square brackets] are optional conditions.

Note3: The | bar symbol is used to separate the items of the list.

2.15.1. The SELECT Statement

The general syntax for SELECT statement is;

*SELECT [DISTINCT] * | columns*
FROM table name
[WHERE predicates]
[ORDER BY order sequence]
[GROUP BY group list]
[HAVING having condition]

- **The WHERE Clause:** The WHERE clause uses a defined set of predicates, or logical expressions, to define the filtering conditions.
- **The ORDER BY Clause:** The ORDER BY clause is used in a SELECT statement to order the retrieved rows in the result set. Order is determined by the values in a comma separated list of one or more columns. Listing direction may be controlled by adding either the ASC or DESC modifier to the ORDER BY column list.
- **The GROUP BY Clause:** The GROUP BY clause is used in conjunction with the aggregate functions to combine rows with the same column value into a single row. The criteria for combining rows are based on the column list specified in the GROUP BY clause.

- **The HAVING Clause:** The HAVING clause is used in a SQL statement to limit the rows retrieved to those in which the aggregated columns meet the HAVING clause criteria. To utilize the HAVING clause, both a GROUP BY clause and one or more aggregated columns must be present in the statement

2.15.2. The DELETE Statement

The general syntax for DELETE statement is;

DELETE FROM table name
[WHERE predicates]

DELETE statement can be used to delete one or more rows from a relation. When used without the WHERE clause, all rows in a table will be deleted. The WHERE clause is used in conjunction with selection predicates to limit the rows that are deleted.

2.15.3. The INSERT Statement

The general syntax for INSERT statement is;

INSERT INTO table name
[(Column list)]
VALUES (update values)

INSERT statement can be used to add new rows of data to a relation. The syntax for this DML statement changes a bit. The keyword FROM is replaced by INTO before declaring the target relation name. A comma-separated should be included between the list of column names that is surrounded by parentheses to the statement to explicitly define the data targets. If column list not included the statement the data contained in the VALUES clause is inserted into the columns of the relation on a positional basis, as the columns appear in the relation definition.

2.15.4. The UPDATE Statement

The general syntax for UPDATE statement is;

UPDATE table name
SET column name = update value
[, column name = update value]
[WHERE predicates]

The UPDATE statement modifies column values in one or more rows of a relation. By default, all rows in the relation will be modified according to the column/value list that follows the SET clause. Each expression in the SET clause is composed of a column name, the equal sign, and an appropriate data value.

CHAPTER THREE

DATABASE CONCEPT OF BORLAND DELPHI 7

3.1. Overview of Delphi's Database features

Delphi has powerful and reliable data-management capabilities. Delphi enables users to create robust database applications quickly and easily. Delphi database applications can work directly with desktop databases like Paradox, dBase, the Local InterBase Server, and ODBC data sources like Microsoft Access. Database applications let users interact with information that is stored in databases.

Delphi provides support for relational database applications. Relational databases organize information into tables, which contain rows (records) and columns (fields). These tables can be manipulated by simple operations known as the relational calculus.

When designing a database application, the user must understand how the data is structured. Based on that structure, user can then design a user interface to display data to the users and allow the users to enter new information or modify existing data.

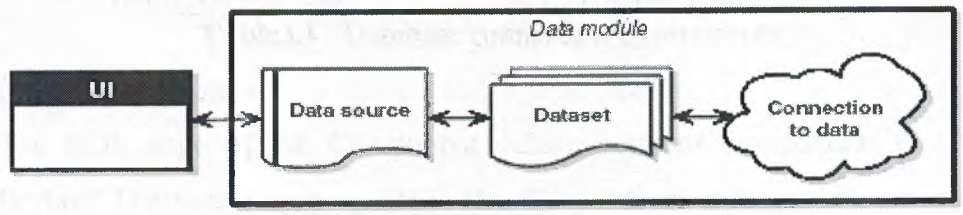
Building a database application is similar to building any other Delphi application.

- Creating and managing *projects*
- Creating *forms* and managing *units*
- Working with *components*, *properties,* and *events*
- Writing simple Object Pascal source code

3.2. General Database structure of Delphi

Database applications are built from user interface elements, components that represent database information (datasets), and components that connect these to each other and to the source of the database information. How to organize these pieces is the architecture

of developer's database application. The figure 3.1 illustrates the basic database structure of Delphi;



USER INTERFACE	DATA MODULE		
Isolating the user (UI) interface on a form has several advantages as; 1. Flexibility 2. Changing database info don't require to rewrite UI 3. Changing the UI don't require change the portion of the database 4. Lets user to Develop common forms	By isolating user interface into its own form, data module can be used to house the components that represent database information (datasets), and the components that connect these datasets.		
	data source The data source acts as a conduit between the user interface and a dataset hat represents information from a database.	**Dataset** Heart of database application is the dataset. This component represents a set of records from the underlying database.	**data connection** There are 4 basic mechanisms for connecting to the data: 1. Connecting directly to a database server. 2. Using a dedicated file on disk. 3. Connecting to another dataset. 4. Obtaining data from an RDS, or ADO datasets.

Figure3.1 Generic Database Architecture.

3.3. Connecting to a Database

Most dataset components can connect directly to a database server. Once connected, the dataset communicates with the server automatically. When the dataset is opened, it populates itself with data from the server, and when records are posted, they are sent back the server and applied. A single connection component can be shared by multiple datasets, or each dataset can use its own connection.

Each type of dataset connects to the database server using its own type of connection component, which is designed to work with a single data access mechanism. The following table (Table 3.1) lists these data access mechanisms and the associated connection components:

Data access mechanism	Connection component
Borland Database Engine (BDE)	TDatabase
ActiveX Data Objects (ADO)	TADOConnection
dbExpress	TSQLConnection
InterBase Express	TIBDatabase

Table3.1 Database connection Components.

I. The BDE page of the Component palette contains components that use the Borland Database Engine (BDE). The BDE defines a large API for interacting with databases. Of all the data access mechanisms, the BDE supports the broadest range of functions and comes with the most supporting utilities. It is the best way to work with data in Paradox or dBase tables. However, it is also the most complicated mechanism to deploy.

II. The ADO page of the Component palette contains components that use ActiveX Data Objects (ADO) to access database information through OLEDB. ADO is a Microsoft Standard. There is a broad range of ADO drivers available for connecting to different database servers. Using ADO-based components lets the user to integrate the application into an ADO-based environment.

III. The dbExpress page of the Component palette contains components that use dbExpress to access database information. dbExpress is a lightweight set of drivers that provide the fastest access to database information. In addition, dbExpress components support cross-platform development because they are also available on Linux. However, dbExpress database components also support the narrowest range of data manipulation functions.

IV. The InterBase page of the Component palette contains components that Access InterBase databases directly, without going through a separate engine layer.

V. The Data Access page of the Component palette contains components that can be used with any data access mechanism. This page includes *TClientDataset*, which can work with data stored on disk or, using the *TDataSetProvider* component also on this page, with components from one of the other groups.

Although each type of dataset uses a different connection component, they are all descendants of *TCustomConnection*. They all perform many of the same tasks and

surface many of the same properties, methods, and events. This chapter discusses many of these common tasks.

3.4. Borland Database Engine (BDE)

When we write a database application in Delphi, we need to use some *database engine* to access a data in a database. The database engine permits the user to concentrate on what data is wanted to be accessed, instead of how to access it. From the first version, Delphi provides database developers with the BDE (Borland Database Engine). Beside the BDE, Delphi from the fifth version supports Microsoft ADO database interface.

The BDE is a common data access layer for all of Borland's products, including Delphi and C++ Builder. The BDE consists of a collection of DLLs and utilities. The beauty of the BDE is the fact that all of the data manipulation is considered "transparent" to the developer. BDE comes with a set of drivers that enables user's application to talk to several different types of databases. These drivers translate high-level database commands (such as open or post) and tasks (record locking or SQL construction) into commands specific to a particular database type: Paradox, dBase, MS Access or any ODBC data source. The BDE API (Application Programming Interface) consists of more than 200 procedures and functions, which are available through the BDE unit.

Fortunately, user almost never needs to call any of these routines directly. Instead, user uses the BDE through the VCL's data access components, which are found on the Data Access page of Component Palette. To access the particular database the application only needs to know the Alias for the database and it will have access to all data in that database. The alias is set up in the BDE Administrator and specifies driver parameters and database locations. The BDE ships with a collection of database drivers, allowing access to a wide variety of data sources. The Standard (native) BDE drivers include Paradox, dBase, MS Access, ASCII text. Of course, any ODBC driver can also be used by the BDE through the ODBC Administrator.

The following figure illustrates the Borland Database Engine (BDE)'s database connection access and the components;

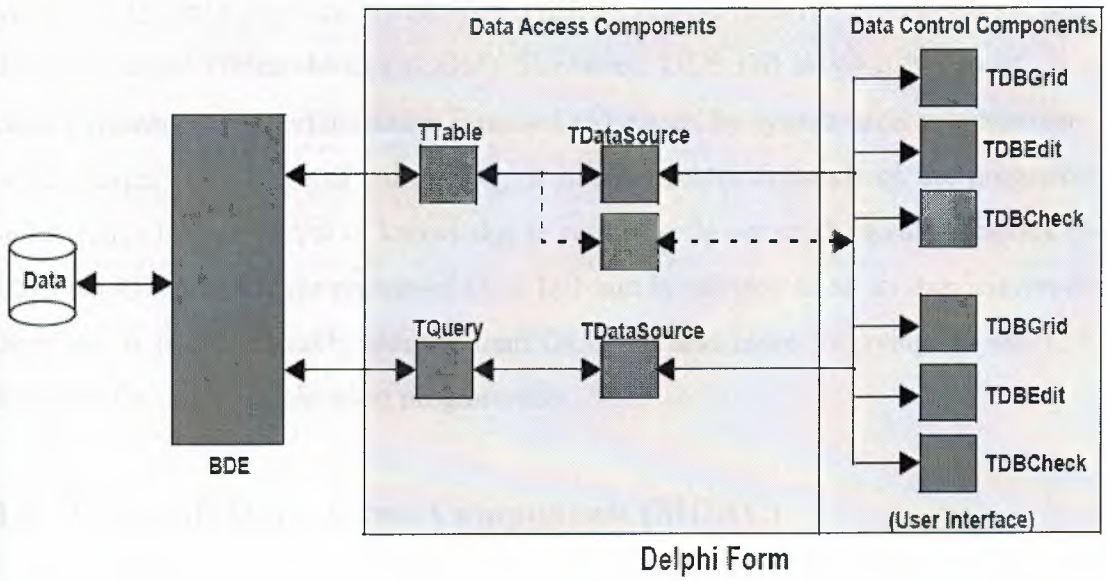


Figure3.2 BDE Database Component Architecture.

Delphi applications that use the BDE to access databases require that users to distribute the BDE with the application. When deploying the BDE with an application, user must use InstallShield Express or another Borland certified installation program.

The BDE has several advantages as well as disadvantages as a database engine. It's not my intention to discuss about why and when you should (or not) use the BDE approach over some non-BDE technique.

3.5. ActiveX Data Object

Since the mid-1980s, database programmers have been on a quest for the "holy grail" of *database independence*. The idea is to use a single API that applications can use to interact with many different sources of data. The use of such an API would release developers from dependence on a single database engine and allow them to adapt to the world's changing demands. Vendors have produced many solutions to this goal, the two most notable early solutions being Microsoft's Open Database Connectivity (ODBC) and Borland's Integrated Database Application Programming Interface (IDAPI), more commonly known as the Borland Database Engine (BDE).

Microsoft started to replace ODBC with OLE DB in the mid-1990s with the success of the Component Object Model (COM). However, OLE DB is what Microsoft would class a *system-level* interface and is intended to be used by system-level programmers. It is very large, complex, and unforgiving. It makes greater demands on the programmer and requires a higher level of knowledge in return for lower productivity. ActiveX Data Objects (ADO) is a layer on top of OLE DB and is referred to as an *application-level* interface. It is considerably simpler than OLE DB and more forgiving. In short, it is designed for use by application programmers.

3.6. Microsoft Data Access Components (MDAC)

ADO is part of a bigger picture called Microsoft Data Access Components (MDAC). MDAC is an umbrella for Microsoft's database technologies and includes ADO, OLE DB, ODBC, and RDS (Remote Data Services). MDAC is also distributed with most Microsoft products that have database content. Borland Delphi 7 ships with MDAC 2.6.

3.6.1. OLE DB Providers

OLE DB providers enable access to a source of data. They are ADO's equivalent to the dbExpress drivers and the BDE SQL Links. When you install MDAC, you automatically install the OLE DB providers shown in Table 3.2:

Driver	Provider	Description
MSDASQL	ODBC Drivers	ODBC drivers (default)
Microsoft.Jet.OLEDB.3.5	Jet 3.5	MS Access 97 databases only
Microsoft.Jet.OLEDB.4.0	Jet 4.0	MS Access and other databases
SQLOLEDB	SQL Server	MS SQL Server databases
MSDAORA	Oracle	Oracle databases
MSOLAP	OLAP Services	Online Analytical Processing
SampProv	Sample provider	OLE DB provider for CSV files
MSDAOSP	Simple provider	providers for simple text Data

Table3.2 OLE DB Providers Included With MDAC.

- The ODBC OLE DB provider is used for backward compatibility with ODBC.
- The Jet OLE DB providers support MS Access and other desktop databases.

- The SQL Server provider supports SQL Server 7, SQL Server 2000, and Microsoft Database Engine (MSDE).
- The OLE DB provider for OLAP can be used directly but is more often used by ADO Multi-Dimensional (ADOMD).

3.6.2. Using dbGo Components

Programmers familiar with the BDE, dbExpress, or IBExpress should recognize the set of components that make up dbGo. The Table 3.3 shows the dbGo Components;

dbGo Component	Description	BDE Equivalent
ADOConnection	Connection to a database	Database
ADOCommand	Executes an action SQL command	No equivalent
ADODataSet	All-purpose descendant of TDataSet	No equivalent
ADOTable	Encapsulation of a table	Table
ADOQuery	Encapsulation of SQLSELECT	Query
ADOStoredProc	Encapsulation of a stored procedure	StoredProc
RDSConnection	Remote Data Services connection	No equivalent

Table3.3 dbGo Components

The dataset components (ADODataSet, ADOTable, ADOQuery, and ADOStoredProc) are implemented almost entirely by their ancestor class, TCustomADODataSet. This component provides the majority of dataset functionality, and its descendants are mostly thin wrappers that expose different features of the same component.

As such, the components have a lot in common. In general, however, ADOTable, ADOQuery, and ADOStoredProc are viewed as "compatibility" components and are used to aid the transition of knowledge and code from their BDE counterparts; these compatibility components are similar to their counterparts but not identical.

3.7. Connection to a Database from ADO

- Drop an ADOTable onto a form. To indicate the database to connect to, ADO uses *connection strings*. Application developer can type in a connection string

by hand if user like. In general, developer will use the connection string editor, shown in Figure 3.3

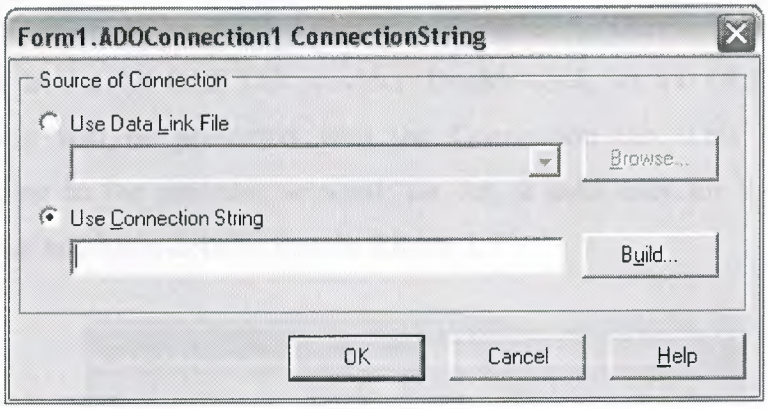


Figure3.3 Connection String Window of ADO connection.

- The user can click Build to go straight to Microsoft's connection string editor, shown in Figure 3.4

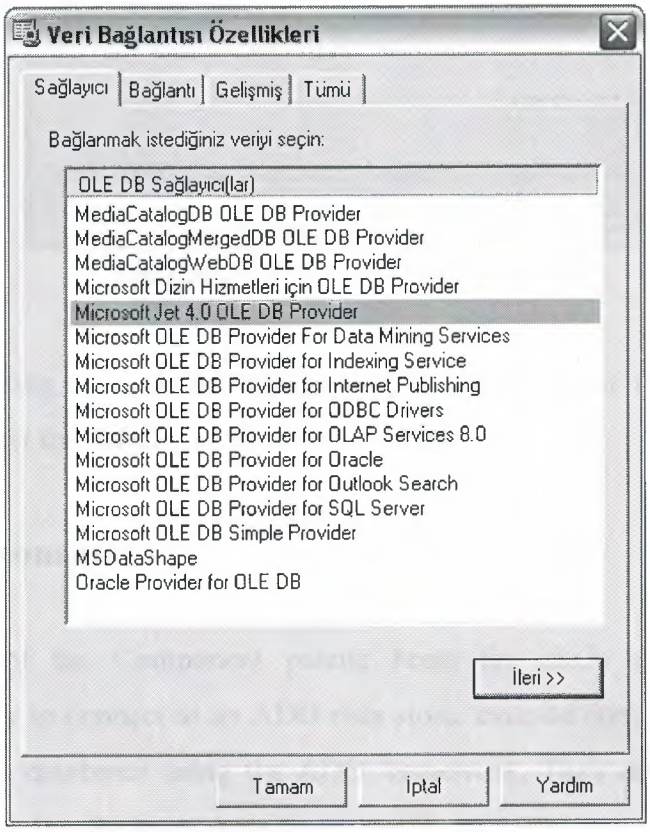


Figure3.4 ADO Connection String Editor.

- The first tab shows the OLE DB providers and service providers installed on applied computer. The list will vary according to the version of MDAC and other software installed.

- Select the Jet 4.0 OLE DB provider. Double-click Jet 4.0 OLE DB Provider, and you will be presented with the Connection tab. This page is varies according to the provider selected; for Jet, it asks user for the name of the database and login details. See the Figure 3.5

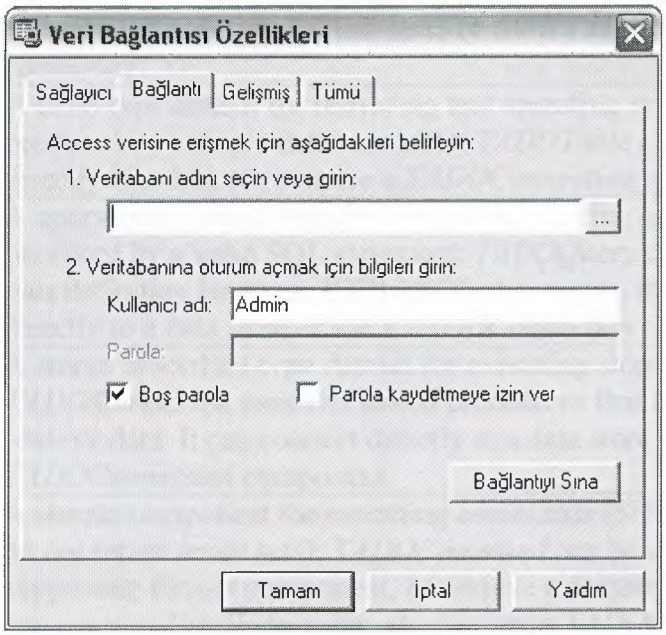


Figure3.5 Connection tab of Editor.

- After selecting the desired Database and accepting it click the Test Connection button to test the validity of the selection.

3.8. ADO Components

The ADO page of the Component palette hosts the *dbGo* components. These components let user to connect to an ADO data store, execute commands, and retrieve data from tables in databases using the ADO framework. They require ADO 2.1 (or higher) to be installed on the host computer.

In addition, *dbGo* includes *TADOCommand*, a simple component that is not a dataset but which represents an SQL command to be executed on the ADO data store.

The following table lists the ADO components.

Component	Usage
TADOConnection	A database connection component that establishes a connection with an ADO data store; multiple ADO dataset and command components can share this connection to execute commands, retrieve data, and operate on metadata.
TADODataSet	The primary dataset for retrieving and operating on data; *TADODataSet* can retrieve data from a single or multiple tables; can connect directly to a data store or use a *TADOConnection* component.
TADOTable	A table-type dataset for retrieving and operating on a recordset produced by a single database table; *TADOTable* can connect directly to a data store or use a *TADOConnection* component.
TADOQuery	A query-type dataset for retrieving and operating on a recordset produced by a valid SQL statement; *TADOQuery* can also execute data definition language (DDL) SQL statements. It can connect directly to a data store or use a *TADOConnection* component.
TADOStoredProc	A stored procedure-type dataset for executing stored procedures; *TADOStoredProc* executes stored procedures that may or may not retrieve data. It can connect directly to a data store or use a *TADOConnection* component.
TADOCommand	A simple component for executing commands (SQL statements that do not return result sets); *TADOCommand* can be used with a supporting dataset component, or retrieve a dataset from a table; It can connect directly to a data store or use a *TADOConnection* component.

Table 3.4 ADO components.

CHAPTER FOUR

TIMETABLE SCHEDULING PROGRAM WITH DELPHI

4.1. Introduction

The computer technology is improving in every single consecutive day. Even imagining the near future of the Improvement of the computer technology becomes such a condition that can not seem to be possible. People, who are thinking improvement is reached its highest level, will seen that they were not right. The software programs are not standing on the same level, they are also improving depending on the computer technology, and automation programs become more demanded in every aspect of life. The automation programs also makes works quite desirable, rapid, and adjustable besides making humans life easier, and they lets people do some other necessary things instead of spending hours on the occupied job.

Educational associations such as high schools or universities actually all of them from primary school to universities are offering courses, lessons, or lectures. Offering courses needs an arrangement to put the courses to right period of time or day of week. Process of this duty is varying between people to person or it is vary between associations, all do this job in some way. Most of them spending lots of times, hours may be days on this little application if a computerized program is not used. The a good solution for replacing all offered courses for the upcoming term is using a software program instead of spending lots of time on forming the new timetable. This process takes just a few minutes by pre-designed software program instead hours or days.

4.2. Timetable program

The timetable scheduling program is actually consisting of three main parts on the developing point of view. First of hose parties the user interfaces of the program, second one is the relational database which as in most useful automation programs, and

the last is the reporting part of it. If it is dialed with the user's point of view the timetable scheduling program has two part one for the setting and keeping the data's such as existing departments, the courses or lecturers in the database, and the second part for forming the most optimal timetable for the offered courses without any clash of courses or lecturers.

On the other hand, at the design stage such as while thinking on the program also the replacement algorithm was the part which, almost half of the developing time spent on it. There are a lot of limitations such as courses cannot be at the same period which are belong to same term and year, as well as if they are belong to same department. And also there are many pieces of things to consider for example the courses of curse cannot be at the same period which is offered by the same lecturer.

As it is mentioned before the program has two parts on the user's point, the rest of this chapter will be discussed the program on this condition. But it is not meant that the programming stages will not be discussed; only explanation process will follow this rule. Now let us take a tour of the program to have more enough information.

4.2.1. The main appearance of the program

The main window of the program is the base of all the other windows except the report window of the program, it means that the main window is in the fsMDIform form style format and the parent form of all fsMDIChild forms at all. The appearance of the main window called ***'timetable scheduling program'*** is shown in figure 4.1;

It has plain visual concept that includes Menu at the top and control tab with link buttons to Quick access. User can reach the options of the program using either from the menu bar or from control tab.

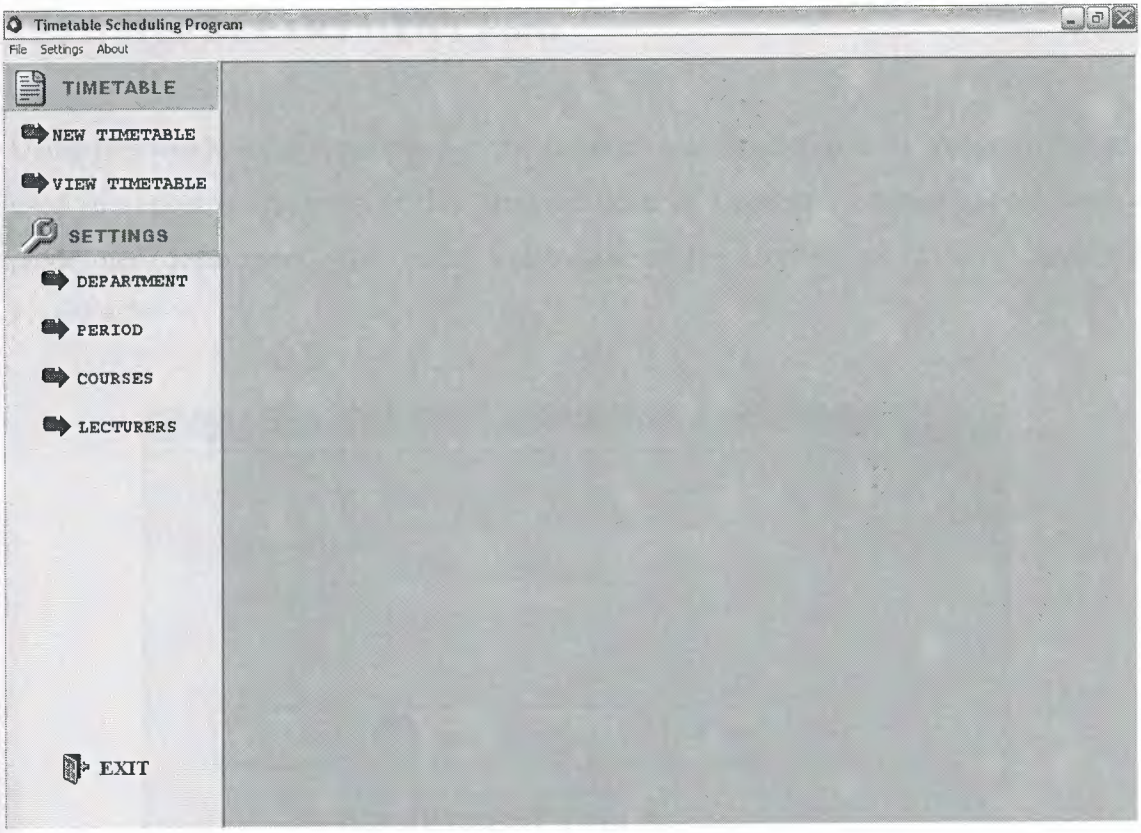


Figure 4.1 Main Windows.

This form is the self creator of other forms. This appearance is inherited to every window of the program, only the preview of the report acts free from this window according to its property.

4.2.2. The Settings portion of the Program

The SETTINGS portion of the program is designed to set all necessary operations to form a time table; they should be certainly defined to have a good timetable without wrong replacements of the courses. There are four categories shown in figure 4.1 that the user have to deal with them are as follows;

- Department: Existing departments of the program owner's association
- Period: Beginning and ending time of period for days
- Courses: All courses that are offered in time.
- Lecturers: Academic personnel

4.2.2.1. The DEPARTMENT Form

Using this window the department of the faculties are introduced to the program that are used in almost every steps of this program such as Lecturers, courses should have a predefined department. The visual illustration of the Department window is in the Figure 4.2;

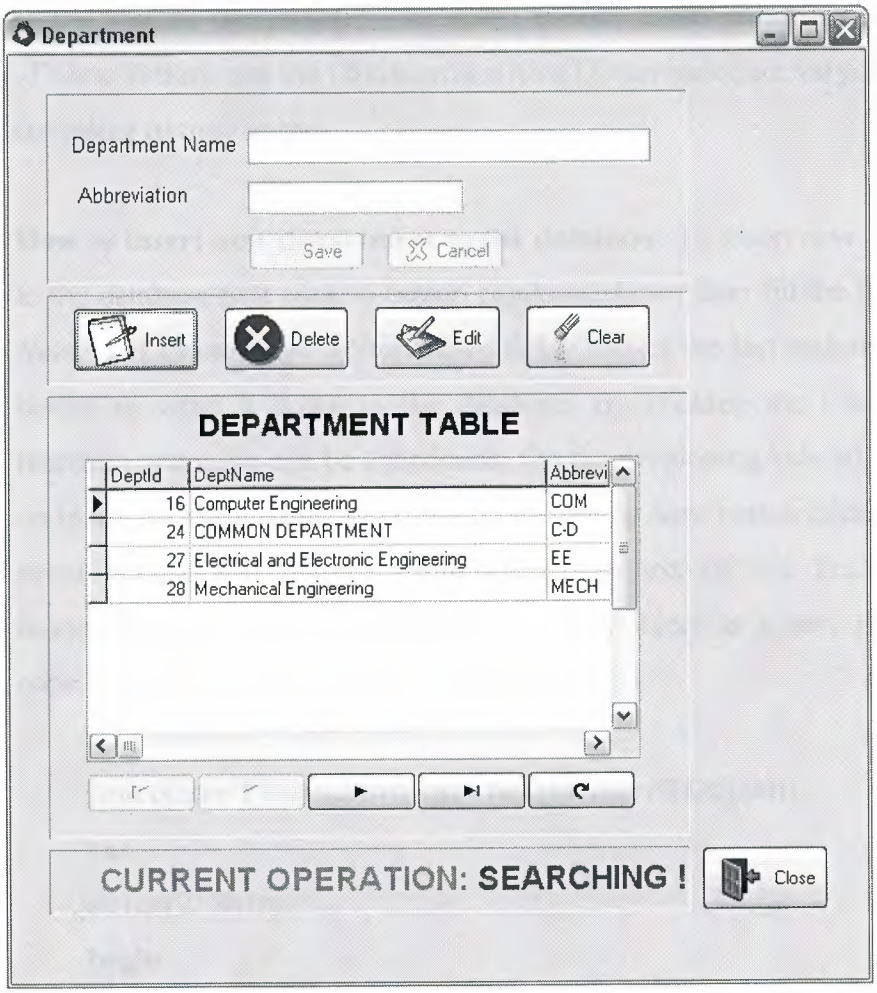


Figure 4.2 Department Windows.

First of all as it is seen in the Figure 4.2 there is predefined department called 'COMMON DEPARTMENT' in Upper Case letter can not be deleted or modified, since it is protected. This department is defined by the developer and also protected because it will be used later on while defining the courses which are common for all departments or more than one department. It will be used in such a case, and will cause those courses not to be clash. For example imagining that there are two

students A and B of departments COM and EE respectively, they are taking a course MATH which is belongs to department X, in this case this Course must have no clash with the timetable of the student A as well as student B. To overcome of this problem the MATH should be assigned to COMMON DEPARTMENT.

Supposing that, the main objective of the Department form can be dealt after explaining the predefined department. The window's some standard tools which are same as the rest of the program, are save, cancel, clear, close buttons. Insert, update, Delete buttons and the DBdatagrid with a DBnavigator are varying between forms according to their usage.

- **How to insert new department to the database:** To insert new department to the database first click to button captioned *Insert* than fill the D*epartment Name* and *Department Abbreviation* fields and as the last action click *save* button to write it down to the database. By clicking the Cancel button insertion operation can be abandoned. On the developing side what is going on in the background of the program is; as far as *Save* button clicked its click event occurs. On buttons' **OnClick** event text of the TextFields are immediately written into *tbdepartment* table object as a new record. The code is simple pascal code as shown in Table 4.1;

```
procedure TForm2.BitBtn2Click(Sender: TObject);
var
str1,str2: String;
begin
        str1:=Edit1.Text;
        str2:=Edit2.Text;
        ADOTable1.Insert;
        ADOTable1.FieldValues['DeptName']:=str1;
        ADOTable1.FieldValues['Abbreviation']:=str2;
        ADOTable1.Post;
        DBGrid1.Refresh;
        ADOTable1.Close;
```

```
        ADOTable1.Open;
        Edit1.Text:='';
        Edit2.Text:='';
        Edit1.Enabled:=False;
        Edit2.Enabled:=False;
        BitBtn2.Enabled:=False;
        BitBtn3.Enabled:=True;
        BitBtn4.Enabled:=True;
        BitBtn5.Enabled:=False;
        BitBtn1.Enabled:=True;
    end
```

Table4.1 Insertion code core.

- **How to Update an existing Department:** Updating has many similarities with the insertion operation while programming it instead of ADOTable.Insert command just changing it with ADOTable.Edit will solve the problem. Users also has similar operation to do just before editing an existing data user should chose the data from DBgrid than click the *Edit* buton. Then values of selected data will be replaced to EditBoxes and ready to be updated. Whitout selected any row from the DBGrid will cause the program to display a designed error message (see Figure 4.3) and will have no effect on any data. *Edit* buttons' **OnClick** event has the same Pascal codes only differing while writing to table object, ADOTable1.Edit is used instead ADOTable1.Insert;. Success of the operation will be stated to the user by displaying the message (see Figure 4.3).

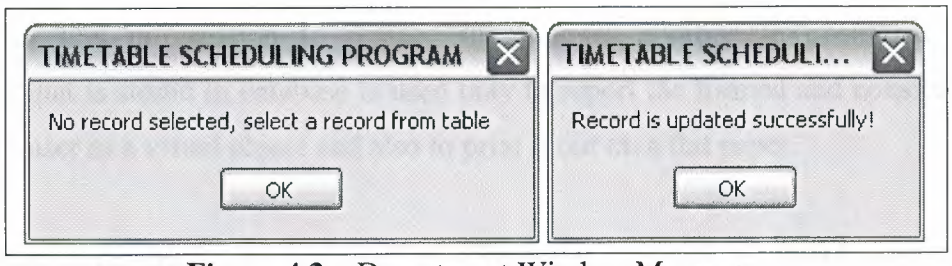


Figure 4.3 Department Window Messages.

- **Deleting the non-necessary data from the department list:** delete operation will delete the selected data from database evenly, and will no return chance. User can delete any row as he/she wish except the protected department mentioned before. Deleting is the easiest operation while working on database application. The pieces of the code in Table 4.2 will explain its simplicity. At the end of the operation the success message will be displayed to the user as well.

```
procedure TForm2.BitBtn3Click(Sender: TObject);
begin
        try
            ADOTable1.Delete;
        except
            ShowMessage('the record can not be deleted');
        end;

      Edit1.Text:='';
      Edit2.Text:='';

end;
```

Table4.2 Code core of Deletion operation with exception handling.

4.2.2.2. The PERIOD Window

Period window just for arranges that when courses begins and when they ends. The arranged times are not applicable while scheduling algorithm process, actually there is no need for this information to replace the courses appropriate positions. The information that is stored in database is used only to report the formed and constructed timetable to user as a visual object and also to print it out on a flat paper.

At the design time mask edit are used to take time intervals, to avoid user made mistakes. Because the time format is may not be acceptable by the program that's why the mask edits' mask properties are set to default **HH:MM** format.

Figure4.4 Period Window.

4.2.2.3. The COURSES Window

With Courses window the courses are described and saved for the program to form a time table. The course data in database has a relationship with the Lecturers; the courses will be load to lecturers' course list to determine which lecturer offer which courses. For that reason user can not allowed to delete an existing course if there is any lecturer who is offering this course.

The general view of the courses window illustrated on Figure 4.5;

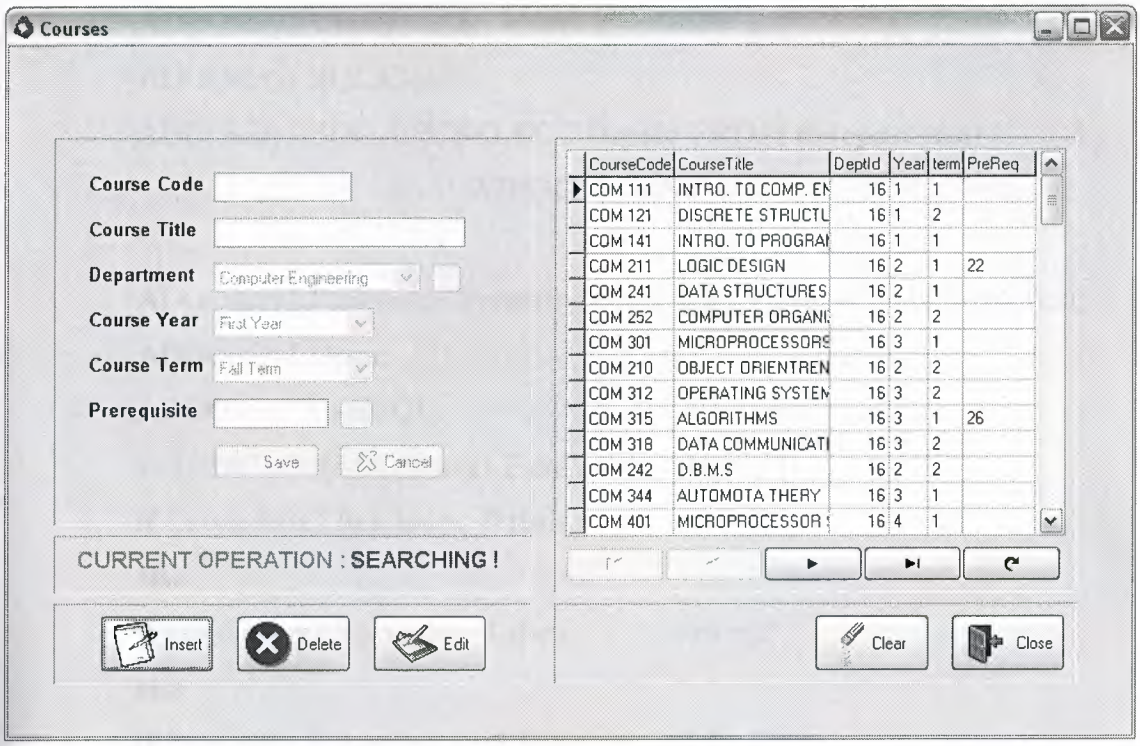


Figure 4.5 Courses Window.

Using this window as shown in Figure 4.5 user can insert new course. The fields describe the all necessary information about a course to create a timetable. The operations those are same as the department window on the users side, but there is some differences to the developer In this form also the department of the course is necessary to fulfill the operation, to do this ComboBox is used and its items are every time is refreshed on OnFormCreate, OnFormActivate, and ComboBox's OnDropDown event occurs, to display the saved departments on the items list of the ComboBox.

ComboBox are lists the Departments by their name, but to write down to the database their department id's are needed, to find and replace the id's of the departments SQL command is used to retrieve the data. The sql statement will be seen in the code piece of the Save button showed in Table 4.3;

```
procedure TForm5.BitBtn1Click(Sender: TObject);
begin
    str1:=Edit1.Text;
    str2:=Edit2.Text;
```

```
ADOQuery1.Close;
ADOQuery1.SQL.Clear;
ADOQuery1.SQL.Add('SELECT DeptId FROM tbdepartment '
                  ' WHERE DeptName=:dep'
                  );
ADOQuery1.Parameters.ParamByName('dep').Value:=ComboBox4.Text;
ADOQuery1.Open;
ADOQuery1.ExecSQL;
str3:=IntToStr(ADOQuery1.FieldValues['DeptId']);
if ComboBox2.ItemIndex=0 then          str4:='1'
else
if ComboBox2.ItemIndex=1 then          str4:='2'
else
if ComboBox2.ItemIndex=2 then          str4:='3'
else          str4:='4';

if ComboBox3.ItemIndex=0 then
 str5:='1'
else
 str5:='2';

str6:=Edit3.Text;
ADOTable1.Insert;
ADOTable1.FieldValues['CourseCode']:=str1;
ADOTable1.FieldValues['CourseTitle']:=str2;
ADOTable1.FieldValues['DeptId']:=str3;
ADOTable1.FieldValues['Year']:=str4;
ADOTable1.FieldValues['Term']:=str5;
ADOTable1.FieldValues['PreReq']:=str6;
ADOTable1.Post;
DBGrid1.Refresh;
End
```

Table4.3 insert operation code with sql statement.

Rest of the operations are same with the department window.

The new thing here is adding prerequisite to the courses. User can add prerequisite to any course either at the first insertion or while updating the data, depending on the situation. The speed button is used to create a prerequisite, when speed button is clicked

The prerequisite dialog will be appeared on the screen and has the available courses on its DBGrid those are the courses that are saved to the database before it is created. The illustration of the prerequisite dialog is as in Figure 4.6;

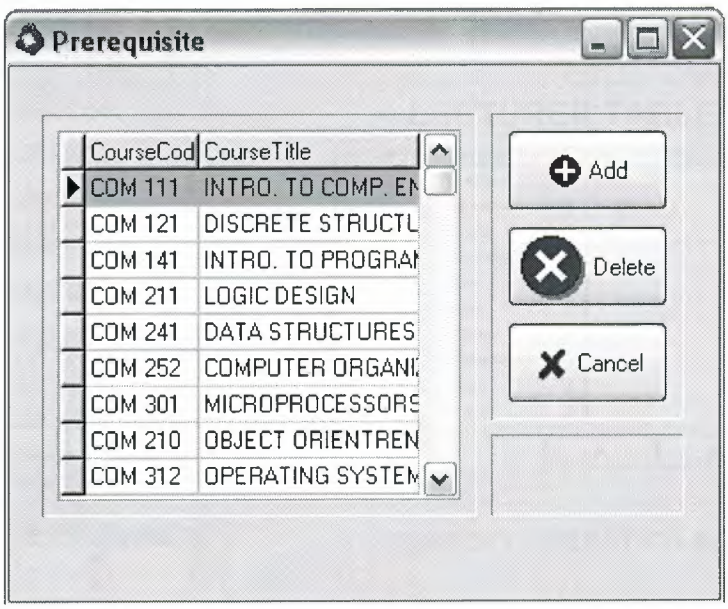


Figure4.6 Prerequisite Dialog Window.

User can select one of the listed courses as the prerequisite of the creator of this dialog. Deleting the prerequisite simple, user will not select the prerequisite to delete, the selected courses prerequisite will be deleted automatically as far as the delete Button clicked.

As a final word about this window when user tries to delete the courses that has a prerequisite the error message will be displayed to the user. Because of the relation it will not be deleted from the database.

4.2.2.4. The LECTURERS Window

The Lecturers window is made-up of two part first one is to insert, update and delete the lecturers with necessary information about their personal information and their departments in which they are dependent of.

The transaction operations insert, update, delete are exactly same as the other transaction operations explained just before this section. Only the format is adapted to the requirement of this subject. The lecturer Window is shown in Figure 4.7;

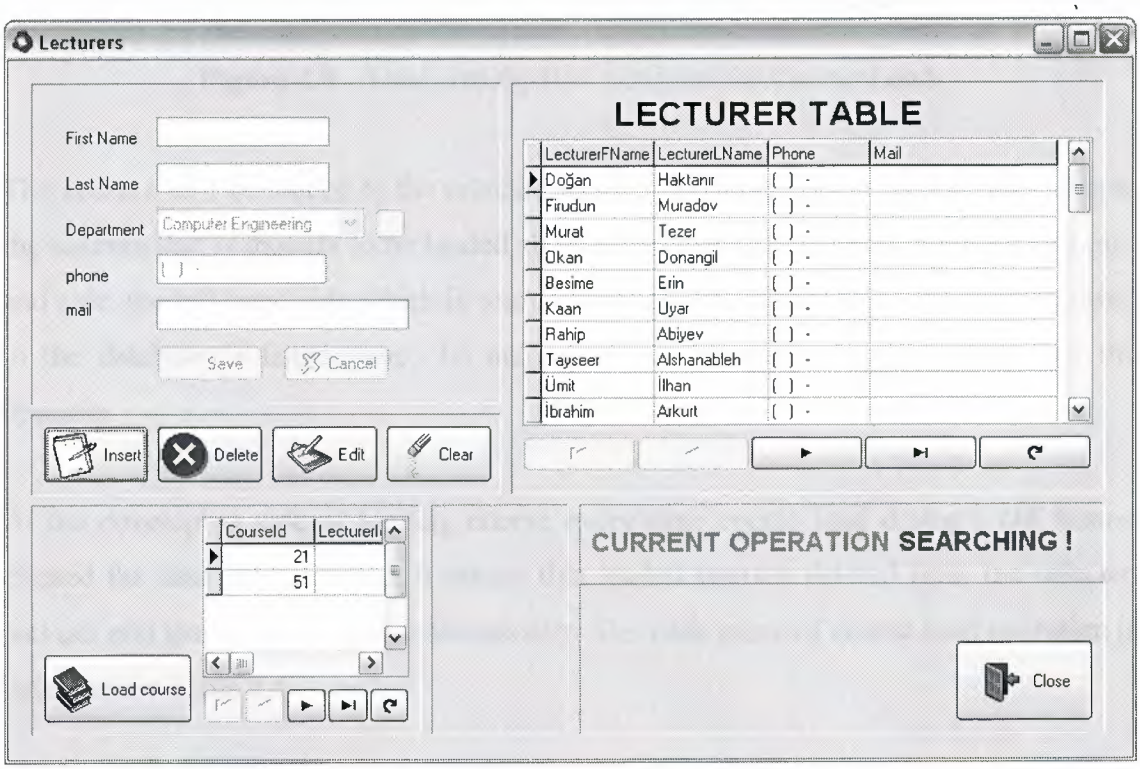


Figure4.7 Lecturer Window.

The second part of this window is loading course to the Lecturers. To load course or courses to any lecturer; select the lecturer that is going to have the course load operation fro DBGrid table than click he button which has a caption '*Load Course*'. Immediately the dual list dialog box will appear by clicking the button (see Figure 4.8).

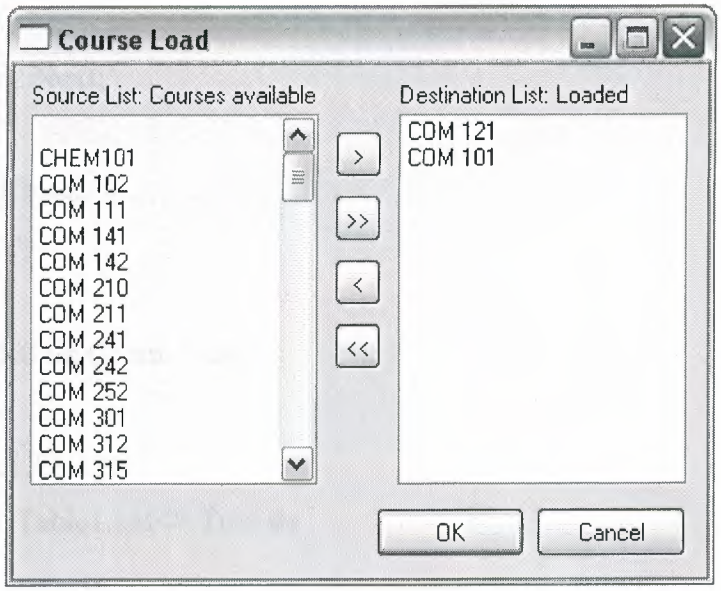


Figure 4.8 DualListDlg-Box windows for Course Load.

The courses will be loaded to the selected lecturer. Loading course now is quite simple the courses that is thought to be loaded should be passed to the Destination list at right-and side, the left-hand side which is source list is the pool for courses, all courses saved in the database is listed there. To unload course from selected lecturer is just the opposite.

At the developing side of loading course every time course load dialog's *OK* button clicked the data are refreshed. It means that loaded courses deleted from the selected lecturer and than loaded again automatically. The code piece of course load operation is listed below Table 4.4;

```
procedure TDualListDlg.OKBtnClick(Sender: TObject);
var
 i:Integer;
begin
 ADOTable2.First;
 while ADOTable2.Eof<>True do
  begin
   if StrToInt(ADOTable2LecturerId.Text)=Form1.glovar2 then
    ADOTable2.Delete
```

```
    else
    ADOTable2.Next;
  end;
 ADOTable2.Close;
 ADOTable2.Open;

 for  i:=0 to DstList.Count-1  do
  begin
   ADOTable1.First;
   while ADOTable1.eof<> True do
    begin
     if ADOTable1CourseCode.Text=DstList.Items.Strings[i] then
      begin
       ADOTable2.Insert;
       ADOTable2.FieldValues['LecturerId']:=Form1.glovar2;
       ADOTable2.FieldValues['CourseId']:=StrToInt(ADOTable1CourseId.Text);
       ADOTable2.Post;
       ADOTable2.Close;
       ADOTable2.Open;
      end;
     ADOTable1.Next;;
    end;
  end;
ADOTable1.Close;
ADOTable2.Close;
ADOConnection1.Close;
close;
end;
```

Table4.4 Code for loadinng course to lecturers.

4.2.3. Timetable Scheduling Wizard

Now it is time to schedule a new timetable for the coming term, after all necessary settings applied to the program. This portion of the program is designed as a wizard, because most computer users are very familiar to do things using wizards. Wizards are make operations easier since they force users to the necessary jobs by specified steps.

Although this wizard make things easy for the user, developing this wizard is not as easy as its usage. There are six steps are left to have a printable timetable on the users perspective. These steps are listed below in Table 4.5;

STEPS	DESCRIPTION
Step1	Database Check
Step2	Offering courses
Step3	Specifying the student population
Step4	Specify the lecturer offering related courses
Step5	A few minute waiting for replacement
Step6	Printing table to the report page

Table4.5 Steps of the Wizard.

The explanation of the wizard will be based on these steps, again all related usage notes and also the developing informations will be in same manner as it is like before in the explanation of Settings part of the program.

As far as user chose new timetable tab from program, it immediately cleans all database table objects which are used while creating a new timetable schedule, for example elderly replaced courses are deleted from the replacement table which is called *'tbreplacement'*, the pointer tables, few dummy tables (there are few dummy tables for make a comparison between courses, and), and also printable course replacement tables are deleted as well. And it passes to the step1; now let's begin the steps with Database Check.

4.2.3.1. Step1, Database Check

Wizard shows the user amount of records of departments, courses, and lecturers. It does not allow user to go next step if those checked tables are not having any record. In this step wizard lets user to go and append these tables, it allows user either insert or delete records into or from tables. To supply this operation there are little speed buttons are replaced to the end of each, these buttons take user to Department, Courses or lecturers windows. When the user comes back it refreshes the database check and displays the appended record counts. And than if there is no check error allow user to go next step. The appearance of the wizard's first step can be seen in the figure 4.9;

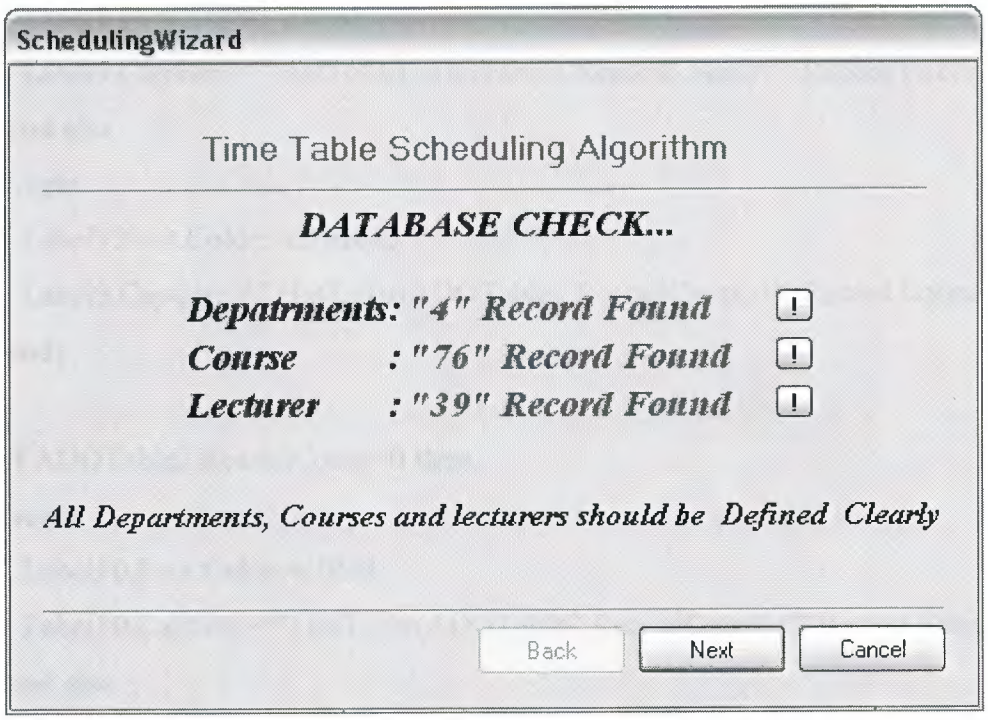


Figure 4.9 The Scheduling Wizard step1.

On the other hand what is happening on the back of the program is by calling this wizard form by the forms Oncreate all record counts of the *'tbdepartment'*, *'tbcourse'*, *'tblecturer'* are calculated and according to the results record counts are displayed on the screen, yellow font color for valid calculation and red font color for invalid calculation is used to display. After checking the record counts if any of the checked property is invalid the button will not be enabled go for next step, vice versa force the button to be enabled.

If user go to the other forms by using either speed buttons or menu bar or quick access tab the wizard form will be OnDeactivate situation, when user come to the wizard back forms' OnActivate method occurs and it does the check operation once more to refres the check results displayed on the screen previously, and do the buttons' possibility of enable (See the Table 4.6 same for both forms OnCreate and OnActivate Methods).

```
procedure TDualListDlg1.FormCreate(Sender: TObject);
begin
  if ADOTable1.RecordCount=0 then
  begin
   Label9.Font.Color:=clRed;
   Label9.Caption:=""+IntToStr(ADOTable1.RecordCount)+" Record Found ';
  end else
  begin
   Label9.Font.Color:=clGreen;
   Label9.Caption:=""+IntToStr(ADOTable1.RecordCount)+" Record Found ';
  end;

  if ADOTable2.RecordCount=0 then
  begin
   Label10.Font.Color:=clRed;
   Label10.Caption:=""+IntToStr(ADOTable2.RecordCount)+" Record Found ';
  end else
  begin
   Label10.Font.Color:=clGreen;
   Label10.Caption:=""+IntToStr(ADOTable2.RecordCount)+" Record Found ';
  end;

  if ADOTable3.RecordCount=0 then
  begin
   Label11.Font.Color:=clRed;
   Label11.Caption:=""+IntToStr(ADOTable3.RecordCount)+" Record Found ';
```

```
    end else
    begin
      Label11.Font.Color:=clGreen;
      Label11.Caption:=""+IntToStr(ADOTable3.RecordCount)+'" Record Found ';
    end;
  //check the next button. valid? - not valid?
    if (ADOTable1.RecordCount=0) or
       (ADOTable2.RecordCount=0) or
       (ADOTable3.RecordCount=0) then
        Button2.Enabled:=False;

  end;
```

Table4.6 Wizard form code core for OnCreate Method.

4.2.3.2. Step2, Offering Courses

Coming to the step two means that the first step is passed correctly. By the second step the two list boxes will be appear on the screen, the list settled on the left side displays all the courses those are exist in the database and right settled one is to see which courses are offered for coming term. User can select courses one by one or can select using multi select property. Selecting course can be done by double clicking the courses that are listed on the left which is actually source list or select one and click button that is captioned '>' or if user want to select all courses to offer for this term then clicks the '>>' button as it is like so in most windows applications. To go the next step click next is enough in this step. There is no check for buttons possibility of enable for this time. The appearance of the window is illustrated on Figure 4.10;

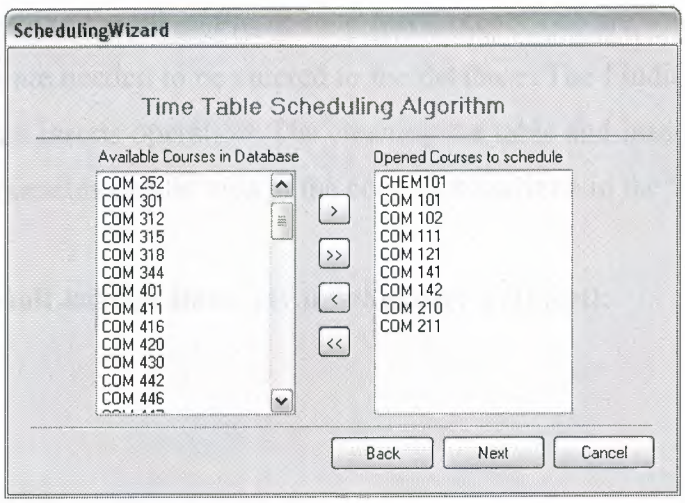


Figure 4.10 The Scheduling Wizard Step2.

When the wizard is reached the step2 the source list is filled up from the database's table object called *'tbcourse'*. A piece of code can be seen on below table (Table 4.7).

```
while ADOTable2.Eof<>true do
begin
  SrcList.Items.add(ADOTable2.FieldValues['CourseCode']);
  ADOTable2.Next;
end;
```

Table4.7 The code core of the filling source list to offer courses.

Then the relation between source lis and destination list begins. There are three procedures and a function is used to follow the action of the user. The function and three procedures are;

- function GetFirstSelection(List: TCustomListBox): Integer;
- procedure MoveSelected(List: TCustomListBox; Items: TStrings);
- procedure SetItem(List: TListBox; Index: Integer);
- procedure SetButtons;

After selecting all courses those are going to be offered, pressing the *'Next'* button will take the user to the next step. By the time that passing to next step the *'Next'* buttons' OnClick event occurs, with this event occurrence the selected courses are inserted into database's *'tbcourse_status'* table. This table is made of two fields at all one is CourseId, and the other is the Course Population. In this step the CourseId's are filled

the table after cleaning the table. Please note that CourseCode are selected from source list but CourseIds are needed to be entered to the database. The Finding the courseIds is also embedded into inserts operation. The cleaning the table and inserting into the table with searching CourseIds can be seen in the code piece written in the Table 4.8;

```
procedure TDualListDlg1.Button4Click(Sender: TObject);
var
 i:Integer;
begin
 ADOTable4.First;
 while ADOTable4.Eof <> True do
    ADOTable4.Delete;
 ADOTable4.Close;

 ADOTable4.Open;
 ADOTable4.First;
 for i:=0 to DstList.Count - 1 do
 begin
  ADOTable2.First;
  while ADOTable2.eof<> True do
   begin
    if ADOTable2CourseCode.Text=DstList.Items.Strings[i] then
     begin
      ADOTable4.Insert;
      ADOTable4.FieldValues['CourseId']:=StrToInt(ADOTable2CourseId.Text);
      ADOTable4.FieldValues['population']:=50;
      ADOTable4.FieldValues['CourseCode']:=ADOTable2CourseCode.Text;
      ADOTable4.Post;
      ADOTable4.Close;
      ADOTable4.Open;
     end;
    ADOTable2.Next;;
   end;
```

```
  end;
end;
```

Table4.8 The code of constructing SourseStatus.

4.2.3.3. Step3, Stating the Course Populations

This step is just for determining the population of courses which are offered in previous step by default all courses have population 50. In this case population actually indicates that how many group will there be for currently dealing course. For this project it is fixed that, a group is consists of fifteen students. If user do not want any course in current term, the should go back to the previous step and take that course out from the offered course list and come back again to this step. The user have to determine only courses that have more than one group, rest has already one group anyway. The user interface of this step is shown in the Figure 4.11;

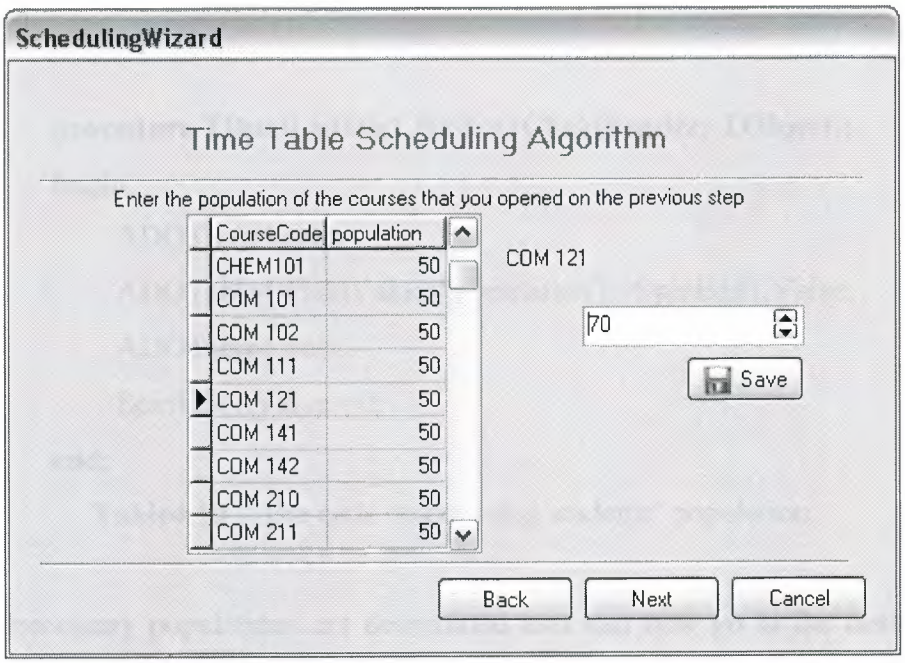


Figure 4.11 The Scheduling Wizard Step3.

As it is seen in the Figure 4.11 user select the course to increase or decrease the student population from DBGrid than either using the arrow key of the keyboard or just writing it as a plain text determine the population, and click the *Save* button. While doing this

operation firstly DBGrid's **OnCellClick** event occurs (see the DBGrid OnCellClick event's code on Table 4.9);

```
procedure TDualListDlg1.DBGrid1CellClick(Column: TColumn);
begin
Label15.Caption:=ADOTable4CourseCode.Text;
SpinEdit1.SetFocus;
SpinEdit1.Value:=ADOTable4.FieldValues['population'];
end;
```

Table4.9 Code of DBGrid's OnCellClick event.

When *Save* button clicked then the determined population actually the SpinEditBox's text is written to the database, as it is illustrated on the Table4.10, but this time data is not inserted as a new record because it already exists in the table remember that in previous step status of the course were inserted to the table now its population is determined.

```
procedure TDualListDlg1.BitBtn1Click(Sender: TObject);
begin
    ADOTable4.Edit;
    ADOTable4.FieldValues['population']:=SpinEdit1.Value;
    ADOTable4.Post;
    SpinEdit1.Value:=0;
end;
```

Table4.10 The code determining students' population.

After all necessary populations are determined user can now go to the next step just simply clicking the *Next* button. When user clicks that button, immediately OnClick event occurs and perform the actions designed by the developer. The action is determining the groups according to the course population. A piece code is shown to illustrate this action in Table 4.11;

```
procedure TDualListDlg1.Button7Click(Sender: TObject);
begin
 ADOTable4.First;
 while ADOTable4.Eof <> True do
  begin
    gr:=ceil(StrToInt(ADOTable4population.Text)/50);
    for  i:= 1 to gr do
     begin
       ADOTable5.Insert;
       ADOTable5.FieldValues['CourseId']:=ADOTable4CourseId.Text;
       ADOTable5.FieldValues['groups']:=IntToStr(i);
       ADOTable5.Post;
       ADOTable5.Close;
       ADOTable5.Open;
     end;
    ADOTable4.Next;
  end;
end;
```

Table4.11 Code for determining groups of courses.

4.2.3.4. Step4, Determine which Lecturer will deal with offered Courses

This is the last step for user to do some action. Although it is the last for user, it is the longest part for user, because user should determine the lecturers for all courses one by one and also for every course group to go to next step.

The appearance of the window is like in Figure 4.11;

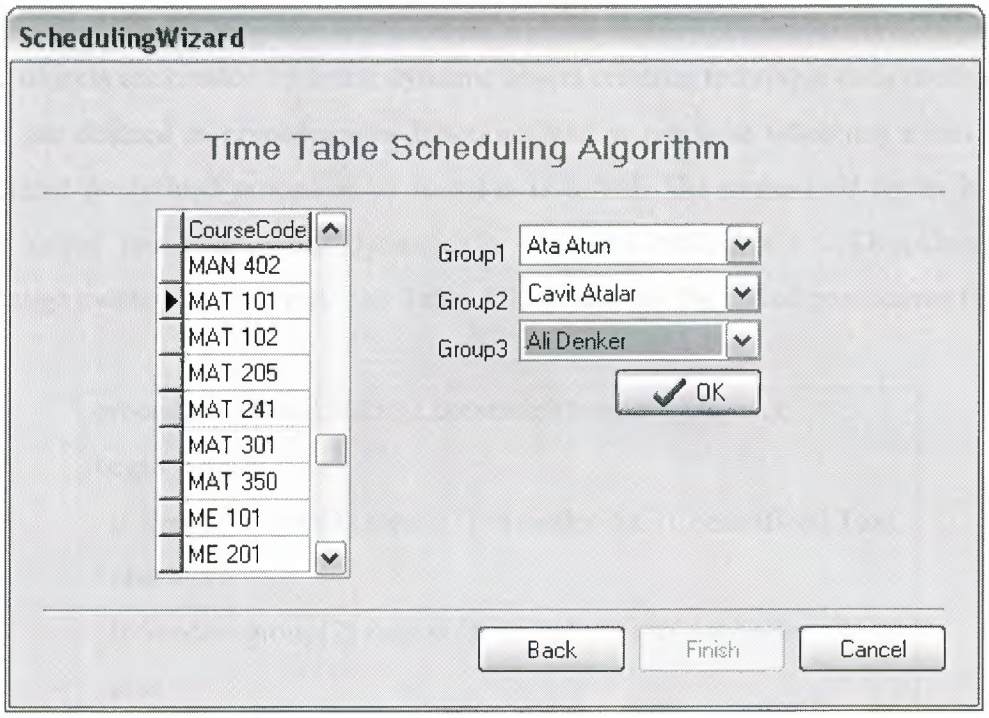


Figure 4.12 The Scheduling Wizard Step4.

This part is designed to create components dynamically. When user select a course from DBGrid, OnCellClick event occurs and goes to the *'tbcoursegroup'* table to find the CourseId of that course, than goes to the *'tblecturer_course'* table to find the lecturers whose are offering the course, and finally creates ComboBoxes, Labels, and the OK button by using dynamic object creating technique (see Table 4.12).

```
begin
        group[x]:=TComboBox.Create(DualListDlg1);
        group[x].Left:=253;
        group[x].Top:=24*(x-1)+80;
        group[x].Width:=121;
        group[x].Height:=21;
        group[x].Parent:=DualListDlg1;
        group[x].Style:=csDropDownList;
        group[x].OnDropDown:=drpdwnclick;
        group[x].OnChange:=combosel;
end;
```

Table4.12 Sample Dynamic objects creating.

While objects are created by using dynamic object creating technique their methods and events are defined as procedures or functions, and at run time when any event occurs the related predefined procedure or function is called. The underlined codes in Table shows called procedure when dynamically created ComboBox's OnDropDown, and OnChange events are occurred. The Table 4.13 illustrates the called procedures for

```
procedure TDualListDlg1.combosel(Sender:TObject);
begin
 if Sender=group[1] then s[1]:=(sender As TComboBox).Text
 else
 if Sender=group[2] then s[2]:=(sender As TComboBox).Text
 else
 if Sender=group[3] then s[3]:=(sender As TComboBox).Text
 else
 if Sender=group[4] then s[4]:=(sender As TComboBox).Text
 else
 if Sender=group[5] then s[5]:=(sender As TComboBox).Text
 else
 if Sender=group[6] then s[6]:=(sender As TComboBox).Text;
end;
```

Table4.13 Code for called Procedure by the dynamic object.

As far as ser finishes this job the finish button will appear at the right bottom of the wizard, it means all things are finished for user to do the rest of the operations are job of the program.

4.2.3.5. Step5, Finish the job

When the finish button is clicked the program begins to assign courses to the most optimal place on the timetable. This operation takes maybe few minutes according to the courses offered and their group amounts. To replacement of courses takes some stages such as opening all related actual and dummy tables, trying to replace using

primary specially developed algorithm or secondary, etc. (see the Figure shown in Figure 4.13 to have a knowledge about the step5);

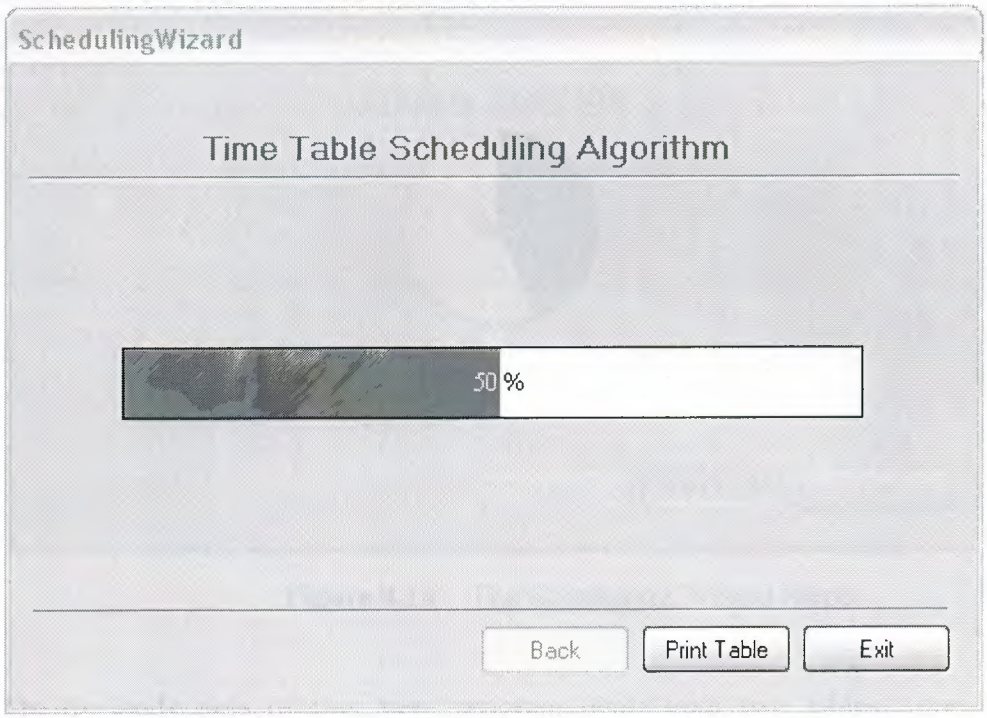


Figure 4.13 The Scheduling Wizard Step5.

The algorithm used in this project is more complex then the other part of the program. Algorithm is specially designed only for this project using fundamentals of algorithm developing, and mathematical knowledge.

The program is ready to construct the timetable in printable format using the Delphi's Quick report component, when it finishes the replacement process. There is one more step to finish the wizard is printing table. The next topic will discuss the printing process.

4.2.3.6. Step6, Printing the Timetable

This is the last action for both user and program to see the timetable in well designed printable format. Clicking the button *Print Table* will perform the operation that the user needs at all (see the Figure 4.14 about user interface of the printing table).

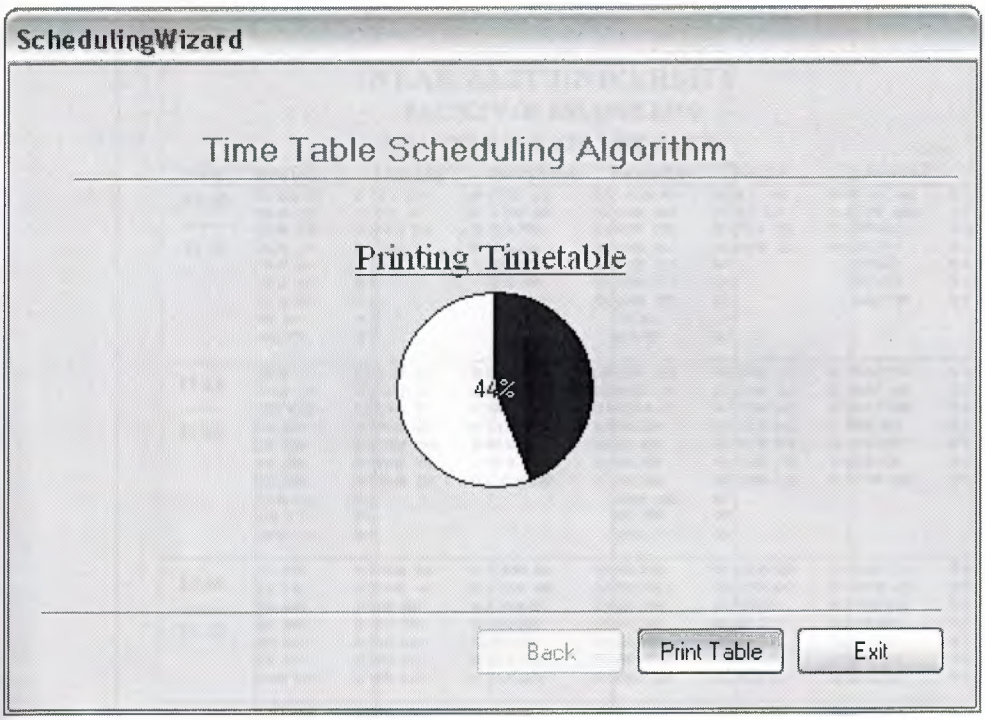
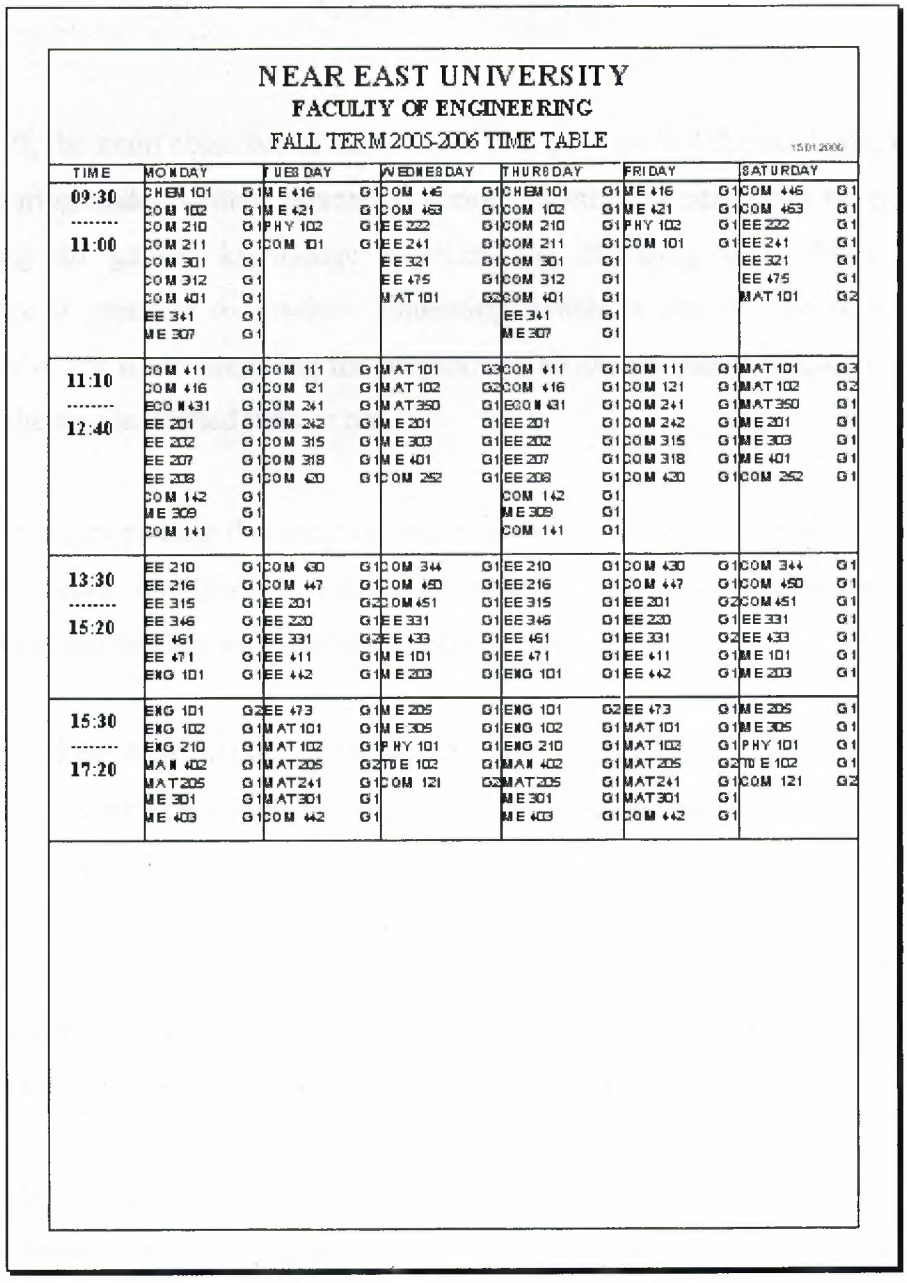


Figure 4.14 The Scheduling Wizard Step6.

On the code side of this step, program deals with two tables one is the table '*tbreplacecourse'* and the other is *'tbscheduler'* for to replace all data from the list that is not easy to understand for human to list that is in table view and easy to understand.

The algorithm can be investigated by looking at the Appendix B.

A sample report of the program is seen in Figure 4.15;



NEAR EAST UNIVERSITY

FACULTY OF ENGINEERING

FALL TERM 2005-2006 TIME TABLE

15/01/2006

TIME	MONDAY		TUESDAY		WEDNESDAY		THURSDAY		FRIDAY		SATURDAY	
09:30	CHEM 101	G1	ME 416	G1	COM 446	G1	CHEM 101	G1	ME 416	G1	COM 446	G1
.....	COM 102	G1	ME 421	G1	COM 463	G1	COM 102	G1	ME 421	G1	COM 463	G1
11:00	COM 210	G1	PHY 102	G1	EE 222	G1	COM 210	G1	PHY 102	G1	EE 222	G1
	COM 211	G1	COM 101	G1	EE 241	G1	COM 211	G1	COM 101	G1	EE 241	G1
	COM 301	G1			EE 321	G1	COM 301	G1			EE 321	G1
	COM 312	G1			EE 475	G1	COM 312	G1			EE 475	G1
	COM 401	G1			MAT 101	G2	COM 401	G1			MAT 101	G2
	EE 341	G1					EE 341	G1				
	ME 307	G1					ME 307	G1				
11:10	COM 411	G1	COM 111	G1	MAT 101	G3	COM 411	G1	COM 111	G1	MAT 101	G3
.....	COM 416	G1	COM 121	G1	MAT 102	G2	COM 416	G1	COM 121	G1	MAT 102	G2
12:40	ECON 431	G1	COM 241	G1	MAT 350	G1	ECON 431	G1	COM 241	G1	MAT 350	G1
	EE 201	G1	COM 242	G1	ME 201	G1	EE 201	G1	COM 242	G1	ME 201	G1
	EE 202	G1	COM 315	G1	ME 303	G1	EE 202	G1	COM 315	G1	ME 303	G1
	EE 207	G1	COM 318	G1	ME 401	G1	EE 207	G1	COM 318	G1	ME 401	G1
	EE 208	G1	COM 420	G1	COM 252	G1	EE 208	G1	COM 430	G1	COM 252	G1
	COM 142	G1					COM 142	G1				
	ME 309	G1					ME 309	G1				
	COM 141	G1					COM 141	G1				
13:30	EE 210	G1	COM 430	G1	COM 344	G1	EE 210	G1	COM 430	G1	COM 344	G1
.....	EE 216	G1	COM 447	G1	COM 450	G1	EE 216	G1	COM 447	G1	COM 450	G1
15:20	EE 315	G1	EE 201	G2	COM 451	G1	EE 315	G1	EE 201	G2	COM 451	G1
	EE 346	G1	EE 220	G1	EE 331	G1	EE 346	G1	EE 220	G1	EE 331	G1
	EE 461	G1	EE 331	G2	EE 433	G1	EE 461	G1	EE 331	G2	EE 433	G1
	EE 471	G1	EE 411	G1	ME 101	G1	EE 471	G1	EE 411	G1	ME 101	G1
	ENG 101	G1	EE 442	G1	ME 203	G1	ENG 101	G1	EE 442	G1	ME 203	G1
15:30	ENG 101	G2	EE 473	G1	ME 205	G1	ENG 101	G2	EE 473	G1	ME 205	G1
.....	ENG 102	G1	MAT 101	G1	ME 305	G1	ENG 102	G1	MAT 101	G1	ME 305	G1
17:20	ENG 210	G1	MAT 102	G1	PHY 101	G1	ENG 210	G1	MAT 102	G1	PHY 101	G1
	MAN 402	G1	MAT 205	G2	TDE 102	G1	MAN 402	G1	MAT 205	G2	TDE 102	G1
	MAT 205	G1	MAT 241	G1	COM 121	G2	MAT 205	G1	MAT 241	G1	COM 121	G2
	ME 301	G1	MAT 301	G1			ME 301	G1	MAT 301	G1		
	ME 403	G1	COM 442	G1			ME 403	G1	COM 442	G1		

Figure 4.15 Sample report.

CONCLUSION

First of all, the main objective of this project is to put forward the academic knowledge gained during undergraduate education period. In other words it can be explained as combining all gained knowledge together and changing them form theoretical knowledge to practice to produce something which is useful. The only and most important worry while preparing the project was about whether it would be possible to produce the best in limited time or not.

The other aim of picking this topic is forcing the mind on algorithms and improving the ability of multi directional thinking on matters, beside designing user interface applications and dealing with the hard program codes.

There were lots of difficulties encountered while writing design analysis or designing database of the program. And the most hard and time consumption part of the project was the algorithms which are for making analysis and comparisons between stated criterions for several times to find the optimal solution. Algorithms become longer and larger in time, by adding the every criterion and gone under not controllable situation. Fortunately over all investigations and studying on them acceptable solution is brought up to present and share with whom they are interested in this topic.

All the things required by the project supervisor for this project is tried to filled it up. It would be better if one thing is also considered is that room capacity. If room capacity is considered in this project it could be applicable for all educational associations. Although having plenty desire to add the room capacity to the program, it could not be possible. If it is added to the program, thinking about algorithms would become at least two times harder.

Finally, all these project topics, or problems are dealt and solved in some way beforehand by other developers. These are just for improving the project owners and to put another solution forward.

ABREVIATIONS USED IN THE DOCUMENT

RAD	:	Rapid Application Development
IDE	:	Integrated Development Environment
GUI	:	Graphical User Interface
OOP	:	Object Oriented Programming
NF	:	Normal Form
ODBC	:	Object Database Connectivity
API	:	Application Programming Interface
BDE	:	Borland Database Engine
ADO	:	ActiveX Data Objects
ODBC	:	Open Data Base Connectivity
MDAC	:	Microsoft Data Access Component
UI	:	User Interface
VCL	:	Visual Component Library
CLX	:	Component Library for Cross Platform
Kylix	:	Delphi's adaptation for Unix Based applications
SQL	:	structured Query Language
DB/db	:	Database
DLL	:	Dynamic Link Library
DDL	:	Data Definition Language
DML	:	Data Manipulation Language
DCL	:	Data Control Language
COM	:	Component Object Model
IDAPI	:	Integrated Database Application Programming Interface
RDS	:	Remote Data Services
MSDE	:	Microsoft Database Engine
OLAP	:	Online Analytical Processing
ADOMD	:	ADO Multi-Dimensional
dbGo	:	Component package for ADO
ANSI	:	American National Standard Institute
DBMS	:	Database Management System
RDBMS	:	Relational Database Management System
ISO	:	International Organization for Standardization

REFERENCES

Books

Borland Delphi 7 by İhsan KARAGÜLLE, Türkmen Kitabevi (2003)

Borland Delphi 7 by Zeydin PALA, Türkmen Kitabevi (YYÜ,Van, may-2003)

Borland Delphi 8 & 2005 For .NET FrameWork by Memik YANIK (Seçkin, 2005)

E-books

Mastering Delphi 7 by Marco CANTU, Sybex (2003)

Database Developer's Guide by Borland International.Inc.

Using Microsoft Office 2000 by Edd Bott, Wbody Leonhard (1999)

Visual Library References book by Borland International Inc.

Using ADO from Delphi

The Internet

www.delphi.about.com

www.microsoft.com/data.

www.borland.com

http://internetsecurity.agrreview.com

The NEU library

APPENDIX A

Program database tables

tbcourse : Table

CourseId	CourseCode	CourseTitle	DeptId	Year	term	PreReq
21	COM 111	INTRO. TO COMP. ENG.	16	1	1	
22	COM 121	DISCRETE STRUCTURES	16	1	2	
23	COM 141	INTRO. TO PROGRAMMING	16	1	1	
24	COM 211	LOGIC DESIGN	16	2	1	22
26	COM 241	DATA STRUCTURES	16	2	1	
27	COM 252	COMPUTER ORGANIZATIONS	16	2	2	
28	COM 301	MICROPROCESSORS	16	3	1	
29	COM 210	OBJECT ORIENTRENTED PROG. 1	16	2	2	
30	COM 312	OPERATING SYSTEMS	16	3	2	
31	COM 315	ALGORITHMS	16	3	1	26
32	COM 318	DATA COMMUNICATIONS	16	3	2	
33	COM 242	D.B.M.S	16	2	2	
34	COM 344	AUTOMOTA THERY	16	3	1	
35	COM 401	MICROPROCESSOR SYSTEMS	16	4	1	
36	COM 411	SOFTWARE ENGINEERING	16	4	1	
37	COM 416	COMPUTER NETWORKS	16	4	2	

Record: 23 of 76

tbcourse_group : Table

CourseId	groups	LecturerId
21	1	0
22	1	0
23	1	0
24	1	0
26	1	0
27	1	0
28	1	0
29	1	0
30	1	0
31	1	0
32	1	0
33	1	0

Record: 37 of 76

tbcourse_status : Table

CourseId	population	CourseCode
21	50	COM 111
22	50	COM 121
23	50	COM 141
24	50	COM 211
26	50	COM 241
27	50	COM 252
28	50	COM 301
29	50	COM 210
30	50	COM 312
31	50	COM 315
32	50	COM 318
33	50	COM 242
34	50	COM 344

Record: 29 of 76

tbdummy1 : Table

ID	CourseId	LecturerId	Groups	DeptId	Year	Term
ber)	0	0		0		

Record: 1 of 1

tbdummy2 : Table

ID	CourseId	LecturerId	Groups	DeptId	Year	Term	PeriodNo
ber)	0	0		0			0

Record: 1 of 1

tbdummycourse : Table

CourseId	groups	LecturerId
0		0

Record: 1 of 1

tbdepartment : Table

DeptId	DeptName	Abbreviation
16	Computer Engi	COM
24	COMMON DEF	C-D
27	Electrical and E	EE
28	Mechanical En	MECH
(AutoNumber)		

Record: 5 of 5

tblecturer_course : Table

ID	CourseId	LecturerId
4	21	2
5	51	2
6	22	3
7	83	3
8	37	4
9	22	4
10	26	5
11	23	5
12	84	5
13	40	6
14	24	6
18	31	8
19	29	8
20	45	8

Record: 27 of 92

tblecturer : Table

LecturerId	LecturerFName	LecturerLName	DeptId	Phone	Mail
2	Doğan	Haktanır	16	( ) -	
3	Firudun	Muradov	16	( ) -	
4	Murat	Tezer	16	( ) -	
5	Okan	Donangil	16	( ) -	
6	Besime	Erin	16	( ) -	
7	Kaan	Uyar	16	( ) -	
8	Rahip	Abiyev	16	( ) -	
9	Tayseer	Alshanableh	16	( ) -	
10	Ümit	Ilhan	16	( ) -	
11	İbrahim	Arkurt	16	( ) -	
12	Doğan	İbrahim	16	( ) -	
13	Adil	amirjanov	16	( ) -	
14	Kemal	Ataman	16	( ) -	
15	İmanov	Elbrus	16	( ) -	
16	Adnan	Khashman	16	( ) -	

Record: 39 of 39

tbreport : Table

Time	Code1	G1	Code2	G2	Code3	G3	Code4	G4	Code5	G5	Code6	G6
	CHEM101	G1	COM 241	G1	COM 411	G1	CHEM101	G1	COM 241	G1	COM 411	G1
	COM 102	G1	COM 242	G1	COM 416	G1	COM 102	G1	COM 242	G1	COM 416	G1
	COM 301	G1					COM 301	G1				
	COM 446	G1					COM 446	G1				
	COM 111	G1	COM 252	G1	COM 420	G1	COM 111	G1	COM 252	G1	COM 420	G1
	COM 121	G1	COM 312	G1	COM 430	G1	COM 121	G1	COM 312	G1	COM 430	G1
	COM 210	G1					COM 210	G1				
	COM 121	G2	COM 315	G1	COM 442	G1	COM 121	G2	COM 315	G1	COM 442	G1
	COM 141	G1	COM 318	G1	COM 463	G1	COM 141	G1	COM 318	G1	COM 463	G1
	COM 101	G1					COM 101	G1				
	COM 142	G1	COM 344	G1	COM 450	G1	COM 142	G1	COM 344	G1	COM 450	G1
	COM 211	G1	COM 401	G1	COM 447	G1	COM 211	G1	COM 401	G1	COM 447	G1

Record: 10 of 16

tbperiods : Table

periodNo	StartTime	FinishTime
1	9:3	11:
2	11:1	12:4
3	13:3	15:2
4	15:3	17:2
5	9:3	11:
6	11:1	12:4
7	13:3	15:2
8	15:3	17:2
9	9:3	11:
10	11:1	12:4

Record: 18 of 20

tbscheduler : Table

CourseId	groups	LecturerId
21	1	
22	1	
22	2	
23	1	
24	1	
26	1	
27	1	
28	1	
29	1	
30	1	
31	1	
32	1	

Record: 21 of 28

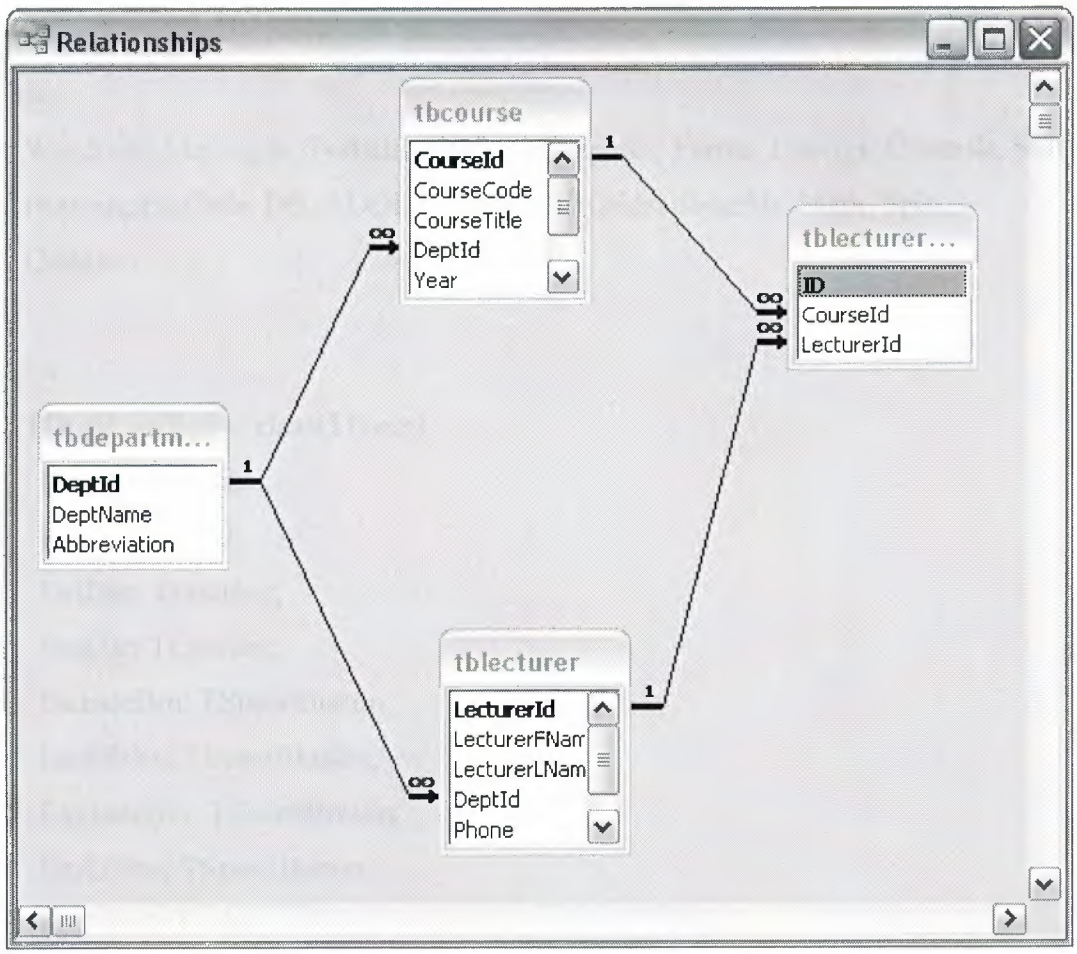
tbreplace_course : Table

CourseId	Groups	LecturerId	PeriodNo
21	1	2	10
22	1	3	10
23	1	5	2
24	1	6	1
26	1	5	5
27	1	7	8
28	1	7	1
29	1	8	1
30	1	10	1
31	1	8	5
32	1	9	6
33	1	10	5
34	1	11	6
35	1	12	1
36	1	13	2
37	1	4	2
38	1	10	6
39	1	13	8
40	1	6	9

Record: 27 of 76

APPENDIX B

Relationships of the database



APPENDIX C

The timetable scheduling wizard form

```
unit Unit11;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Forms, Dialogs, Controls, StdCtrls,
  Buttons, ExtCtrls, DB, ADODB, Grids, DBGrids, dbcgrids, Math, Spin,
  Gauges;

type
  TDualListDlg1 = class(TForm)
    Panel1: TPanel;
    Panel2: TPanel;
    DstList: TListBox;
    SrcList: TListBox;
    IncludeBtn: TSpeedButton;
    IncAllBtn: TSpeedButton;
    ExcludeBtn: TSpeedButton;
    ExAllBtn: TSpeedButton;
    SrcLabel: TLabel;
    DstLabel: TLabel;
    ADOConnection1: TADOConnection;
    ADOTable1: TADOTable;
    ADOTable2: TADOTable;
    ADOTable3: TADOTable;
    DataSource1: TDataSource;
    DataSource2: TDataSource;
    DataSource3: TDataSource;
    Label1: TLabel;
```

Label2: TLabel;
Label3: TLabel;
Label4: TLabel;
Label5: TLabel;
Label6: TLabel;
Label7: TLabel;
Label8: TLabel;
Label9: TLabel;
Label10: TLabel;
Label11: TLabel;
SpeedButton1: TSpeedButton;
SpeedButton2: TSpeedButton;
SpeedButton3: TSpeedButton;
Button1: TButton;
Button2: TButton;
CancelBtn: TButton;
Button3: TButton;
Button4: TButton;
Button5: TButton;
ADOTable1DeptId: TAutoIncField;
ADOTable1DeptName: TWideStringField;
ADOTable1Abbreviation: TWideStringField;
ADOTable2CourseId: TAutoIncField;
ADOTable2CourseCode: TWideStringField;
ADOTable2CourseTitle: TWideStringField;
ADOTable2DeptId: TIntegerField;
ADOTable2Year: TWideStringField;
ADOTable2term: TWideStringField;
ADOTable2PreReq: TWideStringField;
ADOTable3LecturerId: TAutoIncField;
ADOTable3LecturerFName: TWideStringField;
ADOTable3LecturerLName: TWideStringField;
ADOTable3DeptId: TIntegerField;

ADOTable3Phone: TWideStringField;
ADOTable3Mail: TWideStringField;
Label12: TLabel;
ADOTable4: TADOTable;
DataSource4: TDataSource;
Bevel1: TBevel;
ADOQuery1: TADOQuery;
Panel3: TPanel;
Button6: TButton;
Button7: TButton;
Button8: TButton;
DBGrid1: TDBGrid;
Label14: TLabel;
Bevel2: TBevel;
Panel4: TPanel;
ADOTable4CourseId: TIntegerField;
ADOTable4population: TSmallintField;
ADOTable4CourseCode: TWideStringField;
Button9: TButton;
Button10: TButton;
Button11: TButton;
Bevel3: TBevel;
DBGrid2: TDBGrid;
Bevel4: TBevel;
ADOTable5: TADOTable;
DataSource5: TDataSource;
ADOTable5CourseId: TIntegerField;
ADOTable5groups: TWideStringField;
ADOTable5LecturerId: TIntegerField;
ADOTable6: TADOTable;
DataSource6: TDataSource;
ADOTable6CourseId: TIntegerField;
ADOTable6LecturerId: TIntegerField;

DBGrid3: TDBGrid;
DataSource7: TDataSource;
SpinEdit1: TSpinEdit;
BitBtn1: TBitBtn;
Label15: TLabel;
ADOTable7: TADOTable;
DataSource8: TDataSource;
ADOTable7CourseId: TIntegerField;
ADOTable7Groups: TWideStringField;
ADOTable7LecturerId: TIntegerField;
ADOTable7PeriodNo: TIntegerField;
ADOTable8: TADOTable;
ADOTable9: TADOTable;
DataSource9: TDataSource;
DataSource10: TDataSource;
ADOTable8CourseId: TIntegerField;
ADOTable8LecturerId: TIntegerField;
ADOTable8Groups: TWideStringField;
ADOTable8DeptId: TIntegerField;
ADOTable8Year: TWideStringField;
ADOTable8Term: TWideStringField;
ADOTable9CourseId: TIntegerField;
ADOTable9LecturerId: TIntegerField;
ADOTable9Groups: TWideStringField;
ADOTable9DeptId: TIntegerField;
ADOTable9Year: TWideStringField;
ADOTable9Term: TWideStringField;
ADOTable9PeriodNo: TIntegerField;
DBGrid4: TDBGrid;
DBGrid5: TDBGrid;
DBGrid6: TDBGrid;
ADOTable10: TADOTable;
ADOTable11: TADOTable;

```
DataSource11: TDataSource;
DataSource12: TDataSource;
ADOTable10CourseId: TIntegerField;
ADOTable10groups: TWideStringField;
ADOTable10LecturerId: TIntegerField;
ADOTable11CourseId: TIntegerField;
ADOTable11groups: TWideStringField;
ADOTable11LecturerId: TIntegerField;
Panel5: TPanel;
Button12: TButton;
Button13: TButton;
Button14: TButton;
Gauge1: TGauge;
Bevel5: TBevel;
Bevel6: TBevel;
Label20: TLabel;
Label21: TLabel;
Bevel7: TBevel;
Label22: TLabel;
Bevel8: TBevel;
Bevel9: TBevel;
Bevel10: TBevel;
Label13: TLabel;
ADOQuery2: TADOQuery;
DataSource13: TDataSource;
ADOTable12: TADOTable;
Gauge2: TGauge;
procedure IncludeBtnClick(Sender: TObject);
procedure ExcludeBtnClick(Sender: TObject);
procedure IncAllBtnClick(Sender: TObject);
procedure ExcAllBtnClick(Sender: TObject);
procedure FormClose(Sender: TObject; var Action: TCloseAction);
procedure FormCreate(Sender: TObject);
```

```
  procedure FormActivate(Sender: TObject);
  procedure SpeedButton1Click(Sender: TObject);
  procedure SpeedButton2Click(Sender: TObject);
  procedure SpeedButton3Click(Sender: TObject);
  procedure Button2Click(Sender: TObject);
  procedure CancelBtnClick(Sender: TObject);
  procedure Button5Click(Sender: TObject);
  procedure Button3Click(Sender: TObject);
  procedure Button4Click(Sender: TObject);
  procedure Button8Click(Sender: TObject);
  procedure Button6Click(Sender: TObject);
  procedure Button11Click(Sender: TObject);
  procedure Button9Click(Sender: TObject);
  procedure Button7Click(Sender: TObject);
  procedure DBGrid2CellClick(Column: TColumn);
  procedure drpdwnclick(Sender: TObject);
  procedure saveclick(Sender: TObject);
  procedure combosel(Sender:TObject);
  procedure DBGrid1CellClick(Column: TColumn);
  procedure BitBtn1Click(Sender: TObject);
  procedure SpinEdit1KeyPress(Sender: TObject; var Key: Char);
  procedure Button10Click(Sender: TObject);
  procedure Button14Click(Sender: TObject);
  procedure Button13Click(Sender: TObject);


private
  { Private declarations }
  i,idcount:Integer;

  s: array [1..6] of String;

  save:TBitBtn;
```

```
  public
    { Public declarations }
    procedure MoveSelected(List: TCustomListBox; Items: TStrings);
    procedure SetItem(List: TListBox; Index: Integer);
    function GetFirstSelection(List: TCustomListBox): Integer;
    procedure SetButtons;
    procedure copytoscheduler;
    procedure copytodummycourse;
    procedure writereplace;
    procedure dummy2;
    procedure dummy1;
    procedure cleandummy2;
    procedure tryloop;
    procedure tryloop2;
    procedure cleanreplacecourse;
    procedure cleandummycourse;
    procedure copy_replacetodummy2;
  end;

var
  DualListDlg1: TDualListDlg1;
  lbl: array [1..6] of TLabel;
  x,prgrs:Integer;
  group: array [1..6] of TComboBox;
  period,iteration,iterationcount:integer;
  boolecount:Boolean;


implementation

uses Unit2, Unit5, Unit6, StrUtils, Variants, Unit4;

{$R *.dfm}
```

```
procedure TDualListDlg1.IncludeBtnClick(Sender: TObject);
var
  Index: Integer;
begin
  Index := GetFirstSelection(SrcList);
  MoveSelected(SrcList, DstList.Items);
  SetItem(SrcList, Index);

  if DstList.Count > 0 then
    Button4.Enabled:=True
  else
    Button4.Enabled:=False;
end;

procedure TDualListDlg1.ExcludeBtnClick(Sender: TObject);
var
  Index: Integer;
begin
  Index := GetFirstSelection(DstList);
  MoveSelected(DstList, SrcList.Items);
  SetItem(DstList, Index);

  if DstList.Count > 0 then
    Button4.Enabled:=True
  else
    Button4.Enabled:=False;

end;

procedure TDualListDlg1.IncAllBtnClick(Sender: TObject);
var
  I: Integer;
```

```
begin
  for I := 0 to SrcList.Items.Count - 1 do
    DstList.Items.AddObject(SrcList.Items[I],
      SrcList.Items.Objects[I]);
  SrcList.Items.Clear;
  SetItem(SrcList, 0);

  if DstList.Count > 0 then
    Button4.Enabled:=True
  else
    Button4.Enabled:=False;

end;

procedure TDualListDlg1.ExcAllBtnClick(Sender: TObject);
var
  I: Integer;
begin
  for I := 0 to DstList.Items.Count - 1 do
    SrcList.Items.AddObject(DstList.Items[I], DstList.Items.Objects[I]);
  DstList.Items.Clear;
  SetItem(DstList, 0);

  if DstList.Count > 0 then
    Button4.Enabled:=True
  else
    Button4.Enabled:=False;

end;

procedure TDualListDlg1.MoveSelected(List: TCustomListBox; Items: TStrings);
var
  I: Integer;
```

```
begin
  for I := List.Items.Count - 1 downto 0 do
    if List.Selected[I] then
    begin
      Items.AddObject(List.Items[I], List.Items.Objects[I]);
      List.Items.Delete(I);
    end;
end;

procedure TDualListDlg1.SetButtons;
var
  SrcEmpty, DstEmpty: Boolean;
begin
  SrcEmpty := SrcList.Items.Count = 0;
  DstEmpty := DstList.Items.Count = 0;
  IncludeBtn.Enabled := not SrcEmpty;
  IncAllBtn.Enabled := not SrcEmpty;
  ExcludeBtn.Enabled := not DstEmpty;
  ExAllBtn.Enabled := not DstEmpty;
end;

function TDualListDlg1.GetFirstSelection(List: TCustomListBox): Integer;
begin
  for Result := 0 to List.Items.Count - 1 do
    if List.Selected[Result] then Exit;
  Result := LB_ERR;
end;

procedure TDualListDlg1.SetItem(List: TListBox; Index: Integer);
var
  MaxIndex: Integer;
begin
  with List do
```

```
  begin
    SetFocus;
    MaxIndex := List.Items.Count - 1;
    if Index = LB_ERR then Index := 0
    else if Index > MaxIndex then Index := MaxIndex;
    Selected[Index] := True;
  end;
  SetButtons;
end;
//-----------------------------------------------------//
procedure TDualListDlg1.FormClose(Sender: TObject;
  var Action: TCloseAction);
begin
Action:=caFree;
end;

procedure TDualListDlg1.FormCreate(Sender: TObject);
begin
//clear all tables which are related with the Scheduling

cleandummy2;

ADOTable8.First;
while ADOTable8.Eof <> True do
begin
   ADOTable8.Delete;
end;

cleanreplacecourse;
cleandummycourse;

ADOTable11.First;
while ADOTable11.Eof <> True do
```

```
begin
   ADOTable11.Delete;
end;

ADOTable5.First;
while ADOTable5.Eof <> True do
begin
   ADOTable5.Delete;
end;

ADOTable4.First;
while ADOTable4.Eof <> True do
begin
   ADOTable4.Delete;
end;

ADOTable12.First;
while ADOTable12.Eof <> True do
begin
   ADOTable12.Delete;
end;


//tbcourse_status - tbtbcourse_group - tbreplace_course
//tbdummy1 - tb dummy2  are already deleted.....

   i:=0;
   Panel2.Visible:=False;
   Panel3.Visible:=False;
   Panel4.Visible:=False;
   Button1.Enabled:=False;
   Button4.Enabled:=False;
   if ADOTable1.RecordCount=0 then
```

```
begin
  Label9.Font.Color:=clRed;
  Label9.Caption:='"+IntToStr(ADOTable1.RecordCount)+'" Record Found ';
end else
begin
  Label9.Font.Color:=clGreen;
  Label9.Caption:='"+IntToStr(ADOTable1.RecordCount)+'" Record Found ';
end;

if ADOTable2.RecordCount=0 then
begin
  Label10.Font.Color:=clRed;
  Label10.Caption:='"+IntToStr(ADOTable2.RecordCount)+'" Record Found ';
end else
begin
  Label10.Font.Color:=clGreen;
  Label10.Caption:='"+IntToStr(ADOTable2.RecordCount)+'" Record Found ';
end;

if ADOTable3.RecordCount=0 then
begin
  Label11.Font.Color:=clRed;
  Label11.Caption:='"+IntToStr(ADOTable3.RecordCount)+'" Record Found ';
end else
begin
  Label11.Font.Color:=clGreen;
  Label11.Caption:='"+IntToStr(ADOTable3.RecordCount)+'" Record Found ';
end;



if (ADOTable1.RecordCount=0) or (ADOTable2.RecordCount=0) or
```

```
    (ADOTable3.RecordCount=0) then
    Button2.Enabled:=False;

end;

procedure TDualListDlg1.FormActivate(Sender: TObject);
begin
if i=1 then
 begin
  ADOTable1.Close;
  ADOTable1.Open;
  ADOTable2.Close;
  ADOTable2.Open;
  ADOTable3.Close;
  ADOTable3.Open;
  if ADOTable1.RecordCount=0 then
  begin
   Label9.Font.Color:=clRed;
   Label9.Caption:=""+IntToStr(ADOTable1.RecordCount)+" Record Found ';
  end else
  begin
   Label9.Font.Color:=clGreen;
   Label9.Caption:=""+IntToStr(ADOTable1.RecordCount)+" Record Found ';
  end;

  if ADOTable2.RecordCount=0 then
  begin
   Label10.Font.Color:=clRed;
   Label10.Caption:=""+IntToStr(ADOTable2.RecordCount)+" Record Found ';
  end else
  begin
   Label10.Font.Color:=clGreen;
   Label10.Caption:=""+IntToStr(ADOTable2.RecordCount)+" Record Found ';
```

```
  end;

  if ADOTable3.RecordCount=0 then
  begin
    Label11.Font.Color:=clRed;
    Label11.Caption:=""+IntToStr(ADOTable3.RecordCount)+'" Record Found ';
  end else
  begin
    Label11.Font.Color:=clGreen;
    Label11.Caption:=""+IntToStr(ADOTable3.RecordCount)+'" Record Found ';
  end;


  if (ADOTable1.RecordCount=0) or (ADOTable2.RecordCount=0) or
     (ADOTable3.RecordCount=0) then
     Button2.Enabled:=False
  else
    Button2.Enabled:=True;
  end;
 i:=0
end;

procedure TDualListDlg1.SpeedButton1Click(Sender: TObject);
begin
 Form2:=TForm2.Create(self);
end;

procedure TDualListDlg1.SpeedButton2Click(Sender: TObject);
begin
 Form5:=TForm5.Create(self);
end;

procedure TDualListDlg1.SpeedButton3Click(Sender: TObject);
```

```
begin
  Form6:=TForm6.Create(self);
  i:=1;
end;

procedure TDualListDlg1.Button2Click(Sender: TObject);
begin
  Button4.Enabled:=False;
  SrcList.Clear;
  DstList.Clear;
  IncludeBtn.Enabled:=True;
  IncAllBtn.Enabled:=True;
  ExcludeBtn.Enabled:=False;
  ExAllBtn.Enabled:=False;
  ADOTable2.First;
  while ADOTable2.Eof<>true do
  begin
    SrcList.Items.add(ADOTable2.FieldValues['CourseCode']);
    ADOTable2.Next;
  end;

  Panel2.Visible:=True;
  Panel1.Visible:=False;
end;

procedure TDualListDlg1.CancelBtnClick(Sender: TObject);
begin
  ADOTable1.Close;
  ADOTable2.Close;
  ADOTable3.Close;
  ADOTable4.Close;
  close;
end;
```

```
procedure TDualListDlg1.Button5Click(Sender: TObject);
begin
 ADOTable1.Close;
 ADOTable2.Close;
 ADOTable3.Close;
 ADOTable4.Close;
 close;

end;

procedure TDualListDlg1.Button3Click(Sender: TObject);
begin
Panel1.Visible:=True;
Panel2.Visible:=False;
SrcList.Clear;
DstList.Clear;
end;

procedure TDualListDlg1.Button4Click(Sender: TObject);
var
 i:Integer;
begin

 ADOTable4.First;
 while ADOTable4.Eof <> True do
    ADOTable4.Delete;

 ADOTable4.Close;
 ADOTable4.Open;
 ADOTable4.First;
 ADOTable2.Close;
 ADOTable2.Open;
```

```
 for i:=0 to DstList.Count - 1 do
 begin
  ADOTable2.First;
  while ADOTable2.eof<> True do
   begin
    if ADOTable2CourseCode.Text=DstList.Items.Strings[i] then
     begin
      ADOTable4.Insert;
      ADOTable4.FieldValues['CourseId']:=StrToInt(ADOTable2CourseId.Text);
      ADOTable4.FieldValues['population']:=50;
      ADOTable4.FieldValues['CourseCode']:=ADOTable2CourseCode.Text;
      ADOTable4.Post;
      ADOTable4.Close;
      ADOTable4.Open;
     end;
    ADOTable2.Next;;
   end;
 end;

Panel3.Visible:=True;
Panel2.Visible:=False;
end;

procedure TDualListDlg1.Button8Click(Sender: TObject);
begin
 ADOTable1.Close;
 ADOTable2.Close;
 ADOTable3.Close;
 ADOTable4.Close;
 close;
end;
```

```
procedure TDualListDlg1.Button6Click(Sender: TObject);
begin
  Panel2.Visible:=True;
  Panel3.Visible:=False;
end;

procedure TDualListDlg1.Button11Click(Sender: TObject);
begin
  ADOTable1.Close;
  ADOTable2.Close;
  ADOTable3.Close;
  ADOTable4.Close;
  close;
end;

procedure TDualListDlg1.Button9Click(Sender: TObject);
var
  freecount:Integer;
begin
if x>1 then
   begin
      for freecount := 1 to x-1 do
        begin
           group[freecount].Free;
           lbl[freecount].Free   ;
        end;
     save.Free;
   end;

  Panel3.Visible:=True;
  Panel4.Visible:=False;

end;
```

```
procedure TDualListDlg1.Button7Click(Sender: TObject);
var
 i,gr:Integer;
begin
 ADOTable4.Close;
 ADOTable4.Open;

 ADOTable5.First;
 while ADOTable5.Eof <> True do
    ADOTable5.Delete;

 ADOTable4.First;
 while ADOTable4.Eof <> True do
  begin
     gr:=ceil(StrToInt(ADOTable4population.Text)/50);
     for  i:= 1 to gr do
      begin
         ADOTable5.Insert;
         ADOTable5.FieldValues['CourseId']:=ADOTable4CourseId.Text;
         ADOTable5.FieldValues['groups']:=IntToStr(i);
         ADOTable5.Post;
         ADOTable5.Close;
         ADOTable5.Open;
      end;
     ADOTable4.Next;
  end;

 Panel4.Visible:=True;
 Panel3.Visible:=False;
end;

procedure TDualListDlg1.DBGrid2CellClick(Column: TColumn);
```

```
var
 freecount:Integer;
begin

 if x>=1 then
  begin
    for freecount := 1 to x-1 do
     begin
       group[freecount].Free;
       lbl[freecount].Free   ;
     end;
   save.Free;
  end;


x:=1;
ADOTable5.First;
while ADOTable5.Eof <> true do
 begin
  if ADOTable4CourseId.Text = ADOTable5CourseId.Text then
    begin
      group[x]:=TComboBox.Create(DualListDlg1);
      group[x].Left:=253;
      group[x].Top:=24*(x-1)+80;
      group[x].Width:=121;
      group[x].Height:=21;
      group[x].Parent:=DualListDlg1;
      group[x].Style:=csDropDownList;
      group[x].OnDropDown:=drpdwnclick;
      group[x].OnChange:=combosel;

      lbl[x]:=TLabel.Create(Panel4);
      lbl[x].Left:=204;
```

```
        lbl[x].Top:=24*(x-1)+72;
        lbl[x].Width:=38;
        lbl[x].Height:=13;
        lbl[x].Parent:=Panel4;
        lbl[x].Caption:='Group'+IntToStr(x);

        x:=x+1;
       end;
     ADOTable5.Next;
   end;

 idcount:=StrToInt(ADOTable4CourseId.Text);

 save:=TBitBtn.Create(DualListDlg1);
 save.Left:=300;
 save.Top:=24*(x-1)+80;
 save.Width:=75;
 save.Height:=25;
 save.Parent:=DualListDlg1;
 save.Kind:=bkOK;
 save.Visible:=True;
 save.OnClick:=saveclick;

 Button10.Enabled:=False;

end;

procedure TDualListDlg1.drpdwnclick(sender: TObject);
begin
 ADOTable6.Close;
 ADOTable6.Open;
 ADOTable6.First;
 (sender As TComboBox).Clear;
```

```
  while ADOTable6.Eof <> True do
   begin
    if ADOTable6CourseId.Text = ADOTable4CourseId.Text then
     begin
      ADOTable3.First;
      while ADOTable3.Eof <> True do
       begin
        if ADOTable3LecturerId.Text = ADOTable6LecturerId.Text then
         begin
         (sender As TComboBox).Items.Add(ADOTable3LecturerFName.Text+'
'+ADOTable3LecturerLName.Text);
         end;
        ADOTable3.Next;;
       end;
     end;
    ADOTable6.Next;
   end;
end;

procedure TDualListDlg1.combosel(Sender:TObject);
begin
 if Sender=group[1] then s[1]:=(sender As TComboBox).Text
 else
 if Sender=group[2] then s[2]:=(sender As TComboBox).Text
 else
 if Sender=group[3] then s[3]:=(sender As TComboBox).Text
 else
 if Sender=group[4] then s[4]:=(sender As TComboBox).Text
 else
 if Sender=group[5] then s[5]:=(sender As TComboBox).Text
 else
 if Sender=group[6] then s[6]:=(sender As TComboBox).Text;
```

```
end;

procedure TDualListDlg1.saveclick(Sender: TObject);
var
 search,c,id,z,freecount:Integer;
 name,surname:String;
 count:Boolean;
begin
   count:=False;

   for c := 1 to x-1 do
    begin
         if group[c].Text='' then
          begin
              count:=False;
              ShowMessage('Select a lecturer for every course group');
              Break;
          end else
               count:=True;
    end;


 for  c:=1  to x-1 do
  begin
  if count=True then
  begin
    ADOTable5.Open;
    Search:=pos(' ',s[c]);
    name:=LeftStr(s[c],search-1);
    surname:=Copy(s[c],search+1,Length(s[c])-search);

    ADOTable3.First;
    while ADOTable3.Eof <> True do
```

```
    begin
      if (ADOTable3LecturerFName.Text=name)  and
        (ADOTable3LecturerLName.Text=surname) then
        id:=StrToInt(ADOTable3LecturerId.Text);
     ADOTable3.Next;
    end;

  ADOTable5.First;
  while ADOTable5.Eof <> True do
   begin
    if (ADOTable5CourseId.Text=IntToStr(idcount)) and
      (ADOTable5groups.Text=IntToStr(c))then
      begin
          ADOTable5.Edit;
          ADOTable5.FieldValues['LecturerId']:=id;
          ADOTable5.Post;
          ADOTable5.Refresh;
      end;
     ADOTable5.Next;
    end;

ADOTable5.Filter:='lecturerId=0';
ADOTable5.Filtered:=True;
if  ADOTable5.RecordCount=0 then
  Button10.Enabled:=True
else
  Button10.Enabled:=False;
ADOTable5.Filtered:=False;

 end;

end;
```

```
end;

procedure TDualListDlg1.DBGrid1CellClick(Column: TColumn);
begin
Label15.Caption:=ADOTable4CourseCode.Text;
SpinEdit1.SetFocus;
SpinEdit1.Value:=ADOTable4.FieldValues['population'];
end;

procedure TDualListDlg1.BitBtn1Click(Sender: TObject);
begin
 if (SpinEdit1.Value<=0) or (SpinEdit1.Value>300) then
  begin
      ShowMessage('Student population should be in range of 1 and  300');
      SpinEdit1.Value:=0;
      SpinEdit1.SetFocus;
  end else
  begin
     ADOTable4.Edit;
     ADOTable4.FieldValues['population']:=SpinEdit1.Value;
     ADOTable4.Post;
     ADOTable4.Close;
     ADOTable4.Open;
     Label15.Caption:='';
     SpinEdit1.Value:=0;
 end;

 ADOTable4.Filter:='population=0';
 ADOTable4.Filtered:=True;
 if  ADOTable4.RecordCount=0 then
    Button7.Enabled:=True
 else
    Button7.Enabled:=False;
```

```
  ADOTable4.Filtered:=False;

end;

procedure TDualListDlg1.SpinEdit1KeyPress(Sender: TObject; var Key: Char);
begin

end;

//coppy procedure form tbcourse_group to tbscheduler
procedure TDualListDlg1.copytoscheduler;
begin
  ADOTable5.First;
  while ADOTable5.Eof <> True do
    begin
        ADOTable11.Insert;
        ADOTable11.FieldValues['CourseId']:=ADOTable5CourseId.Text;
        ADOTable11.FieldValues['LecturerId']:=ADOTable5LecturerId.Text;
        ADOTable11.FieldValues['Groups']:=ADOTable5groups.Text;
        ADOTable11.Post;
        ADOTable11.Close;
        ADOTable11.Open;
        ADOTable5.Next;
    end;
end;
//copy form tbcourse_group to tbscheduler completed.

//coppy procedure form tbcourse_group to tbdummycourse
procedure TDualListDlg1.copytodummycourse;
begin
  ADOTable5.First;
  while ADOTable5.Eof <> True do
```

```
  begin
      ADOTable10.Insert;
      ADOTable10.FieldValues['CourseId']:=ADOTable5CourseId.Text;
      ADOTable10.FieldValues['LecturerId']:=ADOTable5LecturerId.Text;
      ADOTable10.FieldValues['Groups']:=ADOTable5groups.Text;
      ADOTable10.Post;
      ADOTable10.Close;
      ADOTable10.Open;
      ADOTable5.Next;
  end;
end;
//copy form tbcourse_group to tbdummycourse completed.


//writing procedure for tbreplace_course
procedure TDualListDlg1.writereplace;
begin
//writing to tbreplace_course
ADOTable7.Insert;
ADOTable7.FieldValues['CourseId']:=ADOTable10CourseId.Text;
ADOTable7.FieldValues['Groups']:=ADOTable10groups.Text;
ADOTable7.FieldValues['LecturerId']:=ADOTable10LecturerId.Text;
ADOTable7.FieldValues['PeriodNo']:=period;
ADOTable7.Post;
ADOTable7.Close;
ADOTable7.Open;
Gauge1.Progress:=prgrs;
//tbreplce_course write completed

end;

//writing procedure fortbdummy2
procedure TDualListDlg1.dummy2;
```

```
begin
//searching DeptId,Year,Term for tbdummy2
  ADOTable2.First;
  while ADOTable2.Eof <> True do
    begin
      if ADOTable2CourseId.Text = ADOTable10CourseId.Text then
        begin
//writing to tbdummy2
            ADOTable9.Insert;
            ADOTable9.FieldValues['CourseId']:=ADOTable10CourseId.Text;
            ADOTable9.FieldValues['LecturerId']:=ADOTable10LecturerId.Text;
            ADOTable9.FieldValues['Groups']:=ADOTable10groups.Text;
            ADOTable9.FieldValues['DeptId']:=ADOTable2DeptId.Text;
            ADOTable9.FieldValues['Year']:=ADOTable2Year.Text;
            ADOTable9.FieldValues['Term']:=ADOTable2term.Text;
            ADOTable9.FieldValues['PeriodNo']:=period;
            ADOTable9.Post;
            ADOTable9.Close;
            ADOTable9.Open;
            Break;
        end else
      ADOTable2.Next;
    end;
//tbdummy2 write completed

end;

procedure TDualListDlg1.dummy1;
begin
//searching DeptId,Year,Term for tbdummy1
  ADOTable2.First;
  while ADOTable2.Eof <> True do
    begin
```

```
    if ADOTable2CourseId.Text = ADOTable10CourseId.Text then
     begin
//writing to tbdummy1
          ADOTable8.Insert;
          ADOTable8.FieldValues['CourseId']:=ADOTable10CourseId.Text;
          ADOTable8.FieldValues['LecturerId']:=ADOTable10LecturerId.Text;
          ADOTable8.FieldValues['Groups']:=ADOTable10groups.Text;
          ADOTable8.FieldValues['DeptId']:=ADOTable2DeptId.Text;
          ADOTable8.FieldValues['Year']:=ADOTable2Year.Text;
          ADOTable8.FieldValues['Term']:=ADOTable2term.Text;
          ADOTable8.Post;
          ADOTable8.Close;
          ADOTable8.Open;
          Break;
      end else
    ADOTable2.Next;
  end;
//tbdummy1 write completed
end;

// cleaning tbdummy2
procedure TDualListDlg1.cleandummy2;
begin
 ADOTable9.First;
 while ADOTable9.Eof<>True do
    ADOTable9.Delete;
end;
//cleaning completed

//comparison procedure for course replace begins..
procedure TDualListDlg1.tryloop;
begin
 dummy1;
```

```
ADOTable9.First;
while ADOTable9.Eof <> True do
 Begin
  if ADOTable8CourseId.Text <> ADOTable9CourseId.Text then
   begin
    if ADOTable8LecturerId.Text <> ADOTable9LecturerId.Text then
     begin
      if ADOTable8DeptId.Text <> ADOTable9DeptId.Text then
       begin
        if StrToInt(ADOTable9DeptId.Text) <> 24 then
         begin
           ADOTable9.Next;
           boolecount:=True;
         end
        else
         begin
          if ADOTable8Year.Text <> ADOTable9Year.Text then
           begin
             ADOTable9.Next;
             boolecount:=True;
           end
          else
           begin
            if ADOTable8Term.Text <> ADOTable9Term.Text then
             begin
               ADOTable9.Next;
               boolecount:=True;
             end
            else
             begin
               if ADOTable10.Eof <> True then
                 ADOTable10.Next
               else
```

```
                ADOTable10.First;
                boolecount:=False;
                ADOTable8.Delete;
                Break;
            end;
        end;
    end;
  end
 else
  begin
   if ADOTable8Year.Text <> ADOTable9Year.Text then
    begin
      ADOTable9.Next;
      boolecount:=True;
    end
   else
    begin
     if ADOTable8Term.Text <> ADOTable9Term.Text then
      begin
        ADOTable9.Next;
        boolecount:=True;
      end
     else
      begin
        if ADOTable10.Eof <> True then
          ADOTable10.Next
        else
          ADOTable10.First;
        boolecount:=False;
        ADOTable8.Delete;
        Break;
      end;
    end;
```

```
          end;
        end
      else
        begin
          if ADOTable10.Eof <> True then
            ADOTable10.Next
          else
            ADOTable10.First;
          boolecount:=False;
          ADOTable8.Delete;
          Break;
        end;
      end
    else
      begin
        if ADOTable10.Eof <> True then
          ADOTable10.Next
        else
          ADOTable10.First;
        boolecount:=False;
        ADOTable8.Delete;
        Break;
      end;
  End;

end;
// comparison for replacing ends...

//Second comparison procedure for course replace begins..
procedure TDualListDlg1.tryloop2;
begin
 dummy1;
 ADOTable9.First;
```

```
while ADOTable9.Eof <> True do
 Begin
  if ADOTable8CourseId.Text <> ADOTable9CourseId.Text then
   begin
    if ADOTable8LecturerId.Text <> ADOTable9LecturerId.Text then
     begin
      if ADOTable8DeptId.Text <> ADOTable9DeptId.Text then
       begin
        if StrToInt(ADOTable9DeptId.Text) <> 24 then
         begin
           ADOTable9.Next;
           boolecount:=True;
         end
        else
         begin
          if ADOTable8Year.Text <> ADOTable9Year.Text then
           begin
             ADOTable9.Next;
             boolecount:=True;
           end
          else
           begin
            if ADOTable8Term.Text <> ADOTable9Term.Text then
             begin
                ADOTable9.Next;
                boolecount:=True;
             end
            else
             begin
                ADOTable10.Next;
                boolecount:=False;
                ADOTable8.Delete;
                Break;
```

```
            end;
          end;
        end;
      end
    else
      begin
        if ADOTable8Year.Text <> ADOTable9Year.Text then
          begin
            ADOTable9.Next;
            boolecount:=True;
          end
        else
          begin
            if ADOTable8Term.Text <> ADOTable9Term.Text then
              begin
                ADOTable9.Next;
                boolecount:=True;
              end
            else
              begin
                ADOTable10.Next;
                boolecount:=False;
                ADOTable8.Delete;
                Break;
              end;
          end;
      end;
  end
else
  begin
    ADOTable10.Next;
    boolecount:=False;
    ADOTable8.Delete;
```

```
            Break;
          end;
        end
      else
        begin
            ADOTable10.Next;
            boolecount:=False;
            ADOTable8.Delete;
            Break;
        end;
    End;

end;
//Second comparison for replacing ends...

// cleaning tbreplace_course begins...
procedure TDualListDlg1.cleanreplacecourse;
begin
  ADOTable7.First;
  while ADOTable7.Eof<>True do
     ADOTable7.Delete;
end;
// cleaning tbreplace_course completed...

// cleaning tbdummycourse begins...
procedure TDualListDlg1.cleandummycourse;
begin
  ADOTable10.First;
  while ADOTable10.Eof<>True do
     ADOTable10.Delete;
end;
// cleaning tbdummycourse completed...
```

```
// copy from tbreplace_course to tbdummy2 begins..
procedure TDualListDlg1.copy_replacetodummy2;
begin
 ADOTable7.First;
 while ADOTable7.Eof <> True do
  begin
   if StrToInt(ADOTable7PeriodNo.Text)=period then
    begin
        //searching DeptId,Year,Term for tbdummy1
        ADOTable2.First;
        while ADOTable2.Eof <> True do
         begin
          if ADOTable2CourseId.Text = ADOTable7CourseId.Text then
           begin
             //writing to tbdummy2
             ADOTable9.Insert;
             ADOTable9.FieldValues['CourseId']:=ADOTable7CourseId.Text;
             ADOTable9.FieldValues['LecturerId']:=ADOTable7LecturerId.Text;
             ADOTable9.FieldValues['Groups']:=ADOTable7groups.Text;
             ADOTable9.FieldValues['DeptId']:=ADOTable2DeptId.Text;
             ADOTable9.FieldValues['Year']:=ADOTable2Year.Text;
             ADOTable9.FieldValues['Term']:=ADOTable2term.Text;
             ADOTable9.Post;
             ADOTable9.Close;
             ADOTable9.Open;
             Break;
           end else
          ADOTable2.Next;
         end;
    end;
   ADOTable7.Next;
  end;
```

```
end;
// copy from tbreplace_course to tbdummy2 completed...

procedure TDualListDlg1.Button10Click(Sender: TObject);
var
 freecount,syscount:Integer;
begin

 Panel4.Visible:=False;
 Panel5.Visible:=True;

 if x>=1 then
  begin
    for freecount := 1 to x-1 do
     begin
       group[freecount].Free;
       lbl[freecount].Free   ;
     end;
   save.Free;
  end;

 ADOTable1.Open;    ////////////////////////////
 ADOTable2.Open;    //                        //
 ADOTable3.Open;    //                        //
 ADOTable4.Open;    //                        //
 ADOTable5.Open;    //                        //
 ADOTable6.Open;    // OPEN ALL RELATED TABLES //
 ADOTable7.Open;    //                        //
 ADOTable8.Open;    //                        //
 ADOTable9.Open;    //                        //
 ADOTable10.Open;    //                        //
 ADOTable11.Open;    ////////////////////////////
```

```
copytoscheduler; //copy from tbcourse_group to tbscheduler

boolecount:=False;
syscount:=0;
period:=1;
iteration:=1;
iterationcount:=Floor(ADOTable5.RecordCount/12);

ADOTable11.First;
WHILE ADOTable11.Eof <> True DO
 BEGIN
  prgrs:=1;
  with Gauge1 do
   begin
    MinValue:=0;
    MaxValue:=ADOTable5.RecordCount;
    ShowText:=True;
    Gauge1.Progress:=prgrs;
   end;


  copytodummycourse;
  ADOTable10.First;
  while ADOTable10.Eof <> True do
   begin
       if ADOTable10CourseId.Text = ADOTable11CourseId.Text then
        begin
            boolecount:=False;
            syscount:=0;
            period:=1;
            iteration:=1;
            While period <= 12 Do
            Begin
```

```
while iteration <= iterationcount do
 begin
     if iteration = 1 then
      begin
          prgrs:=prgrs+1;
          writereplace;
          dummy2;
          ADOTable10.Delete;
          if ADOTable10.Eof = True then
           ADOTable10.First;
          iteration:=iteration+1;
          syscount:=1;
      end
     else
      begin
          tryloop;

          if boolecount = True then
           begin
               prgrs:=prgrs+1;
               writereplace;
               dummy2;
               ADOTable8.Delete;
               ADOTable10.Delete;
               if ADOTable10.Eof = True then
                ADOTable10.First;
               boolecount:=False;
               iteration:=iteration+1;
               syscount:=1;
           end else
            syscount:=0;
      end;
      //**/*/*/*/*
```

```
        if (syscount = 0) and (ADOTable10.Eof = True) then
        iteration:=iterationcount;
            //**/*/*/*/*
      end;
    cleandummy2;
    period:=period+1;
    iteration:=1;
End;

if ADOTable10.RecordCount > 0 then
 begin
    ADOTable10.First;
    period:=1;
    cleandummy2;
    While ADOTable10.Eof <> True Do
     Begin
      While period <=12 Do
       Begin
          copy_replacetodummy2;
          tryloop2;

          if boolecount = True then
           begin
              prgrs:=prgrs+1;
              writereplace;
              ADOTable8.Delete;
              ADOTable10.Delete;
              period:=1;
              boolecount:=False;
              Break;
           end;
          period:=period+1;
          cleandummy2;
```

```
                    End;

                End;
              ///-----2
            end;
          Break;
        end
        else
          ADOTable10.Next;
    end;

    if ADOTable10.RecordCount > 0 then
     begin
        cleanreplacecourse;
        cleandummycourse;
        cleandummy2;
        if ADOTable8.RecordCount > 0 then
          ADOTable8.Delete;
        ADOTable11.Next;
     end
    else
      Break;

  END;

  if ADOTable10.RecordCount > 0 then
   ShowMessage('NOT ALL RECORDS CAN BE REPLACED')
  else
   ShowMessage('ALL RECORDS REPLACED SUCCESSFULLY');

Gauge1.Visible:=False;

ADOTable1.Close;
```

```
ADOTable2.Close;
ADOTable3.Close;
ADOTable4.Close;
ADOTable5.Close;
ADOTable6.Close;
ADOTable7.Close;
ADOTable8.Close;
ADOTable9.Close;
ADOTable10.Close;
ADOTable11.Close;

//**********************************************************************
******************\\
end;


procedure TDualListDlg1.Button14Click(Sender: TObject);
begin
ADOTable1.Close;
ADOTable2.Close;
ADOTable3.Close;
ADOTable4.Close;
ADOTable5.Close;
ADOTable6.Close;
ADOTable7.Close;
ADOTable8.Close;
ADOTable9.Close;
ADOTable10.Close;
ADOTable11.Close;
ADOTable12.Close;
ADOConnection1.Close;
Close;
end;
```

```
procedure TDualListDlg1.Button13Click(Sender: TObject);
var
  per,courseId,cnt,cg,it,perish,maxi,prgrs2 : Integer;
  Time,Groups : String;
  Code : array [1..6] of String;
  G : array [1..6] of String;
begin

ADOTable12.Open;
ADOTable7.Open;


  maxi:=ADOTable7.RecordCount;
  prgrs2:=0;
  Gauge2.Visible:=True;
  with Gauge2 do
    begin
      MinValue:=0;
      MaxValue:=maxi;
      ShowText:=True;
      Gauge2.Progress:=prgrs2;
    end;

perish := 0;

WHILE ADOTable7.RecordCount >0 DO
BEGIN
perish :=perish+1;
While (true) do
BEGIN
  per := perish;
  cnt := 1;
```

```
it := 0;
While per <= 12 do
 Begin
    ADOQuery2.Close;
    ADOQuery2.SQL.Clear;
    ADOQuery2.SQL.Add('SELECT tbcourse.CourseCode
,tbreplace_course.CourseId, tbreplace_course.Groups as Groups');
    ADOQuery2.SQL.Add('FROM tbcourse, tbreplace_course');
    ADOQuery2.SQL.Add('WHERE tbcourse.CourseId=tbreplace_course.CourseId');
    ADOQuery2.SQL.Add('AND tbreplace_course.PeriodNo=period');
    ADOQuery2.Parameters.ParamByName('period').Value:=per;
    ADOQuery2.Open;
    ADOQuery2.ExecSQL;

    if ADOQuery2.RecordCount = 0 then
      begin
      per := per+4;
      inc(cnt);
      it := it+1;
      end
    else
      begin
        Code[cnt] := ADOQuery2.FieldValues['CourseCode'];
        G[cnt] := 'G'+ADOQuery2.FieldValues['Groups'];

        Code[cnt+3] := ADOQuery2.FieldValues['CourseCode'];
        G[cnt+3] := 'G'+ADOQuery2.FieldValues['Groups'];

        courseId := ADOQuery2.FieldValues['CourseId'];
        Groups := ADOQuery2.FieldValues['Groups'];

        ADOTable7.First;
        while ADOTable7.Eof <> True do
```

```
            begin
              if (StrToInt(ADOTable7CourseId.Text) = courseId) and
                (ADOTable7Groups.Text = Groups) then
                    ADOTable7.Delete
              else
                ADOTable7.Next;
            end;

          Inc(cnt);
          per:=per+4;
      end;
  End;

if it = 3 then
  begin
      ADOTable12.Insert;
      ADOTable12.FieldValues['Time'] :='_____' ;
      ADOTable12.FieldValues['Code1'] := '_____';
      ADOTable12.FieldValues['G1'] := '____';
      ADOTable12.FieldValues['Code2'] := '_____';
      ADOTable12.FieldValues['G2'] := '____';
      ADOTable12.FieldValues['Code3'] := '_____';
      ADOTable12.FieldValues['G3'] := '____';
      ADOTable12.FieldValues['Code4'] := '_____';
      ADOTable12.FieldValues['G4'] := '____';
      ADOTable12.FieldValues['Code5'] := '_____';
      ADOTable12.FieldValues['G5'] := '____';
      ADOTable12.FieldValues['Code6'] := '_____';
      ADOTable12.FieldValues['G6'] := '____';
      ADOTable12.Post;
      ADOTable1.Close;
      ADOTable1.Open;
      prgrs2:=prgrs2+1;
```

```
        Gauge2.Progress:=prgrs2;
        cnt := 1;
        Break;
    end
  else
    begin
        ADOTable12.Insert;
        ADOTable12.FieldValues['Time'] :=" ;
        ADOTable12.FieldValues['Code1'] := Code[1];
        ADOTable12.FieldValues['G1'] := G[1];
        ADOTable12.FieldValues['Code2'] := Code[2];
        ADOTable12.FieldValues['G2'] := G[2];
        ADOTable12.FieldValues['Code3'] := Code[3];
        ADOTable12.FieldValues['G3'] := G[3];
        ADOTable12.FieldValues['Code4'] := Code[4];
        ADOTable12.FieldValues['G4'] := G[4];
        ADOTable12.FieldValues['Code5'] := Code[5];
        ADOTable12.FieldValues['G5'] := G[5];
        ADOTable12.FieldValues['Code6'] := Code[6];
        ADOTable12.FieldValues['G6'] := G[6];
        ADOTable12.Post;
        ADOTable1.Close;
        ADOTable1.Open;
        prgrs2:=prgrs2+1;
        Gauge2.Progress:=prgrs2;
        cnt := 1;
        it := 0;
    end;

for cg:=0 to 6 do
    begin
        Code[cg]:='';
        G[cg]:='';
```

```
    end;
  END;
  END;
  ADOTable7.Close;
  ADOTable12.Close;
  ADOTable12.Open;
  ADOConnection1.Close;
  ADOConnection1.Open;
  Form4.QuickRep1.Refresh;
  Form4.QuickRep1.Preview;
  Gauge2.Visible:=False;
  Close;
  end;

  end.
```