NEAR EAST UNIVERSITY

Faculty of Engineering

Department of Computer Engineering

Design of a Pharmacy Automation Program With Delphi and Paradox

Graduation Project
COM 400

Students: Hakan Tuna (991014)
Turgut Tuna (981426)

Supervisor: Halil Adahan

Lefkoşa - 2004

ACKNOWLEDGEMENTS

First we want to thank Mr.Halil Adahan to be our advisor. Under this guidance, we successfully overcome many difficulties and learn a lot about Delphi programming with Paradox database .In each discussion, he explained our questions patiently, and we felt our quick progress from his advises. He always helps us a lot either in our study. We asked him many questions in Delphi progragramming skills and Database applications and he always answered our questions quickly and in detail.

Special thanks for Çağlan PHARMACY. With their kind help ,we understand the main targets, before we designing the Pharmacy Automation program. Thanks for Faculty of Engineering for having such a good computational environment.

We also want to thank our friends in Near East University: Cem Uludağ and Tunç Samurkaş. Being with them make our 4 years in NEU of fun.

Finally, We want to thank our family, especially my parent. Without their endless support and love for us, we would never achieve our current position. We wish our mother lives happily always, and our father in the heaven be proud of us.

ABSTRACT

Delphi is the premier Windows development environment. Based upon Object Pascal, Delphi is the first development tool to combine a powerful Object Oriented language with a Rapid Application Development (RAD) Environment, bringing us the power to create high quality Windows applications with short development times.

Delphi is with full support for classes, inheritance, polymorphism, pointer manipulation, and a host of other features.

With its RAD environment, Delphi allows us to create user interfaces in a small fraction of the time it takes with most C++ packages. Delphi's RAD development environment is based on Borland's own Visual Component Library (VCL).

Delphi includes several dozen VCL components encapsulating every type of control, including buttons, timers, listboxs, OLE containers, multimedia tools, common dialog boxes, tabbed pages, Database applications ,BDE tools and lots more. But we are not limited to just these components. Delphi allows us to build custom components and add them into the component panel. Added components work seamlessly within the IDE.

Delphi is also a fully featured database development tool. Delphi uses the Borland Database Engine (BDE), Borland's high performance, high level programming database access system. Several of the VCL components are designed to interface tightly with the BDE. Also the Borland Database Engine has native VCL components to direct access to ODBC and drivers like Oracle, My SQL AND MS SQL.

Delphi can also build and use DLL's, so we can integrate our program with other DLL made with any language. The Delphi compiler also supports inline assembly for high performance optimization, or low level access programming.

TABLE OF CONTENTS

LIST OF FIGURES

Chapter I: Introduction

1.1 Database Structures

Database is a storage which electronic storage for data. In other words database is an electronic storage of data. It is a depository that stores information about different things and also contains relationships among those different things.

Complex and complicated file structures and extra more files between relations or relationships and users access to files in those situations we see these insufficient situations for traditional file system. To solve this matter for data manipulation to hide the data and access the data with the new software technologies developed and directed to DataBase Management System. In DBMS approach data access and data hide are independent to data access apply or application programs. Use of classic file's difference is that the registration designing and file structure's any little variation is to cause the application programs variations and a new again design it.

Database Systems are evaluate the computer systems very important component. Database Management Systems are to be formed, created and organized with each other related data and United of Programs or Community of Programs.

Data community is evaluated Database. Database is the environment about the company's informations on it. Database Systems the environments about the data heaps orderly holds and those data used with various softwares to manages.

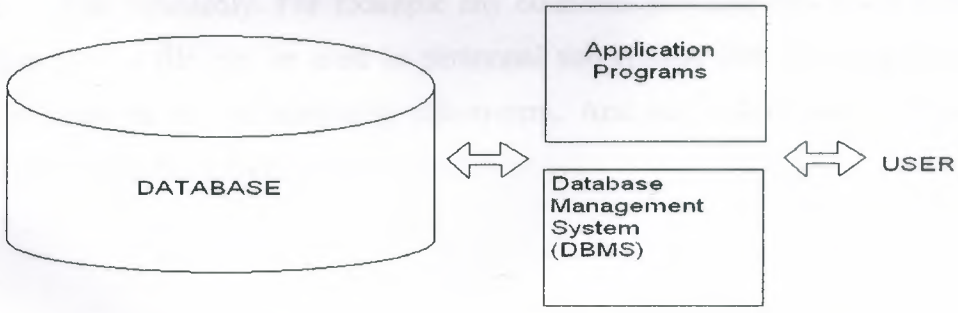

Figure 1.1.1: Relationship Between Database , Database Management System and User.

Database design requires to create entity sets, each describing a set of related entities. Design also requires to establish all the relationships between entity and sets within the database. The different database management software packages handle the creation and the use of relationships in different manners. Depending upon the type of interaction, the relationships are classified into three categories or relationships have three characteristics:

1. One-to-one relationship: A one-to-one relationship is written as 1:1 in short form. 1:1 exists between two entity sets, X and Y, if an entity in entity set X has only one matching entity in entity set Y, the same again for Y.
2. One-to-many relationship: A one-to-many relationship is written as 1:M in short form. It exists between two entity sets, X and Y, if an entity in entity set X has only one matching entity in entity set Y, but an entity in entity set Y has many matching entities in entity set X.
3. Many-to-many relationship: A many-to-many relationship is written as M:N in short form. It exists between two entity sets, X and Y, if an entity in entity set X has many matching entities in entity set Y and an entity in entity set Y has many matching entities in entity set X.

1.1.1 Database System Superiority

Using the Database has many superiorities according to using the traditional file we see and we approve. Database Systems very important benefits are explain below!

1.1.1.1 To Prevent the Data Repeats

Traditional file system's using applications , for each application parts data separately holding. Applications are divided sub-systems and for each sub-system has own data files. Those data repeatedly. For Example any countries province codes and province names formed in a file can be used in personnal sub-system. But, however the same file's copy must be in the marketing sub-system. And any other places it must be repeatedly the same file informations.

Database Systems to consider applications plan and project in entire, establish relations in sub-systems and for many applications project and planning datas in the same data base use with commonly using. Every applications to need datas each other in whole structure.For that reason data source is plans one, for another word in this system which names Database system have a one data source and with this data repeat is protected and prevented.

1.1.1.2 To Provide DATA Consistence

Database Systems have very important superiority is to provide data integrity. Data integrity is explain the data's truth and consistent. Same kind of restrictions can use in Database's to provide data as consistent and truth in integrity work with wholly.

For example: Enter the student information's birth province's code 100 value registered, for error information matter this enter wish can don't accomplish when we want it. For that we can define to restrict it. This restrict checked the data's truth. Those restricts what we defined are consistent Database's data truth.

1.1.1.3 Prevention of the Same Time Access Inconsistence

In Database applications, Database objects can share many different applications. Datas can share at the same time different applications and however different users. For that situations and conditions Database Management System (DBMS) is automatically solve which that the together using matter.

For Example: A Product Stock have 100 unit rulmans and two different users enters 50 unit rulmans and 55 unit rulmans at the same time. Operation is entered at the same time and we can think 100 unit product stocks exit 105 unit at the same time but Database Management System don't give permission to exit those twice users enter. Exists are at the same time but DBMS firstly give permission first exit and then for second exit make a control for preventive work, this is the period integrity whole in DBMS.

1.1.1.4 Data Security

To provide some applications produce datas security is very important situation. To access Database's important information by all Database users is not the wanted situation.

For Example: From User in work Marketing Department Application access to another personnel informations must be protected. Like this, every users access data defines separately. Database Systems present restrictions about access have many developed possibilities. In Database many authorized users can access many applications and those authorities are hide on Database together with datas.

Databases often contain sensitive information. Different databases provide security schemes for protecting that information. Some databases, such as Paradox and dBase, only provide security at the table or field level. When users try to access protected tables, they are required to provide a password. Once users have been authenticated, they can see only those fields (columns) for which they have permission.

1.1.2 Data Models

Database Management Systems have relationship with definite Data Models. One Database structures foundation form by Data Model concept. Order the data logical level for using concepts, structures and operation commonuties named Data Model.

Many Data Models developed at this time around. Those Data Models can grouped four principles.

1 Hierarchical Data Model
2 Network Data Model
3 Relational Data Model
4 Object Oriented Data Model

Hierarchical Data Model and Network Data Model are not using at the moment. Most widespread using Data Model is Relational Data Model. The relational database model is very popular, especially in the personal computer environment.

E. F. Codd developed the relational database model in 1970. The model is based on mathematical set theory, and it uses a relation as the building block of the database. The relation is represented by two dimensional flat structure known as a table. The user does not have to know the mathematical details or the physical aspects of the data, but the user views data in a logical two dimensional structure. A database system that manages

a relational database environment is known as Relational Database Management System (RDMS).

The table is a matrix of rows and columns in which each row represents an entity and each columns represents an attribute. In other word, a table represent an entity set as per the database theory and it represents a relational as per the relational database theory. In daily practice, the terms table, relation, and entity set are used interchangeably. Now Object Oriented Data Model and Relational Data Model together using some Database Management Systems (DBMS).

1.1.3 Database Objects

Database to be formed different structured objects. With help those objects make all Database operations and data's orients procedures. Databases contains objects for many different aims. Those objects most importants are listed in Figure 1.1.3.1

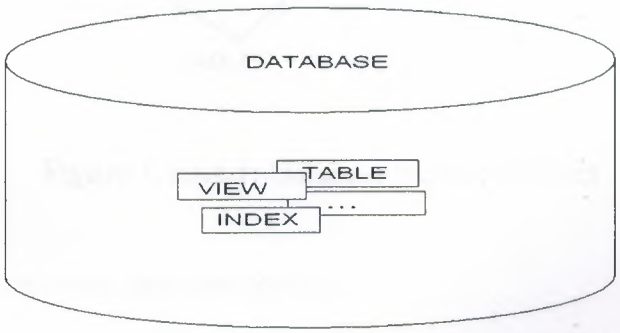

Figure 1.1.3.1: Database Included Objects

Databases created by objects.

1. Tables
2. Views
3. Indexes

1.1.3.1 Tables

Tables are basic structures for Databases. Table structures most important characteristics are in below.

1. Tables can created during at any moment or even when the Database is using by user.
2. We don't need to determine measure for tables. But also the advantage known table size help us (Figure 1.1.3.1.1).

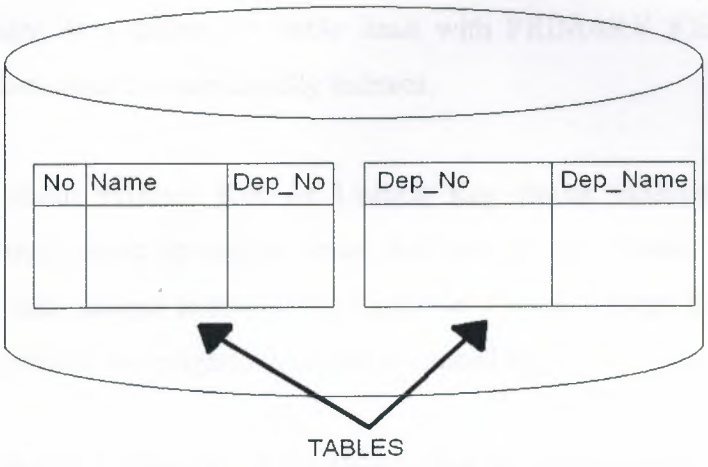

Figure 1.1.3.1.1: Database Included Tables

Tables are the most important database objects.

1.1.3.2 Views

Views help create one or more than one tables logical sub-heaps. View's evaluate is table supported logical table. View is not hide data physically but table is hide data physically. Views are evaluated (hiding) SELECT expressions. A SELECT expression again and again or repeatedly using is necessary, we can define the SELECT expression like view and than view work is possible.

Views are preferring with those reasons below.

1. Views are possibilities of restricted database access. Because view, only shows selected parts of table.
2. Make easy for complicated interrogations.
3. Many same using data defines with many views.

1.1.3.3 Indexes

Indexes provided in a table's lines, with constant column more speedy arrival database objects. User's can created indexes with expressions help and also it can be create automatically. If a define of table limit with PRIMARY KEY or UNIQUE, those criterions can created automatically indexes.

Creatings about Primary Key or Unique Key limits indexes are with those keys characteristics named by unique index but we can also create another criterion work which not only unique indexes. For Example: Create a index for area of Foreign Key brings concerning interrogations or queries speed high.

If defines indexes, quantity of read/write disc to become less, thus arrival to data is more sepedly and effective. Indexes are forming indepent to table. If index for table undefined, read operations scans all table. There are five types of key using see those in below;

1. **Primary Key**.· A single attribute used as a unique identifier.
2. **Composite Key**.· Two or more attributes used as a unique identifier.
3. **Secondary Key**.· A non-key attribute used in the search operation.
4. **Foreign Key**.· An attribute that references primary key of a table.

An added attribute used as a primary key.

1.1.4 Overview of Delphi's Database Features and Capabilities

A Delphi database application is built using Delphi database development tools, Delphi data-access components, and data-aware GUI components. A database application uses Delphi components to communicate with the Borland Database Engine (BDE), which in turn communicates with databases. The following figure 1.4 illustrates the relationship of Delphi tools and Delphi database applications to the BDE and data sources:

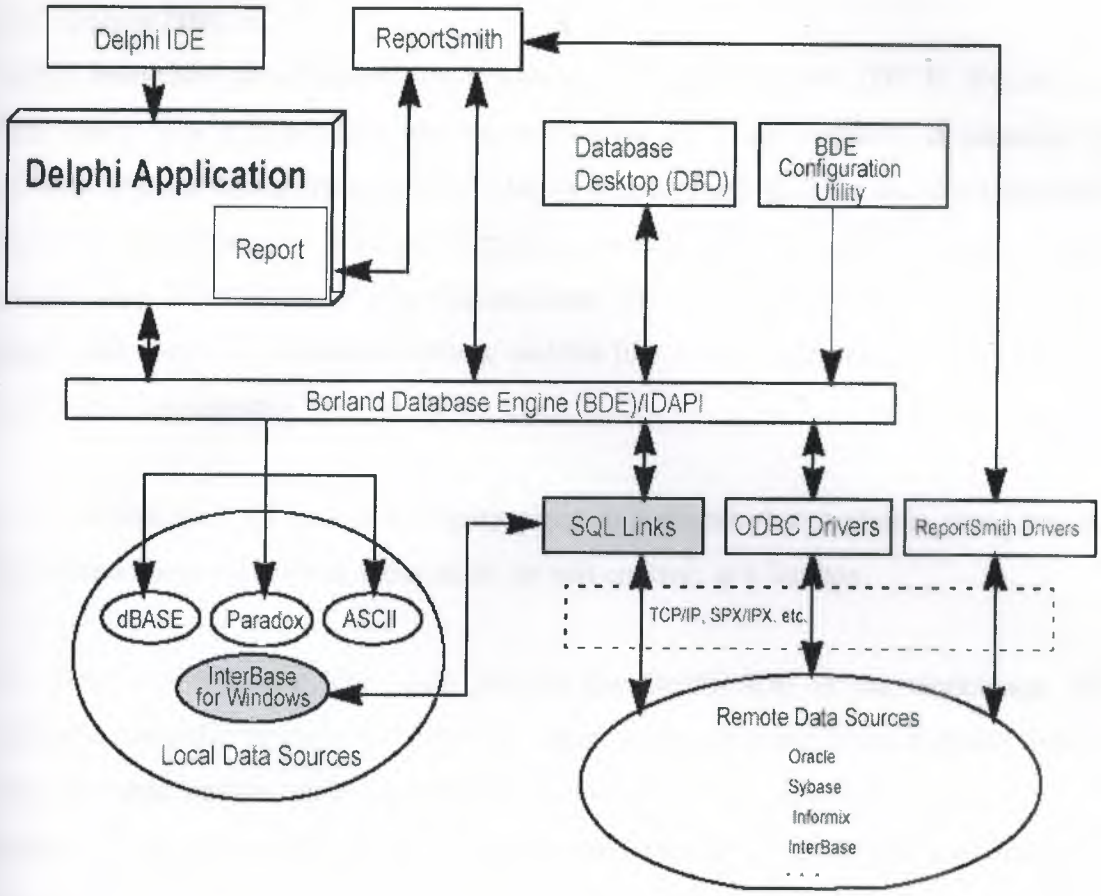

Figure 1.1.4.1: Delphi Database Architecture

1.2 Delphi Structures

Delphi is Borland's best-selling rapid application development (RAD) product for writing Windows applications. With Delphi, we can write Windows programs more quickly and more easily than was ever possible before. We can create Win32 console applications or Win32 graphical user interface (GUI) programs. When creating Win32 GUI applications with Delphi, we have all the power of a true compiled programming language (Object Pascal) wrapped up in a RAD environment. What this means is that we can create the user interface to a program (the user interface means the menus, dialog boxes, main window, and so on) using drag-and-drop techniques for true rapid application development. We can also drop ActiveX controls on forms to create specialized programs such as Web browsers in a matter of minutes.

1.2.1 Delphi IDE

Delphi integrated development environment (IDE). The Delphi IDE is divided into three parts. The top window can be considered the main window. It contains the toolbars and the Component palette. The Delphi toolbars give us one-click access to tasks such as opening, saving, and compiling projects. The Component palette contains a wide array of components that you can drop onto your forms. (Components are text labels, edit controls, list boxes, buttons, and the like.) For convenience, the components are divided into groups.

A component is a self-contained binary piece of software that performs some specific predefined function, such as a text label, an edit control, or a list box.

The Delphi Workspace, The main part of the Delphi IDE is the workspace. The workspace initially displays the Form Designer. It should come as no surprise that the Form Designer enables us to create forms. In Delphi, a form represents a window in our program. The form might be the program's main window, a dialog box, or any other type of window. You use the Form Designer to place, move, and size components as part of the form creation process. Hiding behind the Form Designer is the Code Editor. The Code Editor is where you type code when writing your programs.

The Object Inspector, Form Designer, Code Editor, and Component palette work interactively as you build applications. Now that you've had a look at what makes up the Delphi IDE, let's actually do something.

1.2.1.1 The Object Inspector

Below the main window and on the left side of the screen is the Object Inspector. It is through the Object Inspector that you modify a component's properties and events. You will use the Object Inspector constantly as you work with Delphi. The Object Inspector has two tabs: the Properties tab and the Events tab. A component's properties control how the component operates. For example, changing the Color property of a component changes the background color of that component. The list of properties available varies from component to component, although components usually have several common elements.

The Events tab contains a list of events for a component. Events occur as the user interacts with a component.

An event is something that occurs as a result of a component's interaction with the user or with Windows.

An event handler is a section of code that is invoked in your application in response to an event.

See the Figure 1.2.1.1.1: A connected component expanded in the Object Inspector while working on another component (a Data Source).

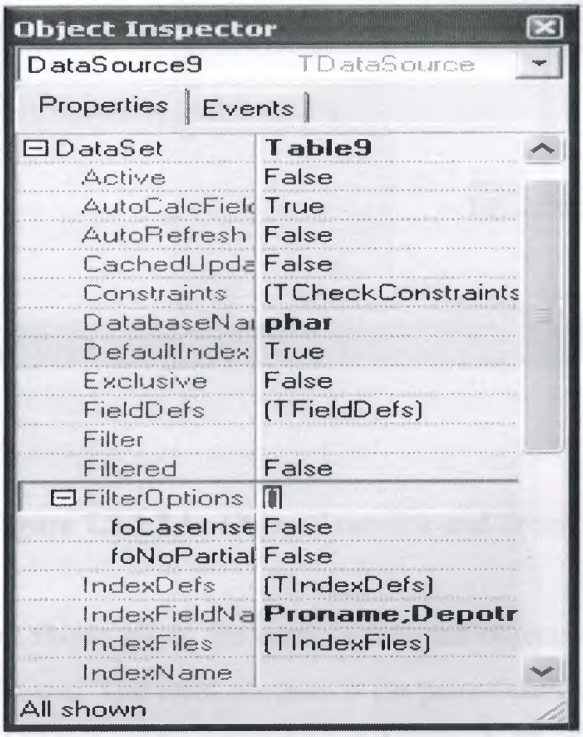

Figure 1.2.1.1.1: Object Inspector

1.2.1.2 The Object TreeView

Figure 1.2.1.2.1 show us the Delphi IDE : Notice the Object TreeView and the TreeView

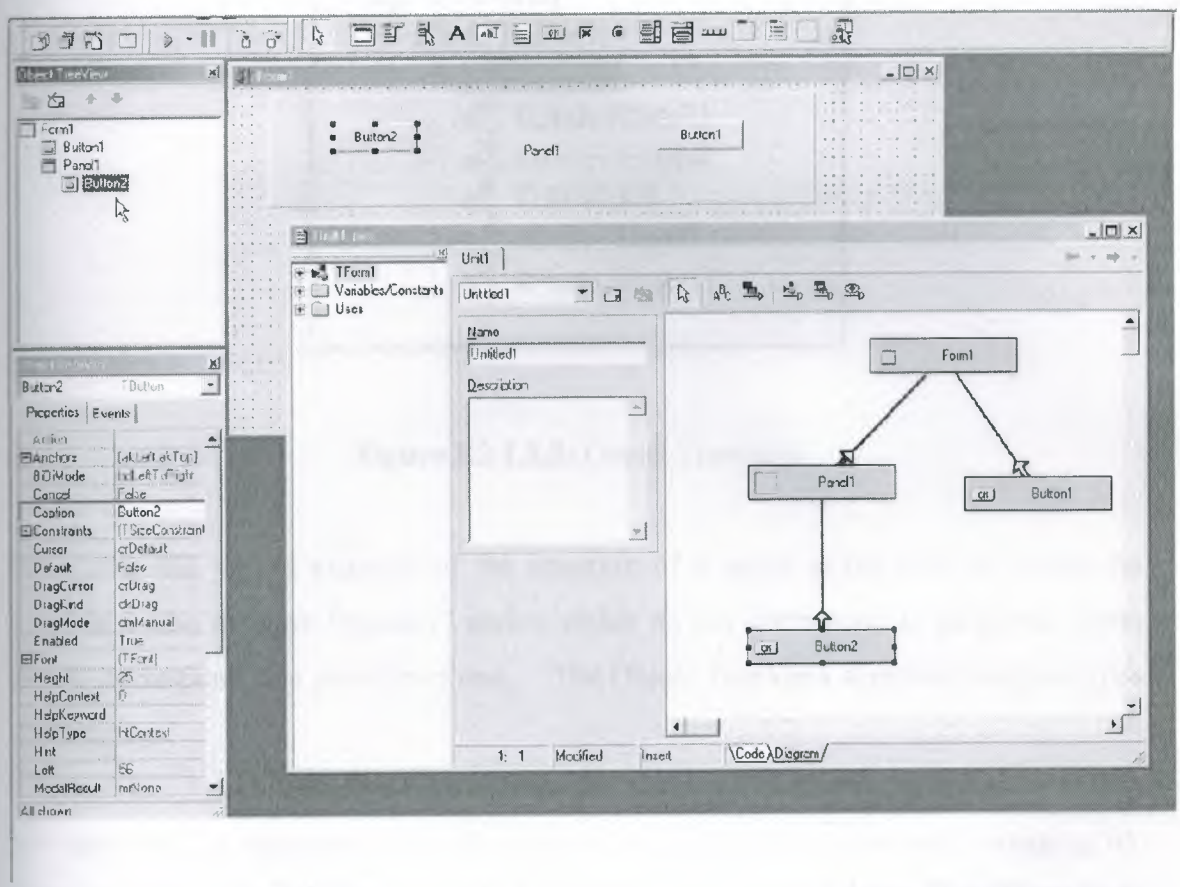

Figure 1.2.1.2.1: Object Treeview and Treeview

The Object TreeView shows all of the components and objects on the form in a tree, representing their relations. The most obvious is the parent/child relation: Place a panel on a form, a button inside it and one outside of the panel. The tree will show the two buttons, one under the form and the other under the panel, as in Figure 2.1. Notice that the TreeView is synchronized with the Object Inspector and Form Designer, so as we select an item and change the focus in any one of these three tools, the focus changes in the other two tools. Besides parent/child, the Object TreeView shows also other relations, such as owner/owned, component/sub-object, collection/item, plus various specific ones, including dataset/connection and data source/dataset relations.

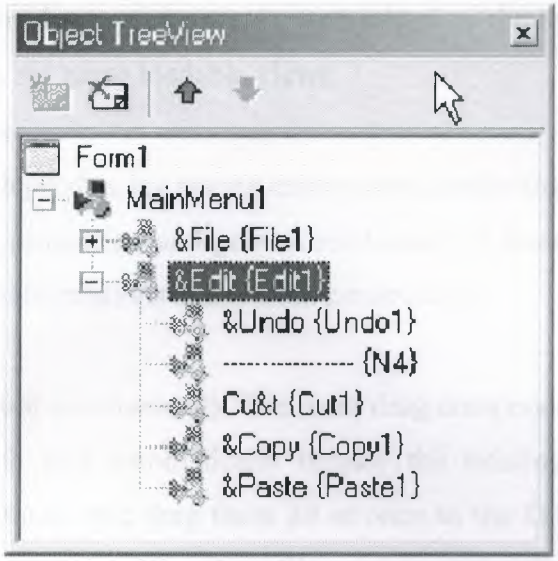

Figure 1.2.1.2.2: Object Treeview

Here, you can see an example of the structure of a menu in the tree. At times, the TreeView also displays "dummy" nodes, which do not correspond to an actual object but do correspond to a predefined one. The Object TreeView supports multiple types of dragging:

. We can select a component from the palette (by clicking it, not actually dragging it), move the mouse over the tree, and click a component to drop it there. This allows us to drop a component in the proper container (form, panel, and others) regardless of the fact that its surface might be totally covered by other components, something that prevents us from dropping the component in the designer without first rearranging those components. Moving instead of cutting provides the advantage that if we have connections among components, these are not lost, as happens when we delete the component during the cut operation. We can drag components from the TreeView to the Diagram view, as we'll see later. Right-clicking any element of the TreeView displays a shortcut menu similar to the component menu we get when the component is in a form. We can even delete items from the tree. The TreeView doubles also as a collection editor.

1. Loadable Views

   Another important change has taken place in the Code Editor window. For any single file loaded in the IDE, the editor can now show multiple views, and these

views can be defined programmatically and added to the system, then loaded for given files—hence the name loadable views.

2. The Diagram View

   This view shows dependencies among components, including parent/child relations, ownership, linked properties, and generic relations. For dataset components, it also supports master/detail relations and lookup connections.

The Diagram is not built automatically. We must drag components from the TreeView to the diagram, which will automatically display the existing relations among the components we drop there, and drag them all at once to the Diagram page. When we release the mouse button, the Diagram will set up a property relation based on the FocusControl property, which is the only property of the label referring to an edit control.

1.2.1.3 Updated Environment Options Dialog Box

Environment Options dialog box have been rearranged, moving the Form Designer options from the Preferences page to the Designer page. There are also the options and pages:

1. The Preferences page of the Environment Options dialog box has a check box that prevents Delphi windows from automatically docking with each other.
2. A new page, Environment Variables, allows us to see system environment (such as the standard path names and OS settings) and set user-defined variables. The nice and clever point is that we can use both system and user-defined environment variables in each of the dialog boxes of the IDE.
3. Another page is called Internet. In this page, we can choose the default file extensions used for HTML and XML files (mainly by the WebSnap framework) and also associate an external editor with each extension.

1.2.1.4 The Form Designer

Another Delphi window we'll interact with very often is the Form Designer, a visual tool for placing components on forms. In the Form Designer, we can select a component directly with the mouse or through the Object Inspector, a handy feature when a control is behind another one or is very small. If one control covers another completely, we can use the Esc key to select the parent control of the current one. We can press Esc one or

more times to select the form, or press and hold Shift while you click the selected component. This will deselect the current component and select the form by default.

1.2.1.5 Compiling and Building Projects

There are several ways to compile a project. If we run it (by pressing F9 or clicking the Run toolbar icon), Delphi will compile it first. When Delphi compiles a project, it compiles only the files that have changed. If you select Compile . Build All instead, every file is compiled. We should only need this second command infrequently, since Delphi can usually determine which files have changed and compile them as required. The only exception is when we change some project options, in which case we have to use the Build All command to put the new options into effect. To build a project, Delphi first compiles each source code file, generating a Delphi compiled unit (DCU). (This step is performed only if the DCU file is not already up-to-date.) The second step, performed by the linker, is to merge all the DCU files into the executable file, optionally with compiled code from the VCL library. The third step is binding into the executable file any optional resource files, such as the RES file of the project, which hosts its main icon, and the DFM files of the forms. You can better understand the compilation steps and follow what happens during this operation if we enable the Show Compiler Progress option .

1.2.2 Objects and Classes

Most modern programming languages support object-oriented programming (OOP). OOP languages are based on three fundamental concepts: encapsulation (usually implemented with classes), inheritance, and polymorphism (or late binding).

Introducing Classes and Objects, The cornerstone of the OOP extensions available in Object Pascal is represented by the class keyword, which is used inside type declarations. Classes define the blueprint of the objects you create in Delphi. As the terms class and object are commonly used and often misused, let's be sure we agree on their definitions. A class is a user-defined data type, which has a state (its representation) and some operations (its behavior). A class has some internal data and some methods, in the form of procedures or functions, and usually describes the generic characteristics and behavior of some similar objects.

An object is an instance of a class, or a variable of the data type defined by the class. Objects are actual entities. When the program runs, objects take up some memory for their internal representation. The relationship between object and class is the same as the one between variable and type.

To declare a new class data type in Object Pascal, with some local data fields and some methods, use the following syntax:

```
type
TDate = class
Month, Day, Year: Integer;
procedure SetValue (m, d, y: Integer);
function LeapYear: Boolean;
end;
```

The following is a complete class definition, with two methods declared and not yet fully defined. The definition of these two methods (the LeapYear function and the SetValue procedure) must be present in the same unit of the class declaration and are written with this syntax:

```
procedure TDate.SetValue (m, d, y: Integer);
begin
Month := m;
Day := d;
Year := y;
end;
function TDate.LeapYear: Boolean;
begin
// call IsLeapYear in SysUtils.pas
Result := IsLeapYear (Year);
end;
```

The method names are prefixed with the class name (using the dot-notation), because a unit can hold multiple classes, possibly with methods having the same names. We can actually avoid retyping the method names and parameter list by using the class completion feature of the editor. Simply type or modify the class definition and press

Ctrl+Shift+C while the cursor is within the class definition itself; this will allow Delphi to generate a skeleton of the definition of the methods, including the begin and end statements.Once the class has been defined, we can create an object and use it as ;

```
var
ADay: TDate;
begin
// create an object
ADay := TDate.Create;
// use the object
ADay.SetValue (1, 1, 2000);
if ADay.LeapYear then
ShowMessage ( 'Leap year: ' + IntToStr (ADay.Year));
// destroy the object
ADay.Free;
end;
```

Notice that ADay.LeapYear is an expression similar to ADay.Year, although the first is a function call and the second a direct data access. We can optionally add parentheses after the call of a function with no parameters. We can find the code snippets above in the source code of the Date1 example; the only difference is that the program creates a date based on the year provided in an edit box.

1.2.2.1 The Self Keyword

Methods are very similar to procedures and functions. The real difference is that methods have an implicit parameter, which is a reference to the current object. Within a method you can refer to this parameter the current object using the Self keyword. This extra hidden parameter is needed when we create several objects of the same class, so that each time we apply a method to one of the objects, the method will operate only on its own data and not affect sibling objects. For example, in the SetValue method of the TDate class, listed earlier, we simply use Month, Year, and Day to refer to the fields of the current object, something you might express as;

```
Self.Month := m;
Self.Day := d;
```

This is actually how the Delphi compiler translates the code, not how we are supposed to write it. The Self keyword is a fundamental language construct used by the compiler, but at times it is used by programmers to resolve name conflicts and to make tricky code more readable.

All we really need to know about Self is that the technical implementation of a call to a method differs from that of a call to a generic subroutine. Methods have an extra hidden parameter, Self. Because all this happens behind the scenes.

If we look at the definition of the TMethod data type in the System unit, we'll see that it is a record with a Code field and a Data field. The first is a pointer to the function's address in memory; the second the value of the Self parameter to use when calling that function address.

1.2.2.2 Overloaded Methods

Object Pascal supports overloaded functions and methods: we can have multiple methods with the same name, provided that the parameters are different. By checking the parameters, the compiler can determine which of the versions of the routine you want to call. There are two basic rules:

1. Each version of the method must be followed by the overload keyword.
2. The differences must be in the number or type of the parameters or both. The return type cannot be used to distinguish between two mwthods.

Overloading can be applied to global functions and procedures and to methods of a class. As an example of overloading, I've added to the TDate class two different versions of the SetValue method:

```
type
TDate = class
public
procedure SetValue (y, m, d: Integer); overload;
procedure SetValue (NewDate: TDateTime); overload;
...//the rest of the class declaration
procedure TDate.SetValue (y, m, d: Integer);
```

```
begin
fDate := EncodeDate (y, m, d);
end;
procedure TDate.SetValue(NewDate: TDateTime);
begin
fDate := NewDate;
end;
```

1.2.2.3 Creating Components Dynamically

In Delphi, the Self keyword is often used when we need to refer to the current form explicitly in one of its methods. The typical example is the creation of a component at run time, where we must pass the owner of the component to its Create constructor and assign the same value to its Parent property. The following program has a simple form with no components and a handler for its OnMouseDown event. We've used OnMouseDown because it receives as its parameter the position of the mouse click (unlike the OnClick event). We need this information to create a button component in that position. Here is the code of the method:

It is very common to write code like the below method using a with statement.

```
procedure TForm1.FormMouseDown (Sender: TObject;
Button: TMouseButton; Shift: TShiftState; X, Y: Integer);
begin
with TButton.Create (Self) do
begin
Parent := Self;
Left := X;
Top := Y;
Width := Width + 50;
Caption := Format ( 'Button in %d, %d', [X, Y]);
end;
end;
```

When writing a procedure like the code you've just seen, we might be tempted to use the Form1 variable instead of Self. In this specific example, that change wouldn't make any practical difference, but if there are multiple instances of a form, using Form1 would be an error. In fact, if the Form1 variable refers to the first form of that type

being created, by clicking in another form of the same type, the new button will always be displayed in the first form. Its Owner and Parent will be Form1 and not the form the user has clicked. In general, referring to a particular instance of a class when the current object is required is bad OOP practice.

1.2.2.4 Class Methods and Class Data

When we define a field in a class, we actually specify that the field should be added to each object of that class. Each instance has its own independent representation (referred to by the Self pointer). In some cases, however, it might be useful to have a field that is shared by all the objects of a class. Other object-oriented programming languages have formal constructs to express this, while in Object Pascal we can simulate this feature using the encapsulation provided at the unit level. We can simply add a variable in the implementation portion of a unit, to obtain a class variable—a single memory location shared by all of the objects of a class. If we need to access this value from outside the unit, we might use a method of the class. However, this forces us to apply this method to one of the instances of the class. An alternative solution is to declare a class method. A class method cannot access the data of any single object but can be applied to a class as a whole rather than to a particular instance. To declare a class method in Object Pascal, we simply add the class keyword in front of it:

```
type
MyClass = class
class function ClassMeanValue: Integer;
```

The use of class methods is not very common in Object Pascal, because we can obtain the same effect by adding a procedure or function to a unit declaring a class. Object-oriented purists, however, will definitely prefer the use of a class method over a routine unrelated to a class. For example, an OOP purist would add a class method for getting the current date to a TDate class instead of using a global function .

1.2.2.5 Private, Protected, and Public

For class-based encapsulation, the Object Pascal language has three access specifiers: private, protected, and public.

Here are the three classic access specifiers:

1 The private directive denotes fields and methods of a class that are not accessible outside the unit (the source code file) that declares the class.
2 Protected dirctive is used to indicate methods and fields with limited visibility. Only the current class and its sub-classes can access protected class .
3 The public directive denotes fields and methods that are freely accessible from any other portion of a program as well as in the unit in which they are defined.

Generally, the fields of a class should be private; the methods are usually public. However, this is not always the case. Methods can be private or protected if they are needed only internally to perform some partial computation. Fields can be protected so that we can manipulate them in subclasses, but only if we are fairly sure that their type definition is not going to change. This means that if two classes are in the same unit, there is no protection for their private fields. Only by placing a class in the interface portion of a unit will you limit the visibility from classes and functions in other units to the public method and fields of the class.

```
type
TDate = class
private
Month, Day, Year: Integer;
public
procedure SetValue (y, m, d: Integer); overload;
procedure SetValue (NewDate: TDateTime); overload;
function LeapYear: Boolean;
function GetText: string;
procedure Increase;
end;
```

In this version, the fields are now declared to be private, and there are some new methods. The first, GetText, is a function that returns a string with the date. You might think of adding other functions, such as GetDay, GetMonth, and GetYear, which simply return the corresponding private data, but similar direct data-access functions are not always needed. Providing access functions for each and every field might reduce the

encapsulation and make it harder to modify the internal implementation of a class. Access functions should be provided only if they are part of the logical interface of the class we are implementing.

Notice that because the only change is in the private portion of the class, we won't have to modify any of our existing programs that use it. This is the advantage of encapsulation!

1.3 Delphi and Database Relation

The original Delphi characteristics are high speed collector, the approach about form based and object oriented, harmonious with Windows programming and component technology. However, the very important element is all others basic Object Pascal language.

The basic characteristic of this programming environment is Delphi's support to database applications.

Let's we can talk about Borland Database Engine (BDE). BDE is coming with Paradox, at the time when Delphi don't exist and BDE developed by Borland to supported many SQL service units and Borland's own local databases.

Using a associated database engine's advantage is applications can be move or transport between same category's different service units.

Using BDE's specific advantages are this technology whole with Delphi, elements are fully and very good documented and the only logical analyze way to arrive local files like Paradox and dBase tables.

Now we are coming this analyze's disadvantage: Borland's BDE develop is coming to end and with another words, now it can not promotion with Delphi.

BDE is with it's advantages and disadvantages still a good analyze, but in long periods BDE's trusty is absolutely suspicious.

1.3.1 Tables and Queries

The simplest traditional way to specify data access in Delphi was to use the BDE Table component. A Table object simply refers to a database table. When we use a Table component, we need to indicate the name of the database you want to use in its DatabaseName property. You can enter an alias or the path of the directory with the table files. The Object Inspector lists the available names, which depend on the aliases installed in the BDE. We also need to indicate a proper value in the TableName property. The Object Inspector lists the available tables of the current database (or directory), so you should generally select the DatabaseName property first.

Another classic dataset is the BDE Query component. A query requires a SQL language command. We can customize a query using SQL more easily than you can customize a table (as long as you know at least the basic elements of SQL, of course). The Query component has a DatabaseName property like the Table component, but it does not have a TableName property. The table is indicated in the SQL statement, stored in the SQL property.

For example, We can write a simple SQL statement like this:

```
select * from Product
```

Where Product is the name of a table and the asterisk (*) indicates that we want to use all of the fields in the table. The efficiency of a table or a query varies depending on the database we are using. In general, we can say that the Table component tends to be faster on local tables, while the Query component tends to be faster on SQL servers, although this is just a very general rule, and in many cases you might have the opposite effect.

The third BDE dataset component is StoredProc, which refers to stored procedures of a SQL server database. You can run these procedures and get the results in the form of a database table. Stored procedures can only be used with SQL servers.

1.3.2 Specific Table Features

The BDE Table component has specific features not shared by all datasets. For example, it has filters, ranges, and specific techniques for locating records. A filter, set in the Filter property and activated by toggling the Filtered property, is available in each dataset, although its role changes depending on the underlying implementation. A range, instead, is specific to a Table and allows you to specify the two extreme values and consider only the record falling within that interval. When using a Table, and particularly a local one, there are specific methods we can use to find a record, such as GotoKey, FindKey, GotoNearest, FindNearest, and Locate. The Locate method is shared by all datasets, and I'll discuss it later along with other general features of the TDataSet class. The other methods are specific of the TTable class and work in conjunction with the index set in the ndexFieldNames property of the component. The simplest approach is to use the FindNearest method for the approximate search and the FindKey method to look for an exact match:

```
// goto
Table1.FindNearest ([EditName.Text]);
// go near
if not Table1.FindKey ([EditName.Text]) then
MessageDlg ('Product not found', mtError, [mbOk], 0);
Classic BDE Components
```

Both find methods use as parameters an array of constants. Each array element corresponds to one of the fields of the current index. We can also pass only the value for the initial field or fields of the index, so the following fields will not be considered.

1.3.3 A Query with Parameters

When we need slightly different versions of the same SQL query, instead of modifying the text of the Query (stored in the SQL property) each time, we can write a query with a parameter and simply change the value of the parameter. For example, if we decide to have a user choose the countries of a continent (using the Product table of the PHAR database), we can write the following parametric query:

```
select *
from Product
where Barcode = :Barcode
```

In this SQL clause, :Barcode is a parameter. We can set its data type and startup value, using the editor of the Params property collection of the Query component. When the form displayed by this program, called Productinfo and uses a list box to provide all the available values for the parameters. Instead of preparing the items of the list box at design time, we can extract the available continents from the same Editing the collection of parameters of a Query component database table as the program starts. This is accomplished using a second query component, with this SQL statement:

```
select distinct Progroup
from Product
```

After activating this query, the program scans its result set, extracting all the values and adding them to the list box:

```
procedure TProductinfo.FormCreate(Sender: TObject);
begin
// get the list of continents
Query2.Open;
while not Query2.EOF do
begin
ListBox1.Items.Add (Query2.Fields [0].AsString);
Query2.Next;
end;
ListBox1.ItemIndex := 0;
// open the first query
Query1.Params[0].Value := ListBox1.Items [0];
Query1.Open;
end;
```

Before opening the query, the program selects as its parameter the first item of the list box, which is also activated by setting the ItemIndex property to 0. When the list box is selected, the program closes the query and changes the parameter:

```
procedure TQueryForm.ListBox1Click(Sender: TObject);
begin
Query1.Close;
```

```
Query1.Params[0].Value := ListBox1.Items [Listbox1.ItemIndex];
Query1.Open;
end;
```

The final refinement is that when the user enters a record with a new product, it is added automatically to the list box. Instead of refreshing the entire list, with the same code executed in the FormCreate method, we can do this by handling the BeforePost event and adding the continent to the list if it is not already there:

```
procedure TProductinfo.Query1BeforePost(DataSet: TDataSet);
var
StrNewCont: string;
begin
// add the continent, if not already in the list
StrNewCont := Query1.FieldByName ('Continent').AsString;
if ListBox1.Items.IndexOf (StrNewCont) < 0 then
ListBox1.Items.Add (StrNewCont);
end;
```

We can add a little extra code to this program to take advantage of a specific feature of parameterized queries. To react faster to a change in the parameters, these queries can be optimized, or prepared. Simply call the Prepare method before the program first opens the query (after setting the Active property of the Query component to False at design time) and call Unprepare once the query won't be used anymore:

```
procedure TProductinfo.FormCreate(Sender: TObject);
begin
...
// prepare and open the first query
Query1.Prepare;
Query1.Params[0].Value := ListBox1.Items [0];
Query1.Open;
end;
procedure TProductinfo.FormDestroy(Sender: TObject);
begin
Query1.Close;
```

```
Query1.Unprepare;
end;
```

Prepared parameterized queries are very important when we work on a complex query. In fact, the BDE or the SQL server must read the text of the query and determine how to process it. If we use the same query (even if a parametric one) over and over, the engine doesn't need to reprocess the query but already knows how to handle it.

Chapter II: Delphi & Database Applications

2.1 Delphi Applications Structure

2.1.1 VCL versus VisualCLX

Delphi introduces the CLX library alongside the traditional VCL library. There are certainly many differences, even in the use of the RTL and code library classes, between developing programs specifically for Windows or with a crossplatform attitude, but the user interface portion is where differences are most striking. The visual portion of VCL is a wrapper of the Window API. It includes wrappers of the native Windows controls (like buttons and edit boxes), of the common controls (like tree views and list views), plus a bunch of native Delphi controls bound to the Windows concept of a window. There is also a TCanvas class that wraps the basic graphic calls, so you can easily paint on the surface of a window.

The following figure 2.1.1.1 shows the relationship of selected classes that make up the VCL hierarchy. The CLX hierarchy is similar to the VCL hierarchy but Windows controls are called widgets (therefore TWinControl is called TWidgetControl, for example), and there are other differences.

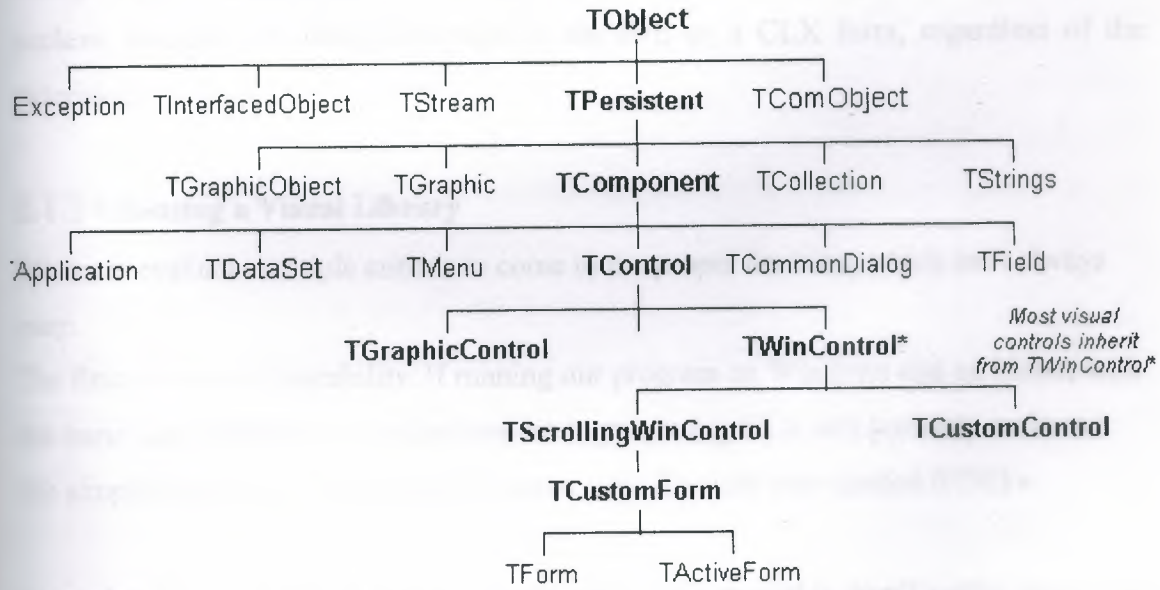

Figure 2.1.1.1: Twidget Control for Cross-Platform Applications.

2.1.2 **DFM and XFM**

As we create a form at design time, this is saved to a form definition file. Traditional VCL applications use the DFM extension, which stands for Delphi form module. CLX applications use the XFM extension, which stands for cross-platform (i.e., *X*) form modules. The actual format of DFM or XFM files, which can be based on a textual or binary representation, is identical. A form module is the result of streaming the form and its components, and the two libraries share the streaming code, so they produce a fairly similar effect. So the reason for having two different extensions doesn't lie in internal compiler tricks or incompatible formats. It is merely an indication to programmers and to the IDE of the type of components you should expect to find within that definition (as this indication is *not* included in the file itself).

If we need to convert a DFM file into an XFM file, we can simply rename the file. However, expect to find some differences in the properties, events, and available components, so that reopening the form definition for a different library will probably cause quite a few warnings.

Apparently Delphi's IDE chooses the active library only by looking at the extension of the form module, ignoring the references in the uses statements. For this reason, do change the extension if you plan using CLX. On Kylix, a different extension is pretty useless, because any form is opened in the IDE as a CLX form, regardless of the extension.

2.1.3 **Choosing a Visual Library**

We must evaluate multiple criteria to come to the proper decision, which isn't always easy.

The first criterion is portability. If running our program on Windows and on Linux, with the same user interface, is a major concern to you, using CLX will probably make our life simpler and let you keep a single source code file with very limited IFDEFs.

For a simple user interface (edits, buttons, grids), this probably won't matter much, but if we havemany tree view and list view controls, the differences will be quite clear. On

the other hand,with CLX you'll be able to let your users select a look-and-feel of their choice, different from the basic Windows look, and use it consistently across platforms.

Using native controls implies also that as soon as we get a new version of the Windowsoperating system, our application will (probably) adapt to it. This is good for the user, but might cause us a lot of headaches in case of incompatibilities. Differences in the Microsoft common controls library over the last few years have been a major source of frustration for Windows programmers in general, including Delphi programmers. Another criterion is the deployment: If we use CLX, we'll have to ship our Windows program with the Qt libraries, which are not commonly available on Windows systems.

2.1.4 Conditional Compilation for Libraries

If we want to keep a single source code file but compile with VCL on Windows and CXL on Linux, we can use platform-specific symbols (such as $IFDEF LINUX) to distinguish the two situations in case of conditional compilation. But what if we want to be able to compile a portion of code for both libraries on Windows?
We can either define a symbol of your own, and use conditional compilation, or (at times) test for the presence of identifiers that exist only in VCL or CLX .:

2.1.4.1 TControl and Derived Classes

One of the most important subclasses of TComponent is TControl, which corresponds to visual components. This base class is available both in CLX and VCL and defines general concepts, such as the position and the size of the control, the parent control hosting it, and more. For an actual implementation, though, you have to refer to its two subclasses. In VCL these are TWinControl and TGraphicControl; in CLX they are TWidgetControl and TGraphicControl. Here are their key features: Window-based controls (also called windowed controls) are visual components based on an operating-system window. A TWinControl in VCL has a window handle, a number referring to an internal Windows structure. A TWidgetControl in CLX has a Qt handle, TControl and Derived Classes

2.1.5 Delphi Application Object

Structure of Delphi applications, it is time to delve into some more details of this global object and its corresponding class. Application is a global object of the TApplication class, defined in the Forms unit and created in the Controls unit.

The TApplication class is a component, but we cannot use it at design time. Some of its properties can be directly set in the Application page of the Project Options dialog box; others must be assigned in code.

To handle its events, instead, Delphi includes a handy ApplicationEvents component. Besides allowing us to assign handlers at design time, the advantage of this component is that it allows for multiple handlers. If we simply place two instances of the ApplicationEvents component in two different forms, each of them can handle the same event, and both event handlers will be executed. In other words, multiple ApplicationEvents components can chain the handlers. Some of these application-wide events, including OnActivate, OnDeactivate, OnMinimize, and OnRestore, allow us to keep track of the status of the application. Other events are forwarded to the application by the controls receiving them, as in OnActionExecute, OnAction- Update, OnHelp, OnHint, OnShortCut, and OnShowHint. Finally, there is the OnException global exceptions. The OnIdle event used for background computing, and the OnMessage event, which fires whenever a message is posted to any of the windows or windowed controls of the application.

Although its class inherits directly from TComponent, the Application object has a window associated with it. The application window is hidden from sight but appears on the Taskbar. This is why Delphi names the window Form1 and the corresponding Taskbar icon Project1.

The window related to the Application object the application window serves to keep together all the windows of an application. The fact that all the top-level forms of a program have this invisible owner window, for example, is fundamental when the application is activated. In fact, when the windows of our program are behind those of other programs, clicking one window in our application will bring all of that application's windows to the front. In other words, the unseen application window is

used to connect the various forms of the application. Actually the application window is not hidden, because that would affect its behavior; it simply has zero height and width, and therefore it is not visible.

When we create a new, blank application, Delphi generates a code for the project file, which includes the following:

```
begin
Application.Initialize;
Application.CreateForm(TForm1, Form1);
Application.Run;
end.
```

As we can see in this standard code, the Application object can create forms, setting the first one as the MainForm (one of the Application properties) and closing the entire application when this main form is destroyed. Moreover, it contains the Windows message loop (started by the Run method) that delivers the system messages to the proper windows of the application. A message loop is required by any Windows application, but we don't need to write one in Delphi because the Application object provides a default one.

2.1.6 Displaying the Application Window

There is no better proof that a window indeed exists for the Application object than to display it. Actually, we don't need to show it—we just need to resize it and set a couple of window attributes, such as the presence of a caption and a border. We can perform these operations by using Windows API functions on the window indicated by the Handle property of the Application object:

```
procedure TForm1.Button1Click(Sender: TObject);
var
OldStyle: Integer;
begin
// add border and caption to the app window
OldStyle := GetWindowLong (Application.Handle, gwl_Style);
SetWindowLong (Application.Handle, gwl_Style,
OldStyle or ws_ThickFrame or ws_Caption);
```

```
// set the size of the app window
SetWindowPos (Application.Handle, 0, 0, 0, 200, 100,
swp_NoMove or swp_NoZOrder);
end;
```

The two GetWindowLong and SetWindowLong API functions are used to access the system information related to the window. In this case, we are using the gwl_Style parameter to read or write the styles of the window, which include its border, title, system menu, border icons, and so on. The code above gets the current styles and adds (using an or statement) a standard border and a caption to the form. The application window is not a form. Executing this code displays the project window, as we can see in Figure 2.1.6.1.

Although there's no need to implement something like this in our own programs, running this program will reveal the relation between the application window and the main window of a Delphi program. This is a very important starting point if you want to understand the internal structure of Delphi applications.

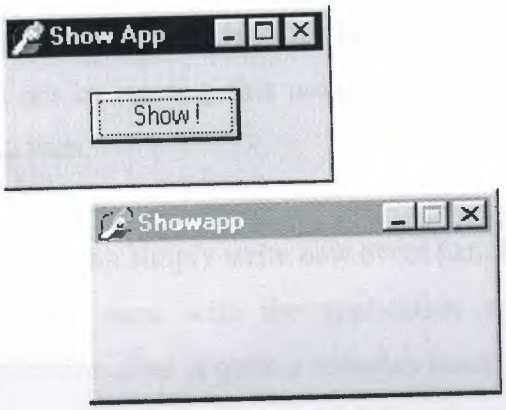

Figure 2.1.6.1: Shows Us, Presented By ShowApp Program's Hidden Application Window.

2.1.7 System Menu Aplications and TMainMenu Component

The Menu Designer lets us easily add menus to our form. We can simply add menu items directly into the Menu Designer window. You can add, delete, and rearrange menu items at design time and we do not have to run the program to see the results. Our application menu are always visible on the Form, as they will appear during runtime. The following example is explaining how the menu generates while the program runs.

```
procedure TForm1.FormCreate(Sender: TObject);
begin
// add a separator and a menu item to the system menu
AppendMenu (GetSystemMenu (Handle, FALSE), MF_SEPARATOR, 0, '');
AppendMenu    (GetSystemMenu    (Handle,    FALSE),    MF_STRING,
idSysAbout,
'&About...');
// add the same items to the application system menu
AppendMenu    (GetSystemMenu    (Application.Handle,    FALSE),
MF_SEPARATOR, 0,          '');
AppendMenu    (GetSystemMenu    (Application.Handle,    FALSE),
MF_STRING, idSysAbout,'&About...');
end;
```

The first part of the code adds the new separator and item to the system menu of the main form. The other two calls add the same two items to the application's system menu, simply by referring to Application.Handle. This is enough to display the updated system menu, as we can see by running this program. The next step is to handle the selection of the new menu item.

To handle form messages, we can simply write new event handlers or message-handling methods. We cannot do the same with the application window, simply because inheriting from the TApplication class is quite a complex issue. Most of the time we can just handle the OnMessage event of this class, which is activated for every message the application retrieves from the message queue.

To handle the OnMessage event of the global Application object, simply add an Application- Events component to the main form, and define a handler for the OnMessage event of this component. In this case, we only need to handle the wm_SysCommand message, and we only need to do that if the wParam parameter indicates that the user has selected the menu item we've just added, idSysAbout:

```
procedure TForm1.ApplicationEvents1Message(var Msg: tagMSG;
var Handled: Boolean);
begin
if (Msg.Message = wm_SysCommand) and (Msg.wParam = idSysAbout)
then
begin
ShowMessage ( 'SysMenu2 example');
Handled := True;
end;
end;
```

This method is very similar to the one used to handle the corresponding system menu item of the main form:

```
procedure WMSysCommand (var Msg: TWMSysCommand);
message wm_SysCommand;
...
procedure TForm1.WMSysCommand (var Msg: TWMSysCommand);
begin
// handle a specific command
if Msg.CmdType = idSysAbout then
ShowMessage ( 'SysMenu2 example');
inherited;
end;
```

2.1.8 Activating Application and Forms and TForm Componet

An application usually contains multiple forms: A main form, which is the primary user interface, and other forms such as dialog boxes, secondary windows (for instance, those that display OLE 2.0 data), and so on. We can begin our form design from one of the many form templates provided in the Object Repository. We can save any form you design as a template that you can reuse in other projects.ü

2.1.8.1 Tasks

1. To make the form stay on top of other open windows (for instance, the Project Manager or Alignment Palette) at runtime, set the FormStyle property to fsStayOnTop.

2. To remove the form's default scroll bars, change the value of the HorzScrollBar and VertScrollBar properties.
3. To make the form a MDI frame or MDI child, use the FormStyle property.
4. To change the form's border style, use the BorderIcons and BorderStyle properties. (The results are visible at runtime).
5. To change the icon for the minimized form, use the Icon property.
6. To specify the initial position of a form in the application window, use the Position property.
7. To specify the initial state of the form, (that is, minimized, maximized, or normal) use the WindowState property.
8. To define the working area of the form at runtime, use the ClientHeight and ClientWidth properties. (Note that ClientHeight and ClientWidth represent the area within the form's border; Height and Width represent the entire area of the form.).
9. To specify which control has initial focus in the form at runtime, use the ActiveControl property.
10. To pass all keyboard events to form, regardless of the selected control, use the KeyPreview property.
11. To specify a particular menu, if our form contains more than one menu, use the Menu property.

2.1.8.2 Activating Application and Forms

This questions answer is a simple example by named ActivApp which is explain itself or other words a selfexplanatory example. This example has two forms. Each form has a Label component (LabelForm) used to display the status of the form. The program uses text and color for this, as the handlers of the OnActivate and OnDeactivate events of the first form demonstrate:

```
procedure TForm1.FormActivate(Sender: TObject);
begin
LabelForm.Caption := 'Form2 Active';
LabelForm.Color := clRed;
end;
```

```
procedure TForm1.FormDeactivate(Sender: TObject);
begin
LabelForm.Caption := 'Form2 Not Active';
LabelForm.Color := clBtnFace;
end;
```

The second form has a similar label and similar code. The main form also displays the status of the entire application. It uses an ApplicationEvents component to handle the OnActivate and OnDeactivate events of the Application object. These two event handlers are similar to the two listed previously, with the only difference being that they modify the text and color of a second label of the form.

If we try running this program, we'll see whether this application is the active one and, if so, which of its forms is the active one. By looking at the output (see Figure 2.1.8.2.1) and listening for the beep, you can understand how each of the activation events is triggered by Delphi.

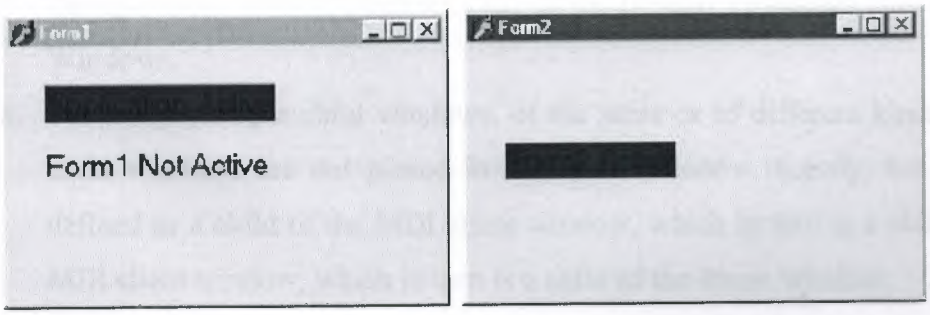

Figure 2.1.8.2.1: The ActivApp example shows whether the application is active and which of the application's forms is active.

2.1.8.3 Creating MDI Form Applications

A common approach for the structure of an application is MDI (Multiple Document Interface). An MDI application is made up of several forms that appear inside a single main form. If you use Windows Notepad, you can open only one text document, because Notepad isn't an MDI application. But with your favorite word processor, you can probably open several different documents, each in its own child window, because

they are MDI applications. All these document windows are usually held by a frame, or application, window.

2.1.8.3.1 MDI in Windows: A Technical Overview

1. The MDI structure gives programmers several benefits automatically. For example, Windows handles a list of the child windows in one of the pull-down menus of an MDI application, and there are specific Delphi methods that activate the corresponding MDI functionality, to tile or cascade the child windows. The following is the technical structure of an MDI application in Windows:
2. The main window of the application acts as a frame or a container.
3. A special window, known as the MDI client, covers the whole client area of the frame window. This MDI client is one of the Windows predefined controls, just like an edit box or a list box. The MDI client window lacks any specific user-interface element, but it is visible. In fact, you can change the standard system color of the MDI work area (called the Application Background) in the Appearance page of the Display Properties dialog box in Windows.
4. There are multiple child windows, of the same or of different kinds. These child windows are not placed in the frame window directly, but each is defined as a child of the MDI client window, which in turn is a child of the MDI client window, which in turn is a child of the frame window.

2.1.8.3.2 Frame and Child Windows in Delphi

Delphi makes the development of MDI applications easy, even without using the MDI Application template available in Delphi . Generally, however, the child form is not created at startup, and we need to provide a way to create one or more child windows. This can be done by adding a menu with a New menu item and writing the following code:

```
var
ChildForm: TChildForm;
begin
ChildForm := TChildForm.Create (Application);
ChildForm.Show;
```

Another important feature is to add a "Window" pull-down menu and use it as the value of the WindowMenu property of the form. This pull-down menu will automatically list all the available child windows. Of course, you can choose any other name for the pull-down menu, but Window is the standard.

To make this program work properly, we can add a number to the title of any child window when it is created:

```
procedure TMainForm.New1Click(Sender: TObject);
var
ChildForm: TChildForm;
begin
WindowMenu := Window1;
Inc (Counter);
ChildForm := TChildForm.Create (Self);
ChildForm.Caption := ChildForm.Caption + ' ' + IntToStr
(Counter);
ChildForm.Show;
end;
```

We can also open child windows, minimize or maximize each of them, close them, and use the Window pull-down menu to navigate among them. The closed forms in Delphi still exist, although they are not visible. In the case of child windows, hiding them won't work, because the MDI Window menu and the list of windows will still list existing child windows, even if they are hidden. For this reason, Delphi minimizes the MDI child windows when you Frame and Child Windows in Delphi try to close them. To solve this problem, we need to delete the child windows when they are closed, setting the Action reference parameter of the OnClose event to caFree.

2.1.8.3.3 The Mdi Form Example

This example is actually a full-blown MDI text editor, because each child window hosts a Memo component and can open and save text files. The child form has a Modified property used to indicate whether the text of the memo has changed (it is set to True in the handler of the memo's OnChange event). Modified is set to False in the Save and Load custom methods and checked when the form is closed (prompting to save the file).

This example is cased on ActionList componenet. The actions are available through some menu items and a toolbar, as you can see in Figure 2.1.8.3.3.1 Next, focus on the code of the custom actions. Once more, this example demonstrates that using actions makes it very simple to modify the user interface of the program, without writing any extra code. In fact, there is no code directly tied to the user interface.

One of the simplest actions is the ActionFont object, which has both an OnExecute handler, which uses a FontDialog component, and an OnUpdate handler, which disables the action (and hence the associated menu item and toolbar button) when there are no child forms:

```
procedure TMainForm.ActionFontExecute(Sender: TObject);
begin
if FontDialog1.Execute then
(ActiveMDIChild as TChildForm).Memo1.Font := FontDialog1.Font;
end;
procedure TMainForm.ActionFontUpdate(Sender: TObject);
begin
ActionFont.Enabled := MDIChildCount > 0;
end;
```

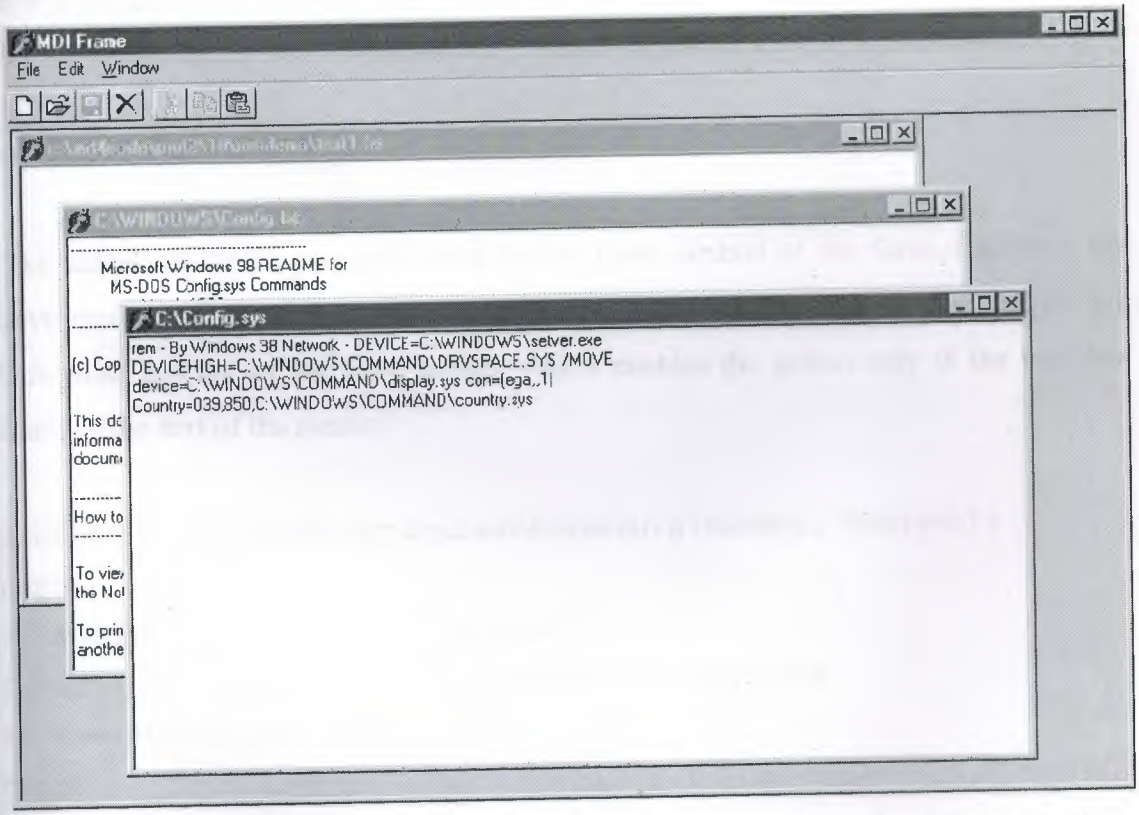

Figure 2.1.8.3.3.1: The Mdi Form Program Uses a Series of Predefined Delphi Actions Connected to a Menu and a Toolbar.

The action named New creates the child form and sets a default filename. The Open action calls the ActionNewExcecute method prior to loading the file:

```
procedure TMainForm.ActionNewExecute(Sender: TObject);
var
ChildForm: TChildForm;
begin
Inc (Counter);
ChildForm := TChildForm.Create (Self);
ChildForm.Caption :=
LowerCase (ExtractFilePath (Application.Exename)) + 'text' +
IntToStr (Counter) + '.txt';
ChildForm.Show;
end;
procedure TMainForm.ActionOpenExecute(Sender: TObject);
begin
if OpenDialog1.Execute then
begin
ActionNewExecute (Self);
(ActiveMDIChild as TChildForm).Load (OpenDialog1.FileName);
end;
end;
```

The actual file loading is performed by the Load method of the form. Likewise, the Save method of the child form is used by the Save and Save As actions. Notice the OnUpdate handler of the Save action, which enables the action only if the user has changed the text of the memo:

```
procedure TMainForm.ActionSaveAsExecute(Sender: TObject);
begin
// suggest the current file name
SaveDialog1.FileName := ActiveMDIChild.Caption;
if SaveDialog1.Execute then
begin
// modify the file name and save
ActiveMDIChild.Caption := SaveDialog1.FileName;
```

```
(ActiveMDIChild as TChildForm).Save;
end;
end;
procedure TMainForm.ActionSaveUpdate(Sender: TObject);
begin
ActionSave.Enabled := (MDIChildCount > 0) and
(ActiveMDIChild as TChildForm).Modified;
end;
procedure TMainForm.ActionSaveExecute(Sender: TObject);
begin
(ActiveMDIChild as TChildForm).Save;
end;
```

2.1.9 Delphi Standart Components

2.1.9.1 TLabel (Label) Component

Use TLabel to add text that the user can't edit to a form. This text can be used to label another control, and can set focus to that control when the user types an accelerator key.

Because TLabel is not a descendant of TWinControl, it does not have its own window and can't receive direct input from the keyboard. To add an object to a form that can respond to keyboard input (other than setting focus to another object when an accelerator key is typed) in addition to displaying text, use TStaticText. To add an object to a form that displays text that a user can scroll or edit, use TEdit.

2.1.9.2 TEdit (Edit) Component

Use a TEdit object to put a standard Windows edit control on a form. Edit controls are used to retrieve text that users type. Edit controls can also display text to the user. When only displaying text to the user, choose an edit control to allow users to select text and copy it to the Clipboard. Choose a label object if the selection capabilities of an edit control are not needed.

TEdit implements the generic behavior introduced in TCustomEdit. TEdit publishes many of the properties inherited from TCustomEdit, but does not introduce any new behavior. For specialized edit controls, use other descendant classes of TCustomEdit or derive from it.

2.1.9.3 TList (List) Component

TList, which stores an array of pointers, is often used to maintain lists of objects. TList introduces properties and methods to

1. Add or delete the objects in the list.
2. Rearrange the objects in the list.
3. Locate and access objects in the list.
5. Sort the objects in the list.

2.1.9.4 TButton (Button) Component

Use TButton to put a standard push button on a form. TButton introduces several properties to control its behavior in a dialog box setting. Users choose button controls to initiate actions.

To use a button that displays a bitmap instead of a label, use TBitBtn. To use a button that can remain in a depressed position, use TSpeedButton.

Note: Since the TButton caption is always centered, changing the BiDi alignment has no effect.

2.1.9.5 TComboBox (ComboBox) Component

A TComboBox component is an edit box with a scrollable drop-down list attached to it. Users can select an item from the list or type directly into the edit box.

At runtime, CLX combo boxes work differently than VCL combo boxes. With the CLX combo box, we can add an item to a drop-down list by entering text and pressing Enter in the edit field of a combo box. We can turn this feature off by setting InsertMode to ciNone. It is also possible to add empty (no string) items to the list in the combo box. Also, if we keep pressing the down arrow key, it does not stop at the last item of the combo box list. It cycles around to the top again.

2.1.9.6 TCheckBox (CheckBox) Component

A TCheckBox component presents an option for the user. The user can check the box to select the option, or uncheck it to deselect the option.

A check box is a toggle that lets the user select an on or off state. When the choice is turned on, the check box is checked. Otherwise, the check box is blank.

Set AllowGrayed to True to give the check box three possible states: checked, unchecked, and grayed. The State property indicates whether the check box is checked (cbChecked), unchecked (cbUnchecked), or grayed (cbGrayed).

Note: Check box controls display one of two binary states. The indeterminate state is used when other settings make it impossible to determine the current value for the check box.

2.1.9.7 TRadioButton (RadioButton) Component

Use TRadioButton to add a radio button to a form. Radio buttons present a set of mutually exclusive options to the user- that is, only one radio button in a set can be selected at a time. When the user selects a radio button, the previously selected radio button becomes unselected. Radio buttons are frequently grouped in a radio group box (TRadioGroup). Add the group box to the form first, then get the radio buttons from the Component palette and put them into the group box.

By default, all radio buttons that are directly contained in the same windowed control container, such as a TRadioGroup or TPanel, are grouped. For example, two radio buttons on a form can be checked at the same time only if they are contained in separate containers, such as two different group boxes.

2.1.9.8 TPanel (Panel) Component

Use TPanel to put an empty panel on a form. Panels have properties for providing a beveled border around the control, as well as methods to help manage the placement of child controls embedded in the panel.

We can also use panels to group controls together, similar to the way we can use a group box, but with a beveled border (or no border) rather than the group box outline. Panels are typically used for groups of controls within a single form. If you intend to use the same grouping in other forms, you may want to use a frame instead.

Although we can use a panel to implement a status bar or tool bar, it is recommended that you use the TToolBar and TStatusBar classes instead.

2.1.10 Delphi Win32 Components

2.1.10.1 TDateTimePicker (DateTimePicker) Component

TDateTimePicker is a visual component designed specifically for entering dates or times. In dmComboBox date mode, it resembles a list box or combo box, except that the drop-down list is replaced with a calendar illustration; users can select a date from the calendar. Dates or times can also be selected by scrolling with Up and Down arrows and by typing.

Date-time picker ignores the BiDiMode setting for right-to-left reading, displaying dates according to the system locale.

TDateTimePicker formats date and time values according to the date and time settings in the Regional Settings of the Control panel on the user's system. Because TDateTimePicker is a wrapper for a Windows control, these formats can't be changed by changing the formatting variables in the SysUtils unit. However, you can use the Windows API call DateTime_SetFormat to programmatically specify these settings.

2.1.10.2 TpageControl (PageControl) Component

Use TPageControl to create a multiple page dialog or tabbed notebook. TPageControl displays multiple overlapping pages that are TTabSheet objects. The user selects a page by clicking the page's tab that appears at the top of the control. To add a new page to a TPageControl object at design time, right-click the TPageControl object and choose New Page.

To create a tabbed control that uses only a single body portion (page), use TTabControl instead.

To create a new page in a page control at design time, right-click the control and choose New Page. At runtime, you add new pages by creating the object for the page and setting its PageControl property:

```
NewTabSheet = TTabSheet.Create(PageControl1);
NewTabSheet.PageControl := PageControl1;
```

To access the active page, use the ActivePage property. To change the active page, we can set either the ActivePage or the ActivePageIndex property.

2.1.11 Delphi Dialog Components

2.1.11.1 TPrintDialog (PrintDialog) Component

The TPrintDialog component displays a standard Windows dialog box for sending jobs to a printer. The dialog is modal and does not appear at runtime until it is activated by a call to the Execute method.

2.1.11.2 TPrinterSetupDialog (PrintSetupDialog) Component

TPrinterSetupDialog displays a modal Windows dialog box for configuring printers. The contents of the dialog vary depending on the printer driver selected. The dialog does not appear at runtime until it is activated by a call to the Execute method.

2.1.12 Delphi Additional Component

2.1.12.1 TSpeedButton (SpeedButton) Component

Use TSpeedButton to add a button to a group of buttons in a form. TSpeedButton introduces properties that can be used to set graphical images that represent the different button states (selected, unselected, disabled and so on). Use other properties to specify multiple images or to rearrange the images and text on the button. TSpeedButton also introduces properties that allow speed buttons to work together as a group. Speed buttons are commonly grouped in panels to create specialized tool bars and tool palettes.

The recommended way to implement the response of the button when the user clicks on it is to assign an action from an action list as the value of the Action property. By setting the Action property, you make the button a client of the action, and the action handles updating the button's properties and responding when the user clicks the button.

If we are not using an action to respond when the user clicks the button, then we can specify the button's response by writing an OnClick event handler.

2.1.12.2 TMaskEdit (MaskEdit) Component

We use a TMaskEdit object to put a masked edit control on our form. Masked edit controls validate the text the user enters against a mask that encodes the valid forms the text can take. The mask can also format text that is displayed to the user.

TMaskEdit implements the generic behavior introduced in TCustomMaskEdit. TMaskEdit publishes many of the properties and methods inherited from TCustomMaskEdit, but does not introduce any new behavior.

2.1.13 Delphi Samples Components

2.1.13.1 TSpinEdit (SpinEdit) Components

TSpinEdit is a generic implementation of spin boxes, as defined in TCustomSpinEdit. Spin boxes allow the user to choose from a range of numeric values by clicking on special "spin buttons", or by pressing vertical arrow keys. Users can also enter the value in a text box, as with an edit control. Spin boxes support non-numeric prefixes and suffixes, such as currency symbols, and special values with non-numeric representations.

2.2 Delphi Database Application Structures

These features enable us to build database applications with live connections to Paradox and dBASE tables, and the Local InterBase Server through the BDE. In many cases, we can create simple data access applications with these components and their properties without writing a line of code. Also in our project we worked with BDE paradox Database. The BDE is built into Delphi components so we can create database applications without needing to know anything about the BDE. The Delphi installation program installs drivers and sets up configuration for Paradox, dBASE, and the Local InterBase Server, so we can begin working with tables native to these systems immediately. The BDE Configuration Utility enables us to tailor database connections and manage database aliases.

Advanced BDE features are available to programmers who need more functionality. These features include local SQL, which is a subset of the industry-standard SQL that enables us to issue SQL statements against Paradox and dBASE tables; low-level API function calls for direct engine access; and ODBC support for communication with other ODBC-compliant databases, such as Access and Btrieve.

Delphi includes Borland ReportSmith, so we can embed database report creation, viewing, and printing capabilities in Delphi database applications. Delphi also includes the Database Desktop (DBD), a tool that enables us to create, index, and query desktop and SQL databases, and to copy data from one source to another.

BDE Configuration Utility Create and manage database connection Aliases used by the BDE.
Local InterBase Server Provides a single-user, multi-instance desktop SQL server for building and testing Delphi applications, before scaling them up to a production database, such as Oracle, Sybase, Informix, or InterBase on a remote server.

2.2.1 Understanding Delphi Database Architecture

In Chapter 1 we discuss the overview of Database architecture and Delphi database features and capabilities.

Delphi uses object-oriented components to create database applications, just as it does with non-database applications. Like standard components, database components have attributes, or properties, that are set by the programmer at design time. These properties can also be set programmatically at run time.

Database components have default behavior that enables them to perform useful functions with little or no programming. The Delphi Component palette provides two database component pages:

1. The Data Access page contains Delphi objects that simplify database access by
   encapsulating database source information, such as the database to connect to, the

tables in that database to access, and specific field references within those tables.

Examples of the most frequently used data access objects include TTable, TQuery, TDataSource, and TReport.

2. The Data Controls page contains data-aware user interface components for displaying database information in forms. Data Control components are like standard user interface components, except that their contents can be derived from or passed to database tables. Examples of the most frequently used data control components include TDBEdit, TDBNavigator, and TDBGrid.

Datasets, such as TTable, TQuery, and TStoredProc components, are not visible at run time, but provide applications their connection to data through the BDE. Data Control components are attached to dataset components by a TDataSource component, to provide a visual interface to data.

The following figure illustrates how data Access and Data Control components relate to the data, to one another, and to user interface in aDelphi Databse Application;

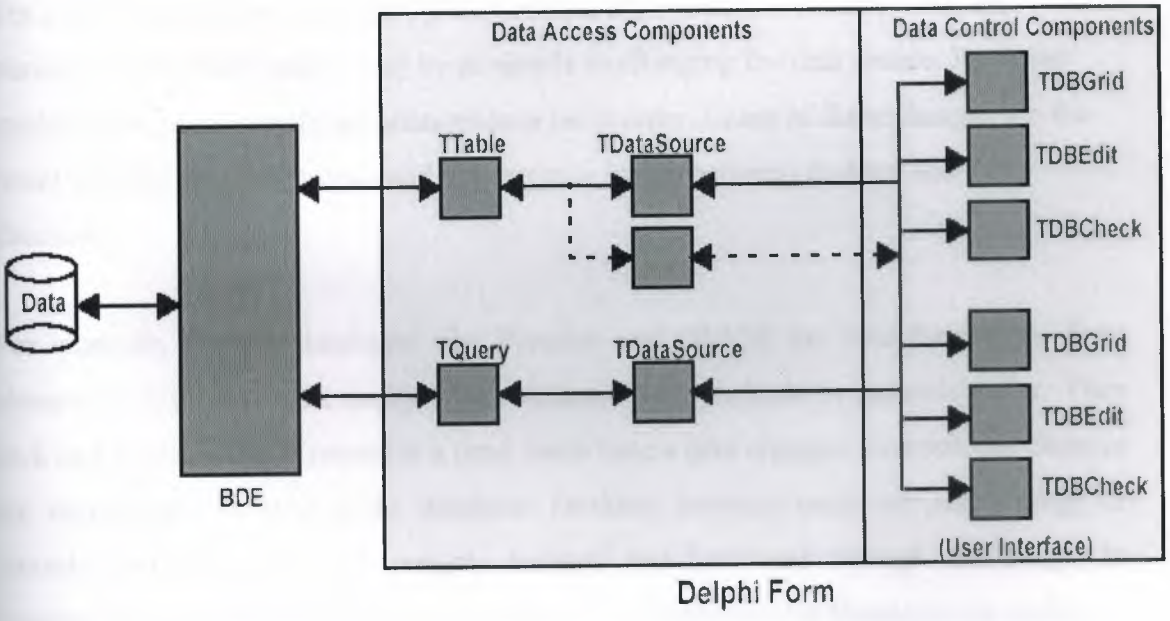

Figure 2.2.1.1: DataBase Components Architecture.

2.2.2 Overview of the Database Desktop

The Database Desktop (DBD) is a database maintenance and data definition tool. It enables programmers to query, create, restructure, index, modify, and copy database tables, including Paradox and dBASE files, and SQL tables. You do not have to own Paradox or dBASE to use the DBD with desktop files in these formats.

The DBD can copy data and data dictionary information from one format to another. For example, you can copy a Paradox table to an existing database on a remote SQL server

2.2.3 Developing Applications for Desktop and Remote Servers

Delphi Client/Server enables programmers to develop and deploy database client applications for both desktop and remote servers. One of Delphi's strengths is the ease with which an application developed for the desktop can be adapted to access data on a remote SQL server. The user interface need not change even if the source of the data changes. To an end user, a Delphi database application looks the same whether it accesses a local database file or a remote SQL database.

For simple applications that use *TQuery* components to access desktop data, the transition to a remote server may be as simple as changing the data source. For other applications, more significant changes may be in order. Some of these changes are the result of differing conventions and concurrency issues between desktop and SQL databases.

For example, desktop databases like Paradox and dBASE are record-oriented. They always display records in ascending or descending alphabetic or numeric order. They lock and access a single record at a time. Each time a user changes a record, the changes are immediately written to the database. Desktop database users can see a range of records, and can efficiently navigate forward and backward through that range. In contrast, data in SQL databases is set-oriented, and designed for simultaneous multiuser access. Record ordering must be specified as part of an SQL query. To accommodate multiuser access to data, SQL relies on transactions to govern access.

2.2.4 Delphi Borland Database Engine (BDE) Components

2.2.4.1 TDataSet (DataSet) Component

TDataSet introduces the basic properties, events, and methods for working with data. Many of these properties, events, and methods are abstract (Delphi) or pure virtual (C++) in TDataSet. Abstract or pure virtual declarations are declarations without implementations. At the TDataSet level they cannot be used or accessed. Developers must use or derive descendants of TDataSet that redeclare and implement these abstract or pure virtual methods. Many of the other TDataSet methods are declared and implemented in TDataSet as virtual methods, but the implementations are merely stubs that are reimplemented in descendants.

TDataSet has several descendants: TBDEDataSet, TCustomADODataSet,TIBCustomDataSet, TCustomSQLDataSet, and TCustomClientDataSet.

1. TBDEDataSet is the base class for datasets that access their data using the Borland Database Engine (BDE). TBDEDataSet descendants include TTable, TQuery, and TStoredProc. Developers who create custom dataset components that use the BDE derive them from TBDEDataSet, TDBDataSet, TQuery, TStoredProc, or TTable.
2. TCustomADODataSet is the base class for datasets that access their data using ActiveX Data Objects (ADO). TCustomADODataSet descendants include TADODataSet, TADOTable, TADOQuery, and TADOStoredProc. Developers who create custom dataset components that use ADO derive them from CustomADODataSet.
3. TIBCustomDataSet is the base class for datasets that directly access the data in InterBase tables. TIBCustomDataSet descendants include TIBDataSet, TIBTable, TIBQuery, and TIBStoredProc. Developers who create custom dataset components that directly access data in an InterBase database derive from TIBCustomDataSet.
4. TCustomSQLDataSet is the base class for unidirectional datasets. Unidirectional datasets are read-only datasets that permit only forward navigation. TCustomSQLDataSet descendants include the dbExpress datasets TSQLDataSet, TSQLQuery, TSQLTable, and TSQLStoredProc.

Developers who create custom dataset components that use dbExpress to access their data derive from TCustomSQLDataSet.

5. TCustomClientDataSet is the base class for in-memory datasets. Client datasets can work with data from files on disk or with data provided by another component via a provider. They cache that data in memory, maintain a record of any changes in a change log, and apply cached updates at a later point back to the source of the data. Developers who create custom datasets that store their data in an in-memory cache derive from TCustomClientDataSet.

Developer's can also derive custom dataset components directly from TDataSet, providing their own mechanisms for accessing and manipulating the data.

2.2.4.2 TStoredProc (StoredProc) Component

Use a TStoredProc object in BDE-based applications to use a stored procedure on a database server. A stored procedure is a grouped set of statements, stored as part of a database server's metadata (just like tables, indexes, and domains), that performs a frequently repeated, database-related task on the server and passes results to the client.

Many stored procedures require a series of input arguments, or parameters, that are used during processing. TStoredProc provides a Params property that enables an application to set these parameters before executing the stored procedure.

TStoredProc reuses the Params property to hold the results returned by a stored procedure. Params is an array of values. Depending on server implementation, a stored procedure can return either a single set of values, or a result set similar to the result set returned by a query.

2.2.4.3 TTable (Table) Component

Use TTable to access data in a single database table using the Borland Database Engine (BDE). TTable provides direct access to every record and field in an underlying database table, whether it is from Paradox, dBASE, Access, FoxPro, an ODBC-compliant database, or an SQL database on a remote server, such as InterBase, Oracle,

Sybase, MS-SQL Server, Informix, or DB2. A table component can also work with a subset of records within a database table using ranges and filters.

At design time, create, delete, update, or rename the database table connected to a TTable by right-clicking on the TTable and using the pop-up menu.

2.2.4.4 TQuery (Query) Component

Use TQuery to access one or more tables in a database using SQL statements. Query components can be used with remote database servers (such as Sybase, SQL Server, Oracle, Informix, DB2, and InterBase), with local tables (Paradox, InterBase, dBASE, Access, and FoxPro), and with ODBC-compliant databases.

Query components are useful because they can access more than one table at a time (called a "join" in SQL). Automatically access a subset of rows and columns in its underlying table(s), rather than always returning all rows and columns.

Note:TQuery is of particular importance to the development of scalable database applications. If there is any chance that an application built to run against local databases will be scaled to a remote SQL database server in the future, use TQuery components from the start to ensure easier scaling later.

2.2.5 Delphi Data Access Components

2.2.5.1 TDataSource (Datasource) Component

Use TDataSource to provide a conduit between a dataset and data-aware controls on a form that enable display, navigation, and editing of the data underlying the dataset. It Links two datasets in a master/detail relationship.

All datasets must be associated with a data source component if their data is to be displayed and manipulated in data-aware controls. Similarly, each data-aware control needs to be associated with a data source component in order for the control to receive and manipulate data. Data source components also link datasets in master-detail relationships.

2.2.6 Delphi Data Controls Components

2.2.6.1 TDBGrid (DBGrid) Component

Put a TDBGrid object on a form to display and edit the records from a database table or query. Applications can use the data grid to insert, delete, or edit data in the database, or simply to display it.

At runtime, users can use the database navigator (TDBNavigator) to move through data in the grid, and to insert, delete, and edit the data. Edits that are made in the data grid are not posted to the underlying dataset until the user moves to a different record or closes the application.

TDBGrid implements the generic behavior introduced in TCustomDBGrid. TDBGrid publishes many of the properties inherited from TCustomDBGrid, but does not introduce any new behavior.

2.2.6.2 TDBEdit (DBEdit) Component

Use TDBEdit to enable users to edit a database field. TDBEdit uses the Text property to represent the contents of the field.

TDBEdit permits only a single line of text. If the field may contain lengthy data that would require multiple lines, consider using a TDBMemo object.

If the application does not require the data-aware capabilities of TDBEdit, use an edit control (TEdit) or a masked edit control (TMaskEdit) instead, to conserve system resources.

2.2.6.3 TDBText (DBText) Component

Use TDBText to display the contents of a field in the current record of a dataset on a form. Field values displayed by database text controls cannot be modified by the user using the text control. To allow the user to edit the field value, use TDBEdit or TDBMemo instead.

If the application does not require the data-aware capabilities of TDBText, use the label component (TLabel) instead to conserve system resources.

2.2.6.4 TDBNavigator (DBNavigator) Component

Use the database navigator on forms that contain data-aware controls, such as TDBGrid or TDBEdit. TDBNavigator lets the user control the dataset when editing or viewing the data.

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

2.2.6.5 TDBMemo (DBMemo) Component

Use TDBMemo to let users edit a field that may contain lengthy textual data or to simply display the contents of such a field. TDBMemo uses the Text property to represent the contents of the field.

TDBMemo permits multiple lines of text. Thus, TDBMemo is appropriate for long alphanumeric fields or text binary large objects (BLOBs). For short alphanumeric fields, consider using a TDBEdit component instead.

If the application doesn't require the data-aware capabilities of TDBMemo, use a memo control (TMemo) instead, to conserve system resources.

2.2.6.6 TDBComboBox (DBComboBox) Component

Use TDBComboBox to allow users to change the value of a field on the current record in a dataset either by selecting an item from a list or by typing in the edit box part of the control. The selected item or entered text becomes the new value of the field if the database combo box's ReadOnly property is false. The combo box can be customized to enable or disable typing in the edit region of the control.

2.2.6.7 TDBLookupComboBox (DBLookupComBox) Component

Use TDBLookupComboBox to provide the user with a convenient drop-down list of lookup items for filling in fields that require data from another dataset.

If TDBLookupComboBox is linked to a lookup field component, it automatically reads the relationship between the field value and the lookup values in the lookup dataset from the field component. The relationship between field values and the corresponding values in the lookup dataset can also be explicitly set using the properties of the lookup combo box when the combo box is not linked to a lookup field component.

2.2.6.8 TDBChart (DBChart) Component

TDBChart derives from TChart /TCustomChart and inherits all TChart functionality. When a Chart Series is connected to a TDBChart component, TDBChart looks in the Series DataSource property.

If DataSource is a TTable, TQuery, TClientDataset or any valid Delphi DataSet component, TDBChart will automatically retrieve its records preserving all Filters and Ranges.We can also filter which records would be inserted by using the Series OnBeforeAdd event.

TDBChart also accepts Chart Series which are connected to another Chart Series and also Chart Series whose points are being manually added by coding. The main difference between TChart and TDBChart is that the last one NEEDS the Borland Database Engine to be correctly installed in the target machine, while TChart does not. The above would be useful in case our application do not need Tables, Querys or any standard Delphi database components.
Changing from a TChart to a TDBChart or vice versa can be done both at design and runtime by changing the Series ParentChart and the Series Values ValueSource properties.

2.2.7 Locating Records in a Table

To show you an example of the use of the Locate method, built the Search example, which has a table connected to EMPLOYEE.DB. The form prepared has the data-awareedit boxes inside a scroll box aligned to the client area, so that a user can freely resize the form without any problems. When the form becomes too small, scroll bars will appear automatically in the area holding the edit boxes. Another feature is a toolbar with buttons connected to Navigating a Dataset some of the predefined dataset actions available in the ActionList component plus two custom actions to host the search code.

The searching capabilities are activated by the two buttons connected to custom actions. The first button is connected to ActionGoto, used for an exact match, and the second to ActionGoNear, for a partial match. In both cases, we want to compare the text in the edit box with the LastName fields of the EMPLOYEE table. If the local table has an index on the field (as in the specific case) Locate will use it, but the method will work with or without indexes (only at a different speed).

If we've never used Locate, at first sight the help file won't be terribly clear. The idea is that you must provide a list of fields you want to search, and a list of values, one for each field. If you pass only one field, the value is passed directly, as in the case of the example:

```
procedure TSearchForm.ActionGotoExecute(Sender: TObject);
begin
if not Table1.Locate ('LastName', EditName.Text, []) then
MessageDlg ('"' + EditName.Text + '" not found', mtError,
[mbOk], 0);
end;
```

If we search for multiple fields, we have to pass a variant array with the list of the values we
want to match. The variant array can be created from a constant array with the VarArrayOf function or from scratch using the VarArrayCreate call. This is a code snippet from the example:

```
Table1.Locate
('LastName;FirstName',VarArrayOf(['Cook', 'Kevin']),[])
```

Finally, we can use the same method to look for a record even if we know only the initial portion of the field we are looking for. Simply add the loPartialKey flag to the Options parameter (the third) of the Locate call.

2.2.8 The Total of a Table Column

Now we will see how we can change some data in the table through the program code. The idea behind this example is quite simple. The EMPLOYEE table we have been

using has a Salary field. A manager of the company could indeed browse through the table and change the salary of a single employee. But what will be the total salary expense for the company? And what if the manager wants to give a 10 percent salary increase (or decrease) to everyone?

These are the two aims of the Total example, which is an extension of the previous program. The toolbar of this new example has some more buttons and actions. There are a few other minor changes from the previous example. I opened the Fields editor of the table and removed the Table1Salary field, which was defined as a TFloatField. Then I selected the New Field command and added the same field, with the same name, but using the TCurrencyField data type. This is not a calculated field; it's simply a field converted into a new (but equivalent) data type. Using this new field type the program will default to a new output format, suitable for currency values.

Now we can turn our attention to the code of this new program. First, let's look at the code of the total action. This action lets us calculate the sum of the salaries of all the employees, then edit some of the values, and compute a new total. Basically, we need to scan the table, reading the value of the Table1Salary field for each record:

```
var
Total: Real;
begin
Total := 0;
Table1.First;
while not Table1.EOF do
begin
Total := Total + Table1Salary.Value;
Table1.Next;
end;
MessageDlg ( 'Sum of new salaries is ' +
Format ('%m', [Total]), mtInformation, [mbOk], 0);
End
```

This code works as you can see from the output in Figure 2.2.8 , but it has some problems. One problem is that the record pointer is moved to the last record, so the

previous position in the table is lost. Another is that the user interface is refreshed many times during the operation.

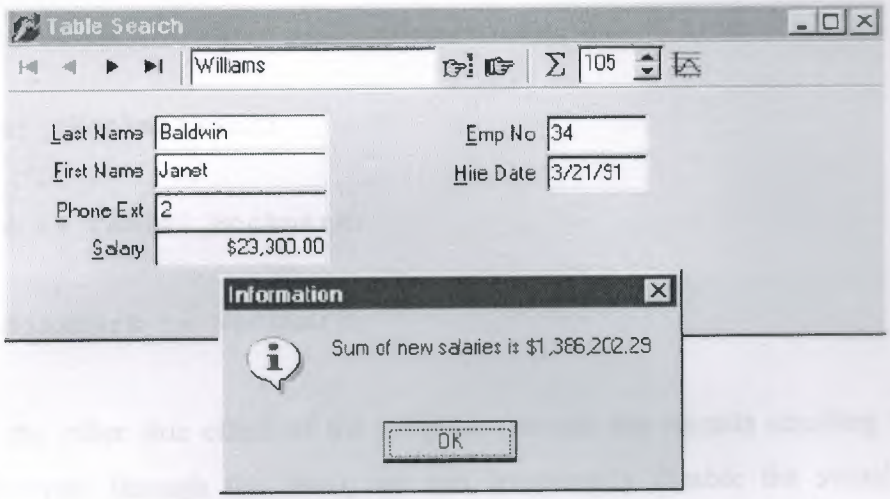

Figure 2.2.8.1 : Shows us the Total program's output about workers' sum of salaries.

2.2.9 Using Bookmarks

To avoid these two problems, we need to disable updates and to store the current position of the record pointer in the table and restore it at the end. This can be accomplished using a table bookmark, a special variable storing the position of a record in a database table. Delphi's traditional approach is to declare a variable of the TBookmark data type, and initialize it while getting the current position from the table:

```
var
Bookmark: TBookmark;
begin
Bookmark := Table1.GetBookmark;
```

At the end of the ActionTotalExecute method, we can restore the position and delete the bookmark with the following two statements:

```
Table1.GotoBookmark (Bookmark);
Table1.FreeBookmark (Bookmark);
```

As a better (and more up-to-date) alternative, we can use the Bookmark property of the TDataset class, which refers to a bookmark that is disposed of automatically. (This is technically implemented as an *opaque string*, a structure subject to string lifetime

management, but it is not a string, so you're not supposed to look at what's inside it.) This is how we can modify the code above:

```
var
Bookmark: TBookmarkStr;
begin
Bookmark := Table1.Bookmark;
...
Table1.Bookmark := Bookmark;
```

To avoid the other side effect of the program (we see the records scrolling while the routine browses through the data), we can temporarily disable the visual controls connected with the table. The table has a DisableControls method we can call before the while loop starts and an EnableControls method we can call at the end, after the record pointer is restored.

Finally, we face some dangers from errors in reading the table data, particularly if the program is reading the data from a server using a network. If any problem occurs while retrieving the data, an exception takes place, the controls remain disabled, and the program cannot resume its normal behavior. So we should use a try/finally block. Actually, if you want to make the program 100 percent error-proof, you should use two nested try/finally blocks. Including this change and the two discussed above, here is the resulting code:

```
procedure TSearchForm.ActionTotalExecute(Sender: TObject);
var
Bookmark: TBookmarkStr;
Total: Real;
begin
Bookmark := Table1.Bookmark;
try
Table1.DisableControls;
Total := 0;
try
Table1.First;
while not Table1.EOF do
```

```
begin
Total := Total + Table1Salary.Value;
Table1.Next;
end;
finally
Table1.EnableControls;
end
finally
Table1.Bookmark := Bookmark;
end;
MessageDlg ('Sum of new salaries is ' +
Format ('%m', [Total]), mtInformation, [mbOK], 0);
end;
```

This code to show you an example of a loop to browse the contents of a table, but keep in mind that there is an alternative approach based on the use of a SQL query returning the sum of the values of a field. When you use a SQL server, the speed advantage of a SQL call to compute the total can be very large, since you don't need to move all the data of each field from the server to the client computer. The server sends the client only the final result.

2.2.10 Editing a Table Column

The code of the increase action is similar to the one we have just seen. The ActionIncrease- Execute method also scans the table, computing the total of the salaries, as the previous method did. Although it has just two more statements, there is a key difference. When we increase the salary, you actually change the data in the table. The two key statements are within the while loop:

```
while not Table1.EOF do
begin
Table1.Edit;
Table1Salary.Value := Round (Table1Salary.Value *
SpinEdit1.Value) / 100;
Total := Total + Table1Salary.Value;
Table1.Next;
end;
```

The first statement brings the table into edit mode, so that changes to the fields will have an immediate effect. The second statement computes the new salary, multiplying the old one by the value of the SpinEdit component (by default, 105) and dividing it by 100. That's a 5 percent increase, although the values are rounded to the nearest dollar. With this program, you can change salaries by any amount even double the salary of each employee with the click of a button.

2.2.11 Customizing a Database Grid

Unlike most other data-aware controls, which are quite simple to use, the DBGrid control has many options. The following sections explore some of the advanced operations you can do using a DBGrid control. A first example shows how to draw in a grid, a second one shows how to clone the behavior of a check box for a Boolean selection inside a grid, and the final example shows how to use the multiple-selection feature of the grid.

2.2.11.1 A Grid Allowing Multiple Selection

This example customizing the DBGrid control relates to multiple selection. we can set up the DBGrid so that a user can select multiple rows (that is, multiple records). This is very easy, since all we have to do is toggle the dgMultiSelect element of the Options property of the grid. Once you've selected this option, a user can keep the Ctrl key pressed and click with the mouse to select multiple rows of the grid, with the effect we can see in Figure 2.2.11.1.1.

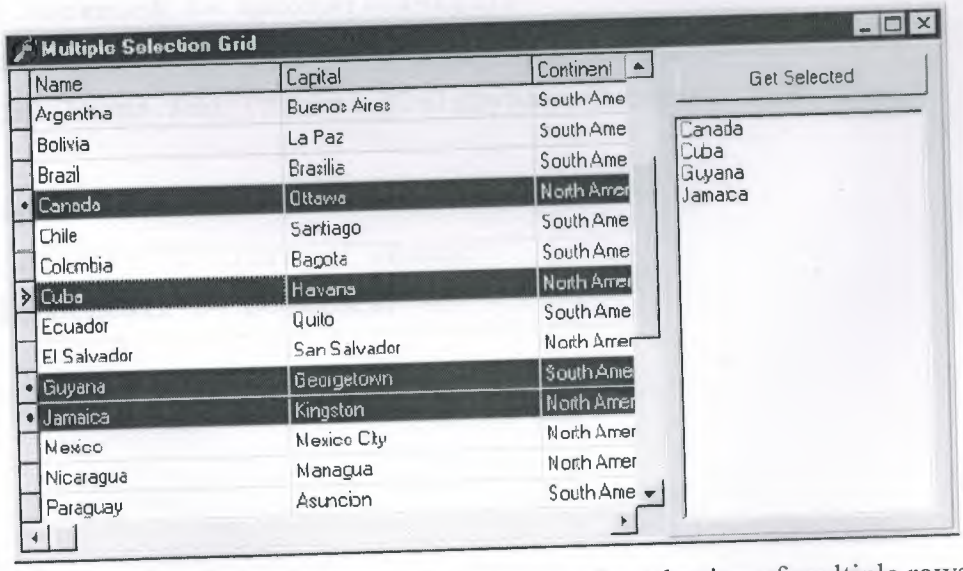

Figure 2.2.11.1.1 DBGrid control that allows the selection of multiple rows.

Since the database table can have only one active record, what information is stored in the grid for the selected items? The grid simply keeps a list of bookmarks to the selected records. This list is available in the SelectedRows property, which is of type TBookmarkList. Besides accessing the number of objects in the list with the Count property, we can get to each bookmark with the Items property, which is the default array property. Each item of the list is on a TBookmarkStr type, which represents a bookmark pointer you can assign to the Bookmark property of the table.

```
procedure TForm1.Button1Click(Sender: TObject);
var
I: Integer;
BookmarkList: TBookmarkList;
Bookmark: TBookmarkStr;
begin
// store the current position
Bookmark := Table1.Bookmark;
try
// empty the list box
ListBox1.Items.Clear;
// get the selected rows of the grid
BookmarkList := DbGrid1.SelectedRows;
for I := 0 to BookmarkList.Count - 1 do
begin
// for each, move the table to that record
Table1.Bookmark := BookmarkList[I];
// add the name field to the listbox
ListBox1.Items.Add (Table1.FieldByName ( 'Name').AsString);
end;
finally
// go back to the initial record
Table1.Bookmark := Bookmark;
end;
end;
```

2.3 Database Applications with Standard Controls

Although it is generally faster to write Delphi applications based on data-aware controls, this is certainly not required. When you need to have very precise control over the user interface of a database application, you might want to customize the transfer of the data from the field objects to the visual controls. The general view is that this is necessary only in very specific cases, as you can customize the data-aware controls extensively by setting the properties and handling the events of the field objects. However, trying to work without the data-aware controls should help you understand the default behavior of Delphi, and introduce some more database-related events. The development of an application not based on data-aware controls can follow two different approaches.

2.3.1 Sending Requests to the Database

You can further customize the user interface of our application if you decide not to handle the same sequence of editing operations as in standard Delphi data-aware controls. This allows us complete freedom, although there might be some side effects (such as limited ability to handle concurrency, something).

For this new example the first edit box with another combo box, and replaced all the buttons related to table operations (which corresponded to DBNavigator buttons) with two custom ones, used to get the data from the database and send an update to it. To underline the difference of this example removed the DataSource component.

The GetData method, connected with the corresponding button, simply gets the fields corresponding to the record indicated in the first combo box:

```
procedure TForm1.GetData;
begin
Table1.FindNearest ([ComboName.Text]);
ComboName.Text := Table1Name.AsString;
EditCapital.Text := Table1Capital.AsString;
ComboContinent.Text := Table1Continent.AsString;
EditArea.Text := Table1Area.AsString;
EditPopulation.Text := Table1Population.AsString;
end;
```

This method is called whenever the user presses the button, selects an item of the combo box, or presses the Enter key while in the combo box:

```
procedure TForm1.ComboNameClick(Sender: TObject);
begin
GetData;
end;
procedure TForm1.ComboNameKeyPress(Sender: TObject; var Key:
Char);
begin
if Key = #13 then
GetData;
end;
```

To make this example work smoothly, at start-up the combo box is filled with all the names of the countries of the table:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
// fill the list of names
Table1.Open;
while not Table1.Eof do
begin
ComboName.Items.Add (Table1Name.AsString);
Table1.Next;
end;
end;
```

With this approach, the combo box becomes a sort of selector of the record.

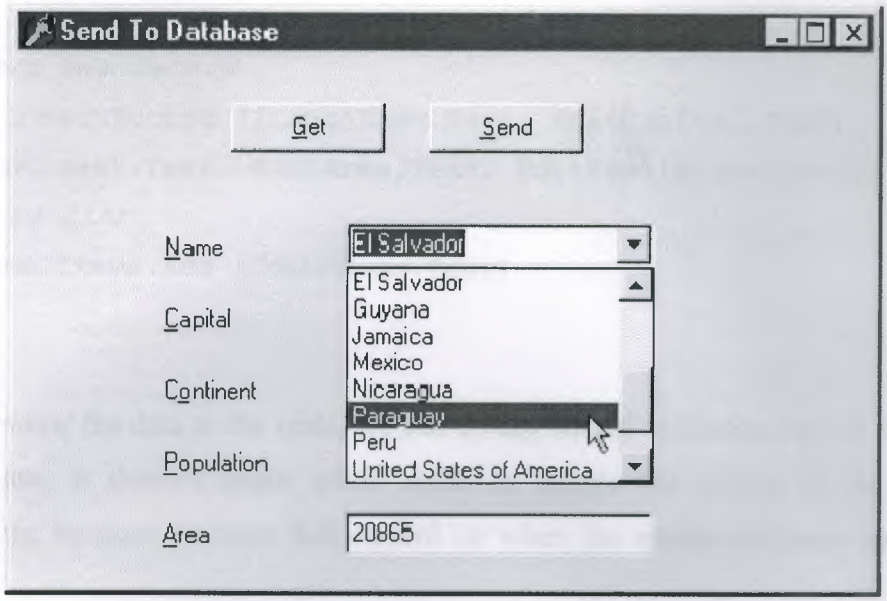

Figure 2.3.1.1: We can Select the Record we want to See in a Combo Box.

Finally, the user can change the values of the controls and click the Send button. The code to be executed depends on whether the operation is an update or an insert. We can determine this by looking at the name (although with this code, a wrong name cannot be modified any more):

```
procedure TForm1.SendData;
begin
// raise an exception if there is no name
if ComboName.Text = '' then
raise Exception.Create ('Insert the name');
// check if the record is already in the table
if Table1.FindKey ([ComboName.Text]) then
begin
// modify found record
Table1.Edit;
Table1Capital.AsString := EditCapital.Text;
Table1Continent.AsString := ComboContinent.Text;
Table1Area.AsString := EditArea.Text;
Table1Population.AsString := EditPopulation.Text;
Table1.Post;
end
else
```

```
begin
// insert new record
Table1.InsertRecord ([ComboName.Text, EditCapital.Text,
ComboContinent.Text, EditArea.Text, EditPopulation.Text]);
// add to list
ComboName.Items.Add (ComboName.Text)
end;
```

Before sending the data to the table, we can do any sort of validation test on the values. In this case, it doesn't make much sense to handle the events of the database components, because we have full control on when the update or insert operation is done.

2.3.2 Database Events

To further illustrate how we can use the events of a database application, written a simple program that logs all the events being fired. This program handles all of the events of a table and a data source component . For each event, simply send its description to a list box, with the effect you can see in Fig.2.3.2.1.1.

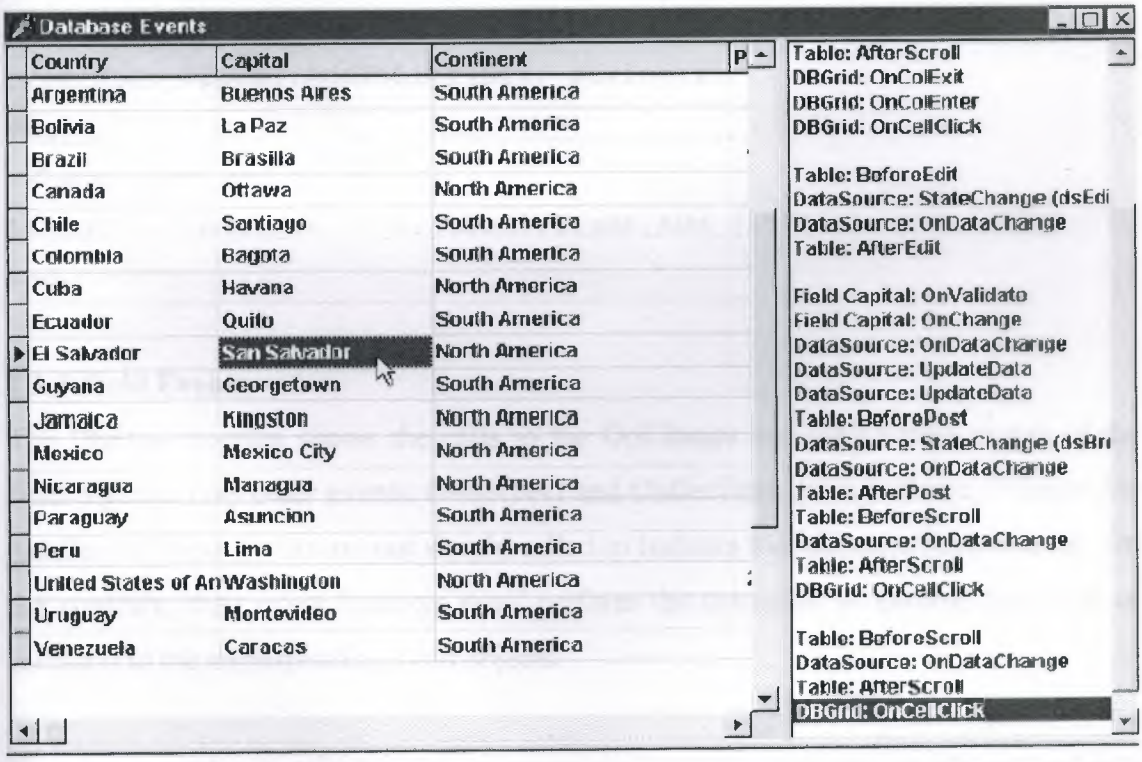

Figure 2.3.2.1.1: Which Logs All the Events Related to Database Components.

Most of the event handlers simply display the name of the component and that of the event,as in;

```
procedure TForm1.Table1AfterEdit(DataSet: TDataset);
begin
AddToList ( 'Table: AfterEdit');
end;
```

The field events are slightly more complex, but they use a single handler for the variousfield components:

```
procedure TForm1.FieldChange(Sender: TField);
begin
AddToList ( 'Field ' + Sender.FieldName +  ': OnChange');
end;
```

The form's AddToList method adds a new item to the list box and selects it, automatically scrolling the list if required:

```
procedure TForm1.AddToList(Str: string);
begin
// add item and select it
Listbox1.ItemIndex := Listbox1.Items.Add (Str);
end;
```

2.3.3 Field Events

The DbEvts program shows the calls to the OnChange and OnValidate events of the field objects. Two other events, OnSetText and OnGetText, are not shown, because the handlers of these events are not simply called to indicate that an operation occurred. On the contrary, their event handlers must perform the operation of getting data from or setting it to the corresponding field objects.

These two events are quite special, and their use is not as simple as it might seem at first sight. For this reason, they require a separate example, named FldText. This is only a

slight revision of the DbAware example described earlier in this chapter, replacing the DBRadioGroup control with a DBListbox control. The problem is that a DBListBox control directly connects with a string field, while we want to connect it with an integer field, with each value indicating an option. Of course, we don't want a user to see or select a number, so we have to map the numbers stored in the database to the strings visible on the screen. In the earlier example, the DBRadioGroup control provided that mapping. Now we have to use an alternative approach.

In the FldText example, the Department field has two handlers for the OnGetText and OnSetText events. In the OnGetText event handler, we can extract the numeric value of the Sender field and set the value of the Text reference parameter:

```
procedure TDbaForm.Table1DepartmentGetText(Sender: TField;
var Text: String; DisplayText: Boolean);
begin
case Sender.AsInteger of
1: Text := 'Sales';
2: Text := 'Accounting';
3: Text := 'Production';
4: Text := 'Management';
else
Text := '[Error]';
end;
end;
procedure TDbaForm.Table1DepartmentSetText(Sender: TField; const
Text: String);
begin
if Text = 'Sales' then
Sender.Value := 1
else if Text = 'Accounting' then
Sender.Value := 2
else if Text = 'Production' then
Sender.Value := 3
else if Text = 'Management' then
Sender.Value := 4
else
```

```
raise Exception.Create ( 'Error in Department field conversion');
end;
```

The effect is that not only is the value visible in the DBListBox (as you can see in Figure 2.3.3.1), it also shows up in the DBGrid. By contrast, in the DbAware example, the grid displayed the numeric value.

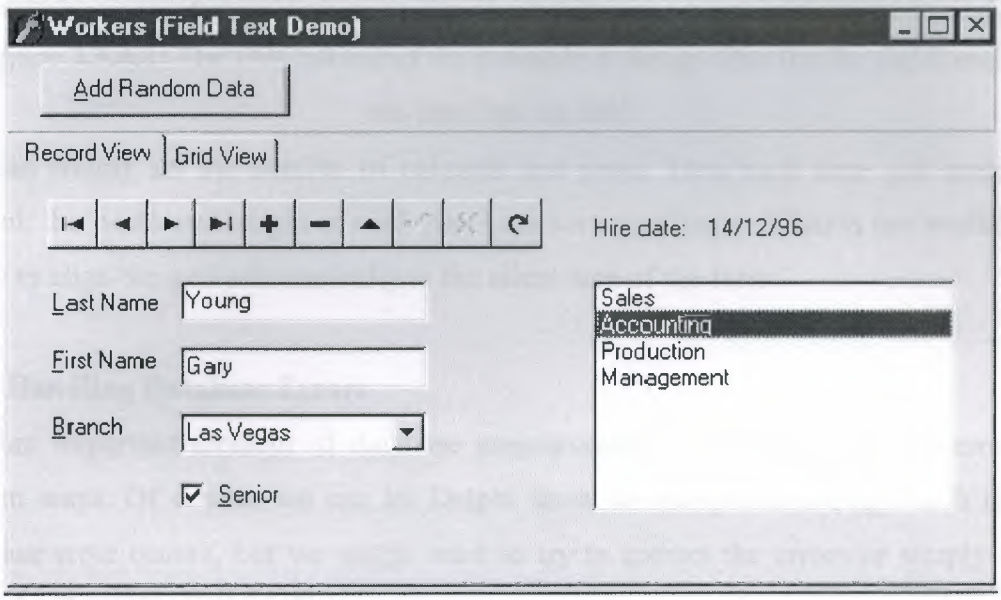

Figure 2.3.3.1: The Output of the FldText Example, Which Demonstrates the Use of the OnGetText and OnSetText Events of the Field Objects.

2.3.4 A Multirecord Grid

So far we have seen that you can either use a grid to display records of a database table or build a form with specific data-aware controls for the various fields, accessing the records one by one. There is a third alternative: use a multirecord object (a DBCtrlGrid), which allows us to place many data-aware controls in a small area of a form and automatically duplicate these controls for multiple records.

At design time, we simply work on the active portion of the grid (see Figure 2.3.4.1, on the right), and at run time, we can see these controls replicated multiple times (see Figure 2.3.4.1, on the left).

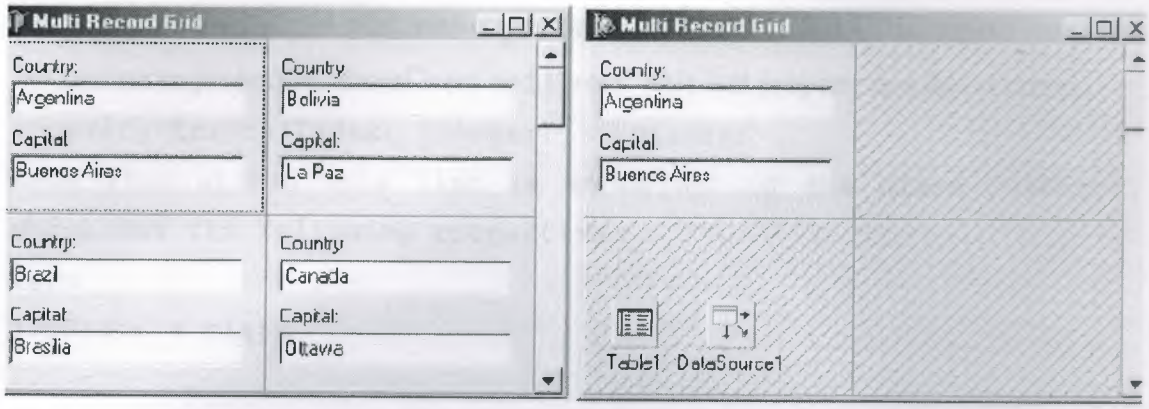

Figure 2.3.4.1: The DBCtrlGrid of the example at design time (on the right) and at run time (on the left).

We can simply set the number of columns and rows. Then each time you resize the control, the width and height of each panel are set accordingly. What is not available is a way to align the grid automatically to the client area of the form.

2.3.5 Handling Database Errors

Another important element of database programming is handling database errors in custom ways. Of course, we can let Delphi show an exception message each time a database error occurs, but we might want to try to correct the errors or simply show more details.

There are basically three approaches we can use to handle database-related errors:

1. We can wrap a try/except block around risky database operations, such as a call to the Open method of Query or to the Post method of a dataset. This is not possible when the operation is generated by the interaction with a data-aware control.
2. We can install a handler for the OnExpection event of the global Application object or use the ApplicationEvent component, as described in the next example.
3. We can handle specific events of the datasets related to errors, as OnPostError, OnEditError, OnDeleteError, OnUpdateError. These events will be discussed later in the example.

While most of the exception classes in Delphi simply deliver an error message, with database exceptions we see a list of errors, showing local BDE error codes and also the

native error codes of the SQL server you are connected to. The EDBEngineError class has two more properties, ErrorCount and Errors. This last property is a list of errors:

```
property Errors[Index: Integer]: TDBError;
```

Each item within this list is an object of the class TDBError, which has the following properties:

```
type
TDBError = class
...
public
property Category: Byte read GetCategory;
property ErrorCode: DBIResult read FErrorCode;
property SubCode: Byte read GetSubCode;
property Message: string read FMessage;
property NativeError: Longint read FNativeError;
end;
```

We used this information to build a simple database program showing the details of the errors in a memo component. To handle all of the errors, the DBError example installs a handler for the OnException event of an ApplicationEvents component. The event handler simply calls a specific method used to show the details of the database error, in case it is an EDBEngineError:

```
procedure TForm1.ApplicationEvents1Exception (Sender: TObject;
E: Exception);
begin
Beep;
if E is EDBEngineError then
ShowError (EDBEngineError (E))
else
ShowMessage (E.Message);
end;
```

We decided to separate the code used to show the error to make it easier for us to copy this code and use it in different contexts. Here is the code of the ShowError method, which outputs all of the available information to the Memo1 component that we added to the form:

```
procedure TForm1.ShowError(E: EDBEngineError);
var
I: Integer;
begin
Memo1.Lines.Add( '');
Memo1.Lines.Add( 'Error: ' + (E.Message));
Memo1.Lines.Add( 'Number of errors: ' + IntToStr(E.ErrorCount));
// iterate through the Errors records
for I := 0 to E.ErrorCount - 1 do
begin
Memo1.Lines.Add( 'Message: ' + E.Errors[I].Message);
Memo1.Lines.Add( ' Category: ' + IntToStr(E.Errors[I].Category));
Memo1.Lines.Add( '        Error        Code:        '        +
IntToStr(E.Errors[I].ErrorCode));
Memo1.Lines.Add( ' SubCode: ' + IntToStr(E.Errors[I].SubCode));
Memo1.Lines.Add( '        Native        Error:        '        +
IntToStr(E.Errors[I].NativeError));
Memo1.Lines.Add( '');
end;
end;
```

Besides this error-handling code, the program has a table and a query, along with the errorrelated event handlers. As already mentioned, we can install an event handler related to specific errors of a dataset. The three events OnPostError, OnDeleteError, and OnEditError have the same structure. Their handlers receive as parameters the dataset, the error itself, and an action we can request from the system; this can be set to daFail, daAbort, or daRetry:

```
Procedure TForm1.Table1PostError(DataSet: TDataSet; E:
EDatabaseError;
var Action: TDataAction);
begin
Memo1.Lines.Add ( ' -> Post Error: ' + E.Message);
end;
```

If we don't specify an action, as in the code above, the default daFail is used, and the exception reaches the global handler. Using daAbort stops the exception and can be used if us event handler already displays a message. Finally, if we have a way to determine the cause of the error and fix it, we can use the daRetry action.

The example has also a DBGrid connected with the table. You can use the DBGrid to perform some illegal operations, such as adding a new record with the same key as an existing one or trying to execute illegal SQL queries.

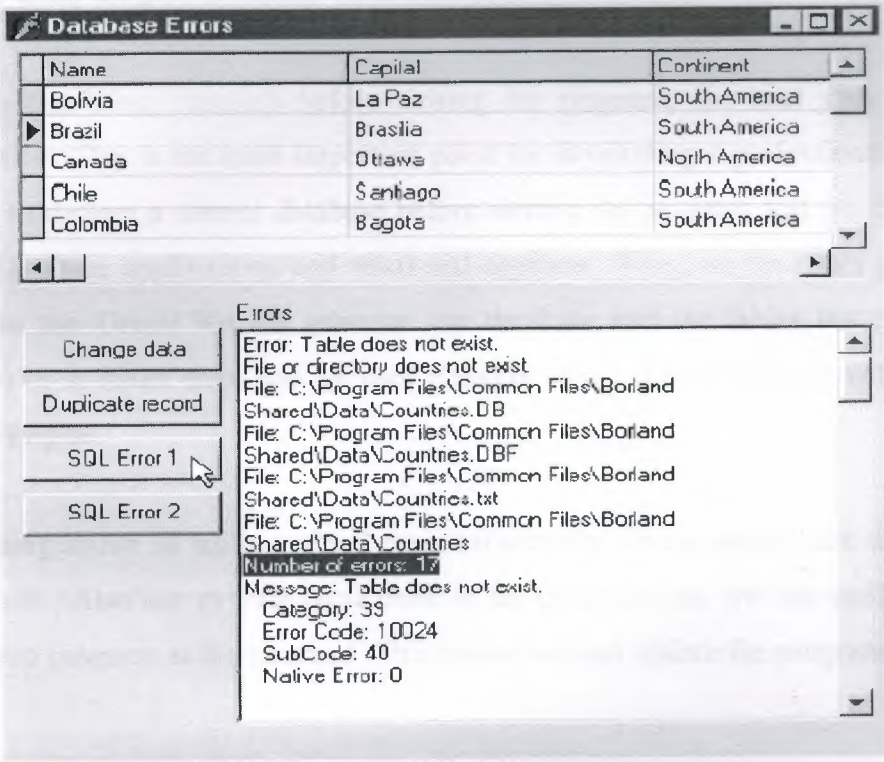

Figure 2.3.5.1: Pressing the Four Buttons on the left of the Memo Generate Errors.

Chapter III: Pharmacy Development Suite

(Experimental Work)

3.1 Short Introduction to Pharmacy Automation Program

First of all in our Graduation Project, we search thoroughly how Pharmacy works? What is the important structures while we design and develop a Pharmacy Automation Program .We did great research about management of Pharmacy .The important point of the software engineering is , understanding the targets to reach the maximum efficiency when the program ends.

When we finish our research before writing the program, the next step was Data Manipulation. This is the most important point for developing a professional program. We have to declare a correct database before writing the program and we discuss the Modern Database applications and relational algebras .When we took this project we planned to use Oracle.We are generate our database and the tables but this is not suitable to us at future steps and we used Paradox which is more flexible with Borland Database Engine.

We are using above 20 tables under 8 program sections .These sections are the basis of the program . Also our experimental work is not a version but we can easily say it is beta version program at the moment .This means we can update the program easily for future.

These 4 important sections are;

1. Pharmacy description module,
2. Depot description module,
3. Product description module,
4. Waybill description module (see *Appendix* A).

The next steps of this chapter includes the important Program sections and continuously explained what we did when we design and develop the Program step by step. We use all of the components that we have explained in Chapter 2 and we give the short program modules belong these components in the next sections.

3.2 Pharmacy Description Module

This program section includes the pharmacy informations , like Pharmacy name, author name and surname, pharmacist name and surname etc. This section is the easiest section but it is very important when we sell this program to pharmacies. At the below figure 3.2.1 shows the screen layout of the pharmacy description module.

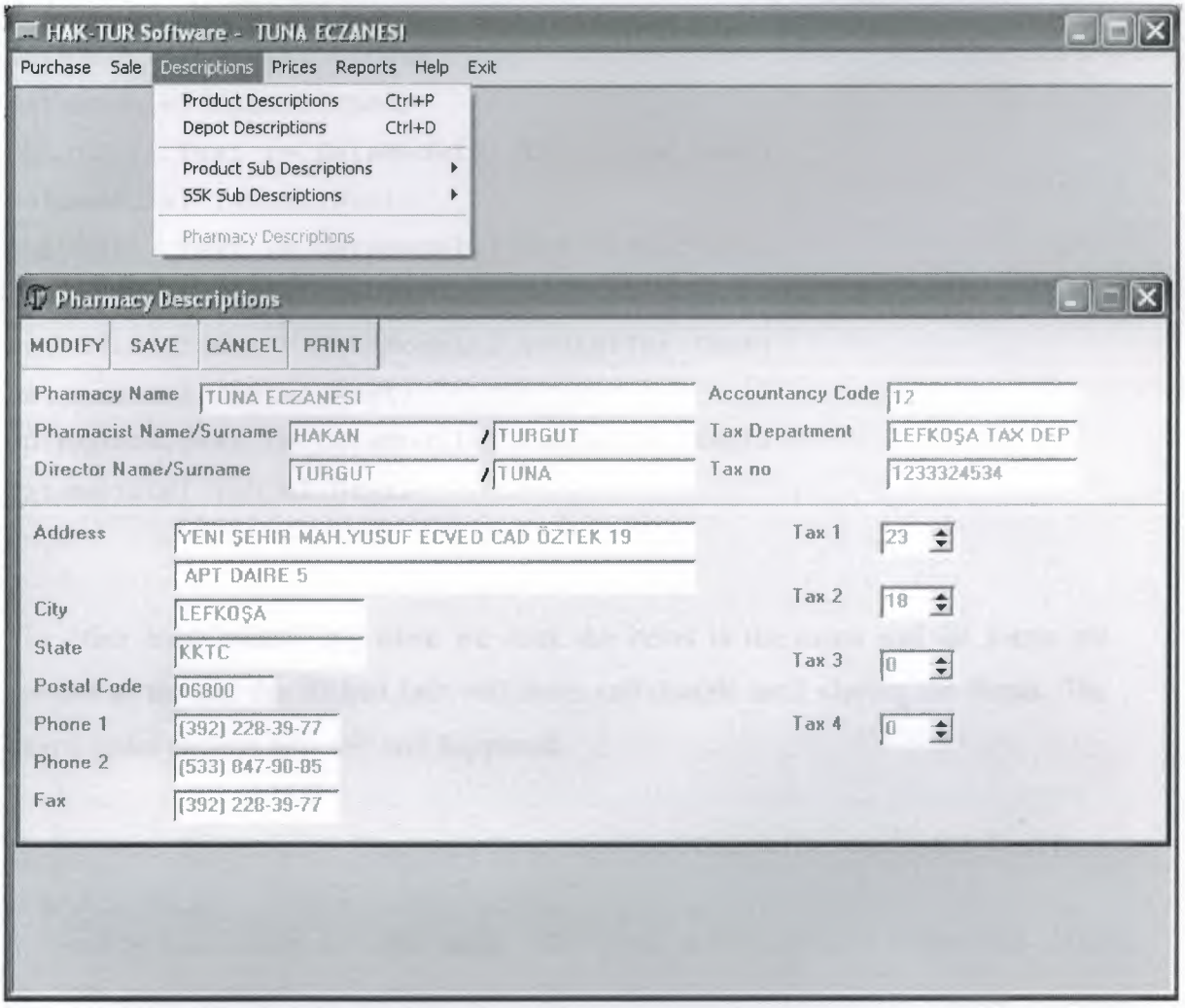

Figure 3.2.1: Pharmacy Description Screen Layout while Executed.

As you can see the program is useful for everybody . Our program has too much error controls. The simplest one is TButton and TDbtext combinations .When you press the "new" button in the form, the save button is enabled and also the TDBTexts. With this program we used too many components (as we declared in Chapter 2). For example the TSpinEdit component has no relation between TDataSource component which is connected to TTable or Tquery component for inserting ,deleting , modifying the database. For that reason we design a very flexible module for set the database fields to

the SpinEdit 's or other declared variables or homogeneus components when opening the Pharmacy Description Form.

```
Procedure TPharmacydesc.Formcreate (Sender :TObject);
begin
//Adding the default saved tax rates to the SpinEdit components
text //property
DataModule1.Table1.Close;
DataModule1.Table1.Open;
SpinEdit1.Text := DataModule1.Table1Tax.Text;
Datamodule1.Table1.Next;
SpinEdit2.Text := DataModule1.Table1Tax.Text;
Datamodule1.Table1.Next;
SpinEdit3.Text := DataModule1.Table1Tax.Text;
Datamodule1.Table1.Next;
SpinEdit4.Text := DataModule1.Table1Tax.Text;
Datamodule1.Table1.Next;
End;
```

The other error control is , when we click the items in the menu and the forms are opened so far and the clicked link will worn and disable until closing the forms. The below codes explain how this will happened.

```
procedure Tpharmacy.PharmacyDescription1Click(Sender: TObject);
begin
//Create the form on the main MDI form and disable selected link
in the main menu.
  Application.CreateForm(Tpharmacydesc, pharmacydesc);
  phar.show;
  PharmacyDescription1.Enabled := False;
end;
```

```
Procedure TPharmacydesc.FormClose(Sender:TObject;varAction:
TCloseAction);
begin
if MessageDlg('Are you sure to Close Pharmacy Description
page?', mtConfirmation,[mbYes, mbNo],0) = mrYes Then
     begin
     table1.Destroy;
     table2.Destroy;
     //enabling the closed forms link on the main menu
     pharmacy.PharmacyDescription1.Enabled := true;
     Action := caFree;
     end
     else
     Action := caNone;
end;
```

HINT: For the next sections we will not explain same things to you as we explained.

3.3 Depot Description Module

This section for describe Medicine Depots Name ,Address ,Author name and surname for future Waybill stock entry .This section similar to Pharmacy Description Module. The main difference is in this module we can enter to many depot descriptions to the database and modify them .This modules has some extra button description for find the Depots continuously. The following figure 3.3.1 shows you the Depot Descriptions screen layout.

This section we want to explain the print statement. For printing the descriptions we use an extra component called TPrintDialog. The following codes explains how the print will work correctly.

```
procedure Tdepotinfo.PrintClick(Sender: TObject);
var
      H,i,j,lineno,line2no:integer;
begin
if printdialog1.Execute then
   begin
   //declare the font type and size
   printer.begindoc;
   printer.canvas.Font.Name := 'MS sanf serif';
   printer.canvas.Font.Size := 10;
   lineno := 360;
   line2no := 350;
   //Print the First descriptions to last descriptions
   table1.First;
   //Print the -pharmacy depot information- word x:150, y:250
   printer.Canvas.TextOut(150,     250,     'PHARMACY     DEPOT
INFORMATIONS');
   printer.Canvas.MoveTo(150, 290);
   printer.Canvas.Lineto(2000, 290);
   //Print all data belong to depot table
   While not Table1.Eof do
     begin
     IF lineno >= 2600 then
     Begin
     printer.NewPage;
     lineno:=250;
     line2no:=260;
     end;
     Printer.Canvas.MoveTo(160, line2no);
     Printer.Canvas.Lineto(2000, Line2no);
     printer.Canvas.TextOut(160, lineno, 'Depot Name :');
     Printer.Canvas.TextOut(500, lineno, dbedit2.text);
     Printer.Canvas.TextOut(1350, lineno, 'Tax Department');
     Printer.Canvas.TextOut(1850, lineno, dbedit4.Text);
     lineno:=Lineno + 60;
     Printer.Canvas.TextOut(160, lineno, 'Depot Code :');
     Printer.Canvas.TextOut(500, lineno, dbedit1.text);
```

```
    Printer.Canvas.TextOut(1350, lineno, 'Tax no');
    Printer.Canvas.TextOut(1850, lineno, dbedit5.Text);
    lineno:=lineno + 60;
    Printer.Canvas.TextOut(160, lineno, 'Author :');
    Printer.Canvas.TextOut(500, lineno, dbedit3.text);
    lineno:=lineno + 60;
    Printer.Canvas.TextOut(160, lineno, 'Address :');
    Printer.Canvas.TextOut(500, lineno, dbedit9.Text);
    lineno:=lineno + 60;
    Printer.Canvas.TextOut(160, lineno, 'City :');
    Printer.Canvas.TextOut(500, lineno, dbedit6.Text);
    Printer.Canvas.TextOut(1350, lineno, 'Phone1 :');
    Printer.Canvas.TextOut(1850, lineno, dbedit10.Text);
    lineno:=lineno + 60;
    Printer.Canvas.TextOut(160, lineno, 'State :');
    Printer.Canvas.TextOut(500, lineno, dbedit7.Text);
    Printer.Canvas.TextOut(1350, lineno, 'Phone2 :');
    Printer.Canvas.TextOut(1850, lineno, dbedit11.Text);
    lineno:=lineno + 60;
    Printer.Canvas.TextOut(160, lineno, 'Postal Code :');
    Printer.Canvas.TextOut(500, lineno, dbedit8.Text);
    Printer.Canvas.TextOut(1350, lineno, 'Fax :');
    Printer.Canvas.TextOut(1850, lineno, dbedit12.Text);
    lineno := lineno + 80;
    line2no := line2no + 440;
    Table1.Next;
    end;
    printer.enddoc
 end;
end;
```

This codes concordant only for A4 paper. But also this is the beta version of program. In most professional Programs like Hospital Automation program has the Text Editors . This Text editors can be easily re modify the default data and give some flexible add – ons to user.

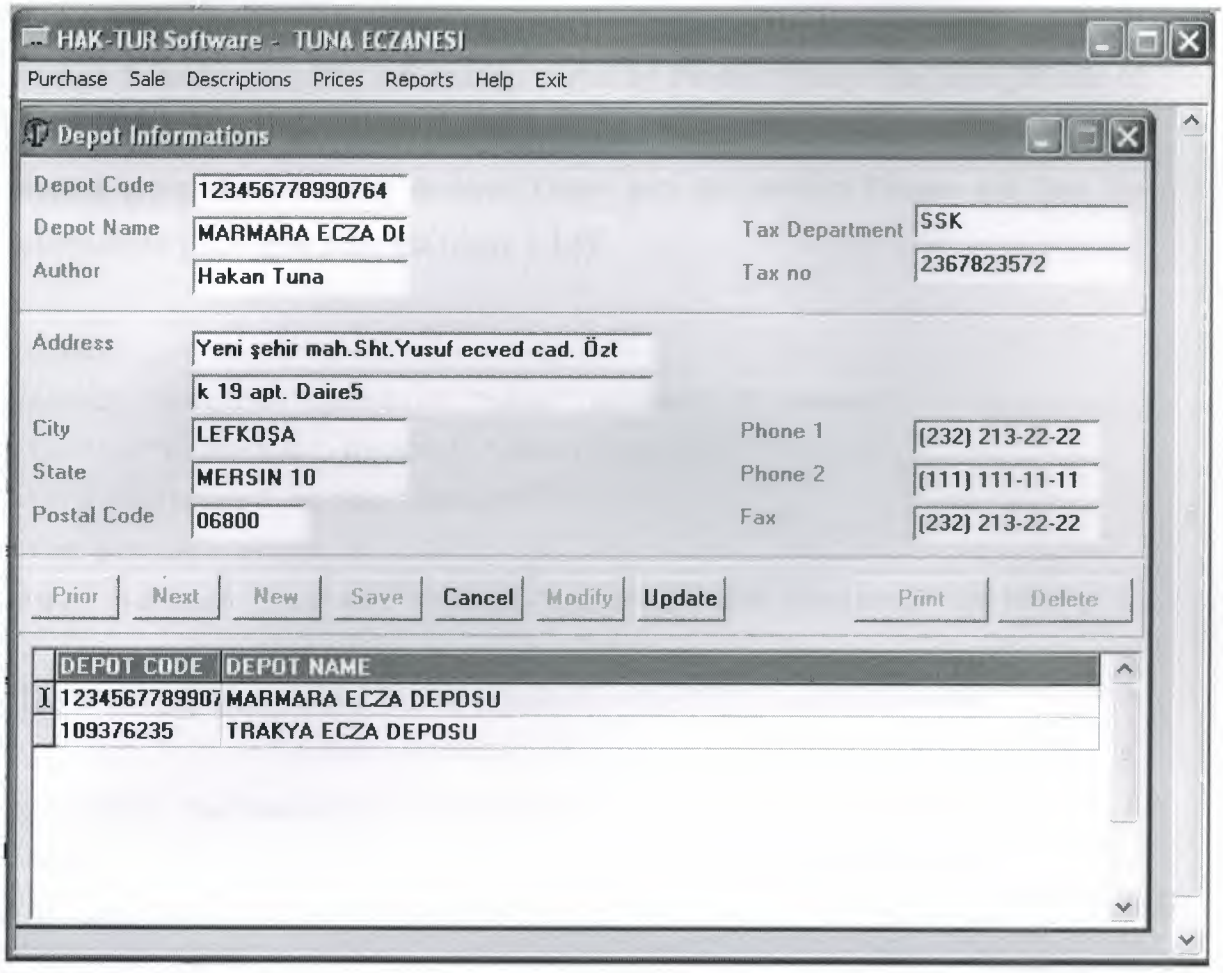

Figure 3.3.1 Depot Descriptions Screen Layout while Executed.

3.4 Product Description Module

This section is the most difficult part of the Pharmacy Program before Waybill Entry Section.(the waybill entry section source code will be given in Appendix A). In this section we don't use the one table of data .We are using 12 tables For declare for one product . Because of the user will not give the similar information of products for (for exanmple Firm Descriptions, Product Group Descriptions, Farmosotical Group Descriptions etc..) each product entry. Pharmacist can easily select this sections from TDBLookUpComboBox components on the form. This components directly connected to the Product table, but it looks up the other tables for fast entry in query or table.

This section consist of 3 parts.

1. Product information
2. Product Minimum – Maximum values descriptions for each month (Optional).
3. Product Prospectus

At the same time in product information menu we use SQL Query for while we select product from the grid .The information sorted by Product Group .On the right side of the form we have a TList component for select the product groups. When we select product group the following declared Query gets the product Params and lists the informations in the grid .(See the figure 3.4.1)

```
select
product."barcode",product."name",product."inamount",
product."progroup",product."shelf",product."price",
stock."barcode",stock."amount"
from product,stock
where product."barcode"=stock."barcode" and progroup=:progroup
```

The following source codes will be explain how the SQL commands works.

```
procedure Tproductinfo.FormCreate(Sender: TObject);
begin
 // get the list of continents
  Query2.Open;
  while not Query2.EOF do
  begin
    ListBox1.Items.Add (Query2.Fields [0].AsString);
    Query2.Next;
  end;
  ListBox1.ItemIndex := 0;
  .....
```

The following command is explain the work of command if select the product group the source code will be automatically adds the data to the DBgrid.

```
procedure Tproductinfo.ListBox1Click(Sender: TObject);
begin
 Query1.Close;
 Query1.Params[0].Value :=
 ListBox1.Items [Listbox1.ItemIndex];
 Query1.Open;
end;
```

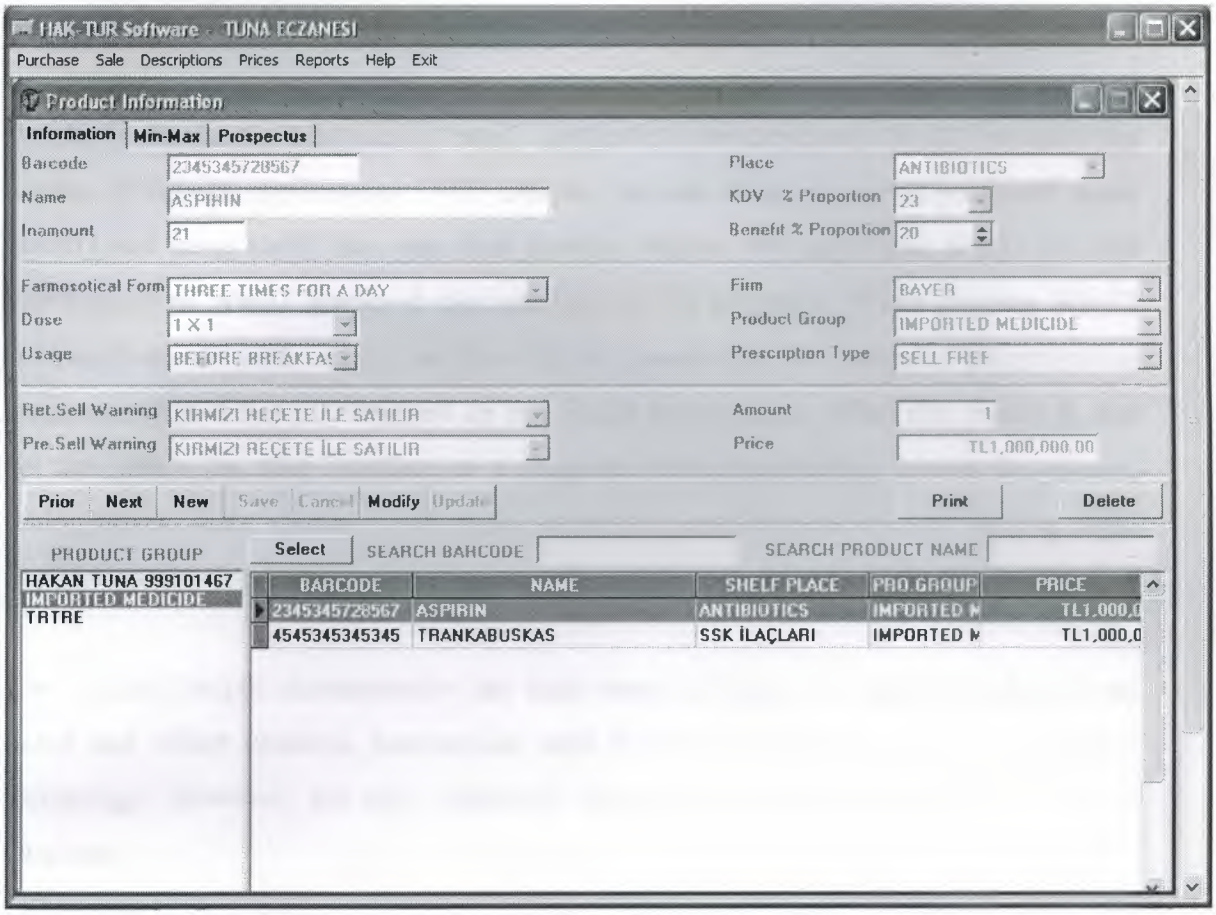

Figure 3.4.1: Product Descriptions Screen Layout while Executed.

In Min-Max page of the program module gets the data for each month's ,minimum and maximum values, and stores them in stock table.

In Prospectus page we Use DBMemo for enter the prospectus of product.The important point is here, the paradox just stores 255 characters in it but we declare this as memo and paradox creates extra file for storing this information in the database.

Summary and Conclusion

Delphi is Borland's best-selling rapid application development (RAD) product for writing Windows applications. With Delphi, we can write Windows programs more quickly and more easily than was ever possible before. We can create Win32 console applications or Win32 graphical user interface (GUI) programs. When creating Win32 GUI applications with Delphi, we have all the power of a true compiled programming language (Object Pascal) wrapped up in a RAD environment. What this means is that we can create the user interface to a program (the user interface means the menus, dialog boxes, main window, and so on) using drag-and-drop techniques for true rapid application development.

The original Delphi characteristics are high speed collector, the approach about form based and object oriented, harmonious with Windows programming and component technology. However, the very important element is all others basic Object Pascal language.

Let's we can talk about Borland Database Engine (BDE). BDE is coming with Paradox, at the time when Delphi don't exist and BDE developed by Borland to supported many SQL service units and Borland's own local databases.

Using BDE's specific advantages are this technology whole with Delphi, elements are fully and very good documented and the only logical analyze way to arrive local files like Paradox and dBase tables.

Delphi introduces the CLX library alongside the traditional VCL library. There are certainly many differences, even in the use of the RTL and code library classes, between developing programs specifically for Windows or with a crossplatform attitude, but the user interface portion is where differences are most striking. The visual portion of VCL is a wrapper of the Window API. It includes wrappers of the native Windows controls (like buttons and edit boxes).

Finally in our Graduation Project, we search thoroughly how Pharmacy works? What is the important structures while we design and develop a Pharmacy Automation Program. We did great research about management of Pharmacy .The important point of the software engineering is , understand the targets for reach the maximum efficiency when the program ends.

References

Ray Lishner,"***Delphi in a Neutshell***", O'Reilly, 2000

Marco Cantu,"***Mastering Delphi 6***", Sybex, 2002

Ezal Balkan,"***Borland Delphi 7.0***", Seckin, 2003

Henry F. Korth, Abraham Silberschatz, "***Database System Concepts***", McGrawhill,1990.

David M. Kroenke,"***Database Concepts***",Prentice Hall, 2002.

John J. Patrick, "***SQL Fundementals",*** Prentice Hall, 2002.

Oya Kalıpsız, "***Computer Database Systems***", Air Combat School, 1993.

http://www.about.delphi.com

http://www.delphiturk.com

http://www.borland.com/delphi/index.html

Appendices
Appendix A

Waybill Entrance Module Source Code,

```
unit wbill;

interface

uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics,
Controls, Forms, Dialogs, ExtCtrls,Strutils , Grids, DBGrids,
StdCtrls, Mask, DBCtrls, ComCtrls, Buttons, DB, DBTables,
dbcgrids, Spin;

type
  Twaybill = class(TForm)
    Panel1: TPanel;
    Panel2: TPanel;
    DBGrid1: TDBGrid;
    Label1: TLabel;
    Label2: TLabel;
    Label3: TLabel;
    depotname: TDBLookupComboBox;
    date1: TDateTimePicker;
    Label4: TLabel;
    Label5: TLabel;
    Label6: TLabel;
    date2: TDateTimePicker;
    invno: TDBEdit;
    explanation: TDBEdit;
    new: TSpeedButton;
    modify: TSpeedButton;
    save: TSpeedButton;
    cancel: TSpeedButton;
    wnew: TSpeedButton;
    wdelete: TSpeedButton;
    wsave: TSpeedButton;
```

```
wcancel: TSpeedButton;
wselect: TSpeedButton;
wexit: TSpeedButton;
delete: TSpeedButton;
Panel3: TPanel;
Table1: TTable;
Table2: TTable;
DataSource1: TDataSource;
DataSource2: TDataSource;
Table1Name: TStringField;
Table1Inv_date: TDateField;
Table1Inv_no: TStringField;
Table1Way_no: TStringField;
Table1Total: TCurrencyField;
Table1Explanation: TStringField;
Label7: TLabel;
Table3: TTable;
DataSource4: TDataSource;
Table3Way_no: TStringField;
Table3Barcode: TStringField;
Table3Amount: TFloatField;
Table3Benefit: TFloatField;
Table3Price: TCurrencyField;
Table3Discount: TFloatField;
Table3Shelf: TStringField;
Table3Kdv: TFloatField;
Table3Proextra: TFloatField;
DataSource3: TDataSource;
Table4: TTable;
Table3Name: TStringField;
amount: TSpinEdit;
proextra: TSpinEdit;
tax: TSpinEdit;
benefit: TSpinEdit;
discount: TSpinEdit;
price: TDBEdit;
total: TDBEdit;
```

```
    Label8: TLabel;
    Label9: TLabel;
    Label10: TLabel;
    Label11: TLabel;
    Label12: TLabel;
    Label13: TLabel;
    Label14: TLabel;
    shelf: TDBLookupComboBox;
    Label15: TLabel;
    Table5: TTable;
    DataSource5: TDataSource;
    Table5Shelf: TStringField;
    Table6: TTable;
    Table7: TTable;
    DataSource6: TDataSource;
    DataSource7: TDataSource;
    Table6Amount: TFloatField;
    Table6Barcode: TStringField;
    Table7Proname: TStringField;
    Table7Explanation: TStringField;
    Table7Time: TTimeField;
    Table7Amount: TFloatField;
    Table7Place: TStringField;
    Table7Price: TCurrencyField;
    Table7Total: TCurrencyField;
    Table7Inout: TStringField;
    update: TSpeedButton;
    DataSource8: TDataSource;
    Table8: TTable;
    Table8Barcode: TStringField;
    Table8Amount: TFloatField;
    Table8Jmin: TStringField;
    Table8Jmax: TStringField;
    Table8Fmin: TStringField;
    Table8Fmax: TStringField;
    Table8Mmin: TStringField;
    Table8Mmax: TStringField;
```

```
Table8Amin: TStringField;
Table8Amax: TStringField;
Table8Maymin: TStringField;
Table8Maymax: TStringField;
Table8Junmin: TStringField;
Table8Junmax: TStringField;
Table8Julmin: TStringField;
Table8Julmax: TStringField;
Table8Augmin: TStringField;
Table8Augmax: TStringField;
Table8Smin: TStringField;
Table8Smax: TStringField;
Table8Omin: TStringField;
Table8Omax: TStringField;
Table8Nmin: TStringField;
Table8Nmax: TStringField;
Table8Dmin: TStringField;
Table8Dmax: TStringField;
Table9: TTable;
DataSource9: TDataSource;
Table9Proname: TStringField;
Table9Explanation: TStringField;
Table9Amount: TFloatField;
Table9Time: TTimeField;
Table9Place: TStringField;
Table9Price: TCurrencyField;
Table9Total: TCurrencyField;
Table9Inout: TStringField;
Table7Depotname: TStringField;
Table9Depotname: TStringField;
cancel2: TSpeedButton;
Panel4: TPanel;
Label16: TLabel;
Label17: TLabel;
Label18: TLabel;
Label19: TLabel;
Label20: TLabel;
```

```
    Label21: TLabel;
    ucost: TLabel;
    usprice: TLabel;
    wttotal: TLabel;
    wdtotal: TLabel;
    witotal: TLabel;
    wtbenefit: TLabel;
    wno: TDBEdit;
    DataSource11: TDataSource;
    Query1: TQuery;
    Table3Wttotal: TCurrencyField;
    Table3Unitcost: TCurrencyField;
    Table3Wdtotal: TCurrencyField;
    Table3Wintotal: TCurrencyField;
    Table3Wtbenefit: TFloatField;
    Query1SUMOFprice: TCurrencyField;
    Query1SUMOFwttotal: TCurrencyField;
    Query1SUMOFwintotal: TCurrencyField;
    Query1SUMOFwdtotal: TCurrencyField;
    Query1way_no: TStringField;
    Label22: TLabel;
    kar: TLabel;
    Label23: TLabel;
    Label24: TLabel;
    Label25: TLabel;
    Label26: TLabel;
    Label27: TLabel;
    Label28: TLabel;
    Table3Total: TCurrencyField;
    Table9Datee: TDateField;
    Table7Datee: TDateField;
    Table1Datee: TDateField;
    Table1Timee: TTimeField;
    procedure wnewClick(Sender: TObject);
    procedure wsaveClick(Sender: TObject);
    procedure wcancelClick(Sender: TObject);
    procedure wnoChange(Sender: TObject);
```

```
    procedure newClick(Sender: TObject);
    procedure cancelClick(Sender: TObject);
    procedure saveClick(Sender: TObject);
    procedure amountChange(Sender: TObject);
    procedure proextraChange(Sender: TObject);
    procedure taxChange(Sender: TObject);
    procedure benefitChange(Sender: TObject);
    procedure shelfClick(Sender: TObject);
    procedure discountChange(Sender: TObject);
    procedure deleteClick(Sender: TObject);
    procedure modifyClick(Sender: TObject);
    procedure updateClick(Sender: TObject);
    procedure FormCreate(Sender: TObject);
    procedure FormClose(Sender: TObject; var Action:
TCloseAction);
    procedure DBGrid1CellClick(Column: TColumn);
    procedure cancel2Click(Sender: TObject);
    procedure Button2Click(Sender: TObject);
    procedure depotnameCloseUp(Sender: TObject);
    procedure wexitClick(Sender: TObject);
  private
    { Private declarations }
  public
    procedure calculation();
  end;

var
  waybill: Twaybill;
  amounts : string;
  barcode : string;
  a, b, c, x, y, z, ucost1, uspricel, wttotal1, wdtotal1,
  witotal1, wtbenefit1 : real;
implementation

{$R *.dfm}
uses probak, pharma;
```

```
procedure Twaybill.wnewClick(Sender: TObject);
begin
    date1.Date := date;
    date2.date := date;
    date1.time := time;
    date2.time := time;
    table1.Close;
    table1.Open;
    table1.insert;
    wnew.Enabled := false;
    wdelete.Enabled := false;
    wsave.Enabled := false;
    wcancel.Enabled := true;
    wselect.Enabled := false;
    wexit.Enabled := true;
    depotname.Enabled := true;
    date2.Enabled := true;
    date1.Enabled := true;
    wno.Enabled := false;
    invno.Enabled := true;
    explanation.Enabled := true;
end;

procedure Twaybill.wsaveClick(Sender: TObject);
begin
    table1Datee.Text := datetostr(date1.date);
    table1Inv_date.Text := datetostr(date2.date);
    table1Timee.Text := timetostr(date1.Time);
    table1.post;
    wnew.Enabled := true;
    wdelete.Enabled := false;
    wsave.Enabled := false;
    wcancel.Enabled := false;
    wselect.Enabled := true;
    wexit.Enabled := true;
    depotname.Enabled := false;
    date2.Enabled := false;
```

```
    date1.Enabled := false;
    wno.Enabled := false;
    invno.Enabled := false;
    explanation.Enabled := false;
end;

procedure Twaybill.wcancelClick(Sender: TObject);
begin
    table1.Cancel;
    table1.Close;
    wnew.Enabled := true;
    wdelete.Enabled := false;
    wsave.Enabled := false;
    wcancel.Enabled := false;
    wselect.Enabled := true;
    wexit.Enabled := true;
    depotname.Enabled := false;
    date2.Enabled := false;
    date1.Enabled := false;
    wno.Enabled := false;
    invno.Enabled := false;
    explanation.Enabled := false;
end;

procedure Twaybill.wnoChange(Sender: TObject);
var
T : boolean;
le , i , count : integer;
lastchar : string;
begin
  wno.Text := trim(wno.Text);
  le := length(wno.Text);
  count := 0;
  for i := 1 to le do
  begin
    lastchar := midstr(wno.text,i,1);
    if ((lastchar = '0') or (lastchar = '1') or
```

```
      (lastchar = '2') or (lastchar = '3') or (lastchar =
'4') or
      (lastchar = '5') or (lastchar = '6') or (lastchar =
'7') or
      (lastchar = '8') or (lastchar = '9')) then
      begin
        count:=count+1;
        //Those lines controls the waybillin table for
overflow
        table4.Close;
        table4.Open;
        T := False;
        Table4.SetKey;
        Table4.IndexFields[0].AsString := wno.Text;
        T := Table4.GotoKey;
        if not T then
         begin
          new.Enabled := true;
          modify.Enabled := false;
          delete.Enabled := false;
          save.Enabled := false;
          update.Enabled := false;
          cancel.Enabled := false;
          cancel2.Enabled := false;
          DBgrid1.Enabled := false;
         end
        else
        begin
          new.Enabled := false;
          modify.Enabled := false;
          delete.Enabled := false;
          save.Enabled := false;
          cancel.Enabled := false;
          cancel2.Enabled := false;
          update.Enabled := false;
          DBgrid1.Enabled := false;
        end;
```

```
      end
      else
      begin
        wno.text := leftstr(wno.text,i-1)+
        midstr(wno.Text,i+1,564);
        if count > 0  then
        begin
          new.Enabled := true;
          modify.Enabled := false;
          delete.Enabled := false;
          save.Enabled := false;
          cancel.Enabled := false;
          cancel2.Enabled := false;
          update.Enabled := false;
          DBgrid1.Enabled := false;
        end;

    end;
  end;
end;

procedure Twaybill.newClick(Sender: TObject);
begin
  table3.close;
  table3.Filter := 'way_no =' + wno.Text;
  table3.Open;
  wnew.Enabled := false;
  wdelete.Enabled := false;
  wsave.Enabled := false;
  wcancel.Enabled := false;
  wselect.Enabled := false;
  wexit.Enabled := false;
  depotname.Enabled := false;
  date2.Enabled := false;
  date1.Enabled := false;
  wno.Enabled := false;
```

```
  invno.Enabled := false;
  explanation.Enabled := false;
  new.Enabled := false;
  modify.Enabled := false;
  save.Enabled := false;
  cancel.Enabled := false;
  dbgrid1.Enabled := false;
  Application.CreateForm(Tproselect, proselect);
  proselect.show;
end;

procedure Twaybill.cancelClick(Sender: TObject);
begin
  table3.Edit;
  amount.Text := '0';
  proextra.Text := '0';
  tax.Text := '0';
  benefit.Text := '0';
  discount.Text := '0';
  table3.cancel;
  new.Enabled := true;
  modify.Enabled := false;
  delete.Enabled := true;
  save.Enabled := false;
  cancel.Enabled := false;
  cancel2.Enabled := false;
  if table3.IndexFieldCount <> 0 then
  begin
  dbgrid1.Enabled := true;
  end
  else
  dbgrid1.Enabled := false;
  update.Enabled := false;
  amount.Enabled := false;
  proextra.Enabled := false;
  tax.Enabled := false;
  benefit.Enabled := false;
```

```
  shelf.Enabled := false;
  price.Enabled := false;
  discount.Enabled := false;
  total.Enabled := false;
  if table3.IndexFieldCount <> 0 then
  begin
   modify.Enabled := true;
  end
  else
  begin
   modify.Enabled := false;
  end;
end;

procedure Twaybill.saveClick(Sender: TObject);
var
T ,T1 ,T2: Boolean;
times ,dates : string;
ben : string;
begin
  if strtoint(tax.text) < 1 then
  begin
    showmessage('You must to give tax percentage');
  end
  else
  if strtoint(benefit.Text) < 1 then
  begin
    showmessage('You must to enter benefit percentage');
  end
  else
  if strtofloat(table3price.Text) < 1 then
  begin
    showmessage('You must to enter product price');
  end
  else
  begin
  calculation;
```

```
times := timetostr(date1.Time);
dates := datetostr(date1.Date);
//adding product amount with product amount
table6.Close;
table6.Open;
T1 := False;
Table6.SetKey;
Table6.indexFields[0].AsString := table3barcode.text;
T1 := Table6.GotoKey;
if T1 then
begin
  table6.Edit;
  table6amount.Text :=
inttostr(strtoint(table6amount.Text)+
  strtoint(table3amount.Text));
  table6.post;
end;
  table3wttotal.Text := floattostr(wttotal1);
  table3wdtotal.Text := floattostr(wdtotal1);
  table3wintotal.Text := floattostr(witotal1);
  table3unitcost.Text := floattostr(ucost1);
  table3.Post;
  //stockmovement update
  table7.Close;
  table7.Open;
  T2 := False;
  Table7.SetKey;
  Table7.IndexFields[0].AsString := table3name.Text;
  table7.IndexFields[1].AsString := table1name.Text;
  table7.IndexFields[2].AsString := dates;
  table7.IndexFields[3].AsString := Times;
  table7.IndexFields[4].AsString := 'waybill in';
  table7.IndexFields[5].AsString := table3amount.Text;
  table7.IndexFields[6].AsString := table3shelf.Text;
  table7.IndexFields[7].AsString := table3price.Text;
  table7.IndexFields[8].AsString := table3wintotal.Text;
```

```
    T2 := Table7.GotoKey;
    if not T2 then
    begin
      table7.Insert;
      table7proname.text := table3name.Text;
      table7depotname.Text := table1name.Text;
      table7explanation.text := 'waybill in';
      table7Datee.text := dates;
      table7Time.Text := times;
      table7amount.Text := table3amount.Text;
      table7place.Text := table3shelf.Text;
      table7price.Text := table3price.Text;
      table7total.Text := table3wintotal.Text;
      table7inout.Text := 'in';
      table7.Post;
    end;
      //filter the calculations
      query1.close;
      query1.Filter := 'way_no =' + wno.Text;
      query1.Open;
      wttotal.Caption := query1SUMOFwttotal.text;
      wdtotal.Caption := query1SUMOFwdtotal.text;
      witotal.Caption := query1SUMOFwintotal.text;
      //benefit calculation
      ben := floattostr(strtofloat(table3price.Text) *
strtofloat(amount.text)/strtofloat(query1SUMOFwintotal.Text)-
1);
      ben := midstr(ben,3,2);
      wtbenefit.Caption := ben;
      kar.Caption := floattostr(strtofloat(table3price.Text) *
strtofloat(amount.text)-
strtofloat(query1SUMOFwintotal.Text));
      //Those lines controls the waybillin table for overflow
      T := False;
      table4.Close;
      table4.Open;
      Table4.SetKey;
```

```
Table4.Fields[0].AsString := wno.Text;
T := Table4.GotoKey;
if not T then
begin
//if not found the waybill no insert the value;

   table1Datee.Text := dates;
   table1Inv_date.Text := datetostr(date2.date);
   table1Timee.Text := times;
   table1way_no.Text := wno.Text;
   table1total.Text := query1SUMOFwintotal.text;
   table1.post;
end
else
begin
//if found the waybill no edit the value
   table1.Edit;
   table1Datee.Text := dates;
   table1Inv_date.Text := datetostr(date2.date);
   table1Timee.Text := times;
   table1way_no.Text := wno.Text;
   table1total.Text := query1SUMOFwintotal.text;
   table1.post;
end;
wnew.Enabled := true;
wsave.Enabled := false;
wcancel.Enabled := false;
wselect.Enabled := true;
wexit.Enabled := true;
new.Enabled := true;
modify.Enabled := true;
save.Enabled := false;
//delete.Enabled := true;
cancel.Enabled := false;
cancel2.Enabled := false;
dbgrid1.Enabled := true;
amount.Enabled := false;
```

```
    proextra.Enabled := false;
    tax.Enabled := false;
    benefit.Enabled := false;
    shelf.Enabled := false;
    price.Enabled := false;
    price.Enabled := false;
    discount.Enabled := false;
    total.Enabled := false;
 end;
end;

procedure Twaybill.amountChange(Sender: TObject);
begin
 table3amount.text := amount.text;
end;

procedure Twaybill.proextraChange(Sender: TObject);
begin
 table3proextra.text := proextra.text;
end;

procedure Twaybill.taxChange(Sender: TObject);
begin
 table3kdv.text := tax.text;
end;

procedure Twaybill.benefitChange(Sender: TObject);
begin
 table3benefit.text := benefit.text;
end;

procedure Twaybill.shelfClick(Sender: TObject);
begin
 table3shelf.text := shelf.text;
end;

procedure Twaybill.discountChange(Sender: TObject);
```

```
begin
 table3discount.text := discount.text;
end;


procedure Twaybill.deleteClick(Sender: TObject);
var
T1,T2 : boolean;
begin
  table3.Edit;
  table8.Close;
  table8.Open;
  T1 := False;
  Table8.SetKey;
  Table8.indexFields[0].AsString := table3barcode.Text;
  T1 := Table8.GotoKey;
  if T1 then
  begin
  //delete stock
    table8.Edit;
    table8amount.Text :=
inttostr(strtoint(table8amount.Text)-
strtoint(table3amount.text));
    table8.Post;
  end;
  //update stockmov table
  table9.Close;
  table9.Open;
  T2 := False;
  Table9.SetKey;
  Table9.IndexFields[0].AsString := table3name.Text;
  table9.IndexFields[1].AsString := table1name.Text;
  table9.IndexFields[2].AsString := table1datee.Text;
  table9.IndexFields[3].AsString := table1timee.Text;
  table9.IndexFields[4].AsString := 'waybill in';
  table9.IndexFields[5].AsString := table3amount.Text;
```

```
  table9.IndexFields[6].AsString := table3shelf.Text;
  table9.IndexFields[7].AsString := table3price.Text;
  table9.IndexFields[8].AsString := table3total.Text;
  T2 := Table9.GotoKey;
  if T2 then
  begin
  //delete stock movement
    table9.Delete;
  end
  else
  begin
    showmessage('You did not update the'+
    'selected data');
  end;
  query1.close;
  query1.Filter := 'way_no =' + wno.Text;
  query1.Open;
  wttotal.Caption := query1SUMOFwttotal.text;
  wdtotal.Caption := query1SUMOFwdtotal.text;
  witotal.Caption := query1SUMOFwintotal.text;
  wtbenefit.Caption :=
floattostr(strtofloat(query1SUMOFprice.Te xt)
/strtofloat(query1SUMOFwintotal.Text)-1);
  query1.Refresh;
  amount.Text := '0';
  proextra.Text := '0';
  tax.Text := '0';
  benefit.Text := '0';
  discount.Text := '0';
  table3.Delete;
  new.Enabled := true;
  modify.Enabled := false;
  delete.Enabled := false;
  save.Enabled := false;
  cancel.Enabled := false;
  cancel2.Enabled := false;
  dbgrid1.Enabled := true;
```

```
  amount.Enabled := false;
  proextra.Enabled := false;
  tax.Enabled := false;
  benefit.Enabled := false;
  shelf.Enabled := false;
  price.Enabled := false;
  discount.Enabled := false;
  total.Enabled := false;
  update.Enabled := false;
end;

procedure Twaybill.modifyClick(Sender: TObject);
begin
  amounts := table3amount.text;
  barcode := table3barcode.Text;
  table3.Edit;
  amount.Enabled := true;
  wnew.enabled := false;
  wdelete.Enabled := false;
  wsave.Enabled := false;
  wcancel.Enabled := false;
  wselect.Enabled := false;
  wexit.Enabled := true;
  proextra.Enabled := true;
  tax.enabled := true;
  benefit.Enabled := true;
  shelf.Enabled := true;
  price.Enabled := true;
  discount.Enabled := true;
  total.Enabled := true;
  update.Enabled := true;
  modify.Enabled := false;
  new.Enabled := false;
  delete.Enabled := false;
  save.Enabled := false;
  cancel.Enabled := true;
  cancel2.Enabled := false;
```

```
  dbgrid1.Enabled := false;
end;

procedure Twaybill.updateClick(Sender: TObject);
var
T2 ,T1: boolean;
ben : string;
begin
 if strtoint(tax.text) < 1 then
  begin
    showmessage('You must to give tax percentage');
  end
  else
  if strtoint(benefit.Text) < 1 then
  begin
    showmessage('You must to enter benefit percentage');
  end
  else
  if strtofloat(table3price.Text) < 1 then
  begin
    showmessage('You must to enter product price');
  end
  else
  begin
  calculation;
  dbgrid1.Enabled := true;
  //update wbillno table
  table3wttotal.Text := floattostr(wttotal1);
  table3wdtotal.Text := floattostr(wdtotal1);
  table3wintotal.Text := floattostr(witotal1);
  table3unitcost.Text := floattostr(ucost1);
  table3.Post;
  table3.Refresh;
  //update stock table
  table8.Close;
  table8.Open;
  T1 := False;
```

```
  Table8.SetKey;
  Table8.indexFields[0].AsString := barcode;
  T1 := Table8.GotoKey;
  if T1 then
  begin
  //update stock
    table8.Edit;
    table8amount.Text :=
inttostr(strtoint(table8amount.Text)-strtoint(amounts));
    table8amount.Text :=
inttostr(strtoint(table8amount.Text)+strtoint(amount.Text));
    table8.Post;
  end;
  //update stockmov table
  table9.Close;
  table9.Open;
  T2 := False;
  Table9.SetKey;
  Table9.IndexFields[0].AsString := table3name.Text;
  table9.IndexFields[1].AsString := table1name.Text;
  table9.IndexFields[2].AsString := table1datee.Text;
  table9.IndexFields[3].AsString := table1timee.Text;
  table9.IndexFields[4].AsString := 'waybill in';
  table9.IndexFields[5].AsString := table3amount.Text;
  table9.IndexFields[6].AsString := table3shelf.Text;
  table9.IndexFields[7].AsString := table3price.Text;
  table9.IndexFields[8].AsString := table3wintotal.Text;
  T2 := Table9.GotoKey;
  if not T2 then
  begin
    table9.edit;
    table9proname.text := table3name.Text;
    table9depotname.Text := table1name.Text;
    table9explanation.text := 'waybill in';
    table9Datee.text := table1datee.Text;
    table9Time.Text := table1timee.Text;
    table9amount.Text := table3amount.Text;
```

```
    table9place.Text := table3shelf.Text;
    table9price.Text := table3price.Text;
    table9total.Text := table3wintotal.Text;
    table9inout.Text := 'in';
    table9.Post;
  end
  else
  begin
    showmessage('You did not update the'+
    'selected data');
  end;
    query1.close;
    query1.Filter := 'way_no =' + wno.Text;
    query1.Open;
    wttotal.Caption := query1SUMOFwttotal.text;
    wdtotal.Caption := query1SUMOFwdtotal.text;
    witotal.Caption := query1SUMOFwintotal.text;
    //benefit calculation
    ben := floattostr(strtofloat(table3price.Text) *
strtofloat(amount.text)/strtofloat(query1SUMOFwintotal.Text)-
1);
    ben := midstr(ben,3,2);
    wtbenefit.Caption := ben;
    kar.Caption := floattostr(strtofloat(table3price.Text) *
strtofloat(amount.text)-
strtofloat(query1SUMOFwintotal.Text));
    //button combinations for error control
    table1.Edit;
    table1total.Text := query1SUMOFwintotal.Text;
    table1.post;
    wnew.Enabled := true;
    update.Enabled := false;
    wsave.Enabled := false;
    wcancel.Enabled := false;
    wselect.Enabled := true;
    wexit.Enabled := true;
    new.Enabled := true;
```

```
    modify.Enabled := true;
    save.Enabled := false;
    cancel.Enabled := false;
    cancel2.Enabled := false;
    dbgrid1.Enabled := true;
    amount.Enabled := false;
    proextra.Enabled := false;
    tax.Enabled := false;
    benefit.Enabled := false;
    shelf.Enabled := false;
    price.Enabled := false;
    price.Enabled := false;
    discount.Enabled := false;
    total.Enabled := false;
end;
end;

procedure Twaybill.FormCreate(Sender: TObject);
begin
 date1.Date := date;
 date2.date := date;
 date1.time := time;
 date2.time := time;
end;
procedure Twaybill.FormClose(Sender: TObject; var Action:
TCloseAction);
begin
 table1.Destroy;
 table2.Destroy;
 table3.Destroy;
 table4.Destroy;
 table5.Destroy;
 table6.Destroy;
 table7.Destroy;
 table8.Destroy;
 table9.Destroy;
 query1.Close;
```

```
 pharmacy.waybill1.enabled := true;
 action := cafree;
end;
procedure Twaybill.DBGrid1CellClick(Column: TColumn);
var
t1 : boolean;
begin
  query1.close;
  query1.Filter := 'way_no =' + wno.Text;
  query1.Open;
  wttotal.Caption := query1SUMOFwttotal.text;
  wdtotal.Caption := query1SUMOFwdtotal.text;
  witotal.Caption := query1SUMOFwintotal.text;
  ucost.Caption := table3unitcost.Text;
  usprice.Caption := table3price.Text;
  table3.Edit;
  amount.Text := table3amount.Text;
  proextra.Text := table3proextra.Text;
  tax.Text := table3kdv.Text;
  benefit.Text := table3benefit.Text;
  discount.Text := table3discount.Text;
  table3.Cancel;
  table3.Refresh;
  //for controlling the data for delete button
  table6.Close;
  table6.Open;
  T1 := False;
  Table6.SetKey;
  Table6.indexFields[0].AsString := table3barcode.text;
  T1 := Table6.GotoKey;
  if T1 then
     delete.Enabled := true
  else
     delete.Enabled := false;
end;
procedure Twaybill.cancel2Click(Sender: TObject);
begin
```

```
    table3.cancel;
    new.Enabled := true;
    modify.Enabled := false;
    delete.Enabled := true;
    save.Enabled := false;
    cancel.Enabled := false;
    cancel2.Enabled := false;
    if table3.IndexFieldCount <> 0 then
    begin
    dbgrid1.Enabled := true;
    end
    else
    wnew.enabled := true;;
    wdelete.Enabled := false;
    wsave.Enabled := false;
    wcancel.Enabled := false;
    wselect.Enabled := true;
    wexit.Enabled := true;
    dbgrid1.Enabled := false;
    update.Enabled := false;
    amount.Enabled := false;
    proextra.Enabled := false;
    tax.Enabled := false;
    benefit.Enabled := false;
    shelf.Enabled := false;
    price.Enabled := false;
    discount.Enabled := false;
    total.Enabled := false;
    if table3.IndexFieldCount <> 0 then
    begin
    modify.Enabled := true;
    end
    else
    begin
    modify.Enabled := false;
    end;
end;
```

```
procedure Twaybill.calculation();
begin
  a := strtofloat(table3amount.Text);
  b := strtofloat(table3proextra.Text);
  c := strtofloat(table3kdv.Text);
  x := strtofloat(table3benefit.Text);
  y := strtofloat(table3price.Text);
  z := strtofloat(table3discount.Text);
  //to find the unit cost
  ucost1 := y - (y * x / 100);
  if x > 0 then
  begin
  //if there is any discount
    ucost1 := ucost1 - (ucost1 * z / 100);
  end;
  //unit sell price
  usprice1 := y;
  //waybill product tax total
  wttotal1 := ucost1 * a * c / 100;
  //waybill product discount total
  wdtotal1 := (y - (y * x / 100)) * a;
  wdtotal1 := wdtotal1 * z / 100;
  //waybill invoice product total
  witotal1 := ucost1 * a;
  //unit cost
  ucost.caption := floattostr(ucost1);
  //unit sell price
  usprice.Caption := floattostr(usprice1);
end;

procedure Twaybill.Button2Click(Sender: TObject);
begin
table3.Next;
end;

procedure Twaybill.depotnameCloseUp(Sender: TObject);
```

```
begin
wno.Enabled := true;
end;

procedure Twaybill.wexitClick(Sender: TObject);
begin
waybill.Close;
end;
end.
```