NEAR EAST UNIVERSITY

Faculty of Engineering

Department of Computer Engineering

HOTEL RESERVATION PROGRAM IN DELPHI PROGRAMMING LANGUAGE

Graduation Project
COM-400

Students: Nilay Zerdeci
Volkan Kamiloğlu

Supervisor: Mr. Halil Adahan

Nicosia-2004

ABSTRACT

In an otel , the Front Desk part is very important to control all the otel's functions. All the otel records are saved in the Front Desk department of an otel.Especially , in luxury and developed otels , there must be many functions to control the otel status. Like Rooms Status , Rezervations , Check-in , Check-out , Events in Otel , Reports and so on. To manage , record and control all these functions is very difficult in big otels. Therefore , the hotel rezervation programs are used.

An Otel Rezervation Program makes some general operations of an otel. And an otel rezervation program is the most important and most necessary part of an otel. This program provides to reach immediately to customer detailed information in otel , rezervations to customers and groups and agent who in otel and will come to otel , Rooms Status like full or empty rooms , clean or dirty rooms , The expences of the customers in otel and outside customers , cashier details like cashier reports , events in otel , services in otel and so on.

Nilkan Soft 1.0 is an otel rezervation program and written on Delphi Programming Language. The records are saved in tables which prepared in Paradox 7.0. Paradox 7.0 is a useful database manage program of Delphi . All the components of interfaces are Delphi's own components.

ACKNOWLEDGEMENTS

" First, We would like to thank our supervisor Mr. Halil Adahan , Dean of Engineering department Prof.Dr.Fahkrettin Memadov , and our teacher Mr.Ümit İlhan for his invaluable advice and belief in our work and ourself over the course of this Graduation Project..

Second, We would like to express our gratitute to Near East University for the scholarship that make the work possible.

Third, We thank our family(Erdoğan Zerdeci , Hasan Kamiloğlu , Ayşe Zerdeci , Z.Emel Kamiloğlu , Nehir Zerdeci , Mehmet and Burcu Kamiloğlu) for their constant encouragement and support during the preparation this project.

Finally, We would also like to thank all our friends (Özgür Berbergil , Ezgi İşgen , Bekir Şimşek , Nil Doğandemir , Ahmet Ağkuş , Burcu Tan , Üzeyir Tunç , Ayça Sağlam , Bülent Kaya , Şeyma Akdağ , Seda Kılınçel) for their advice and support."

TABLE OF CONTENTS

INTRODUCTION

In an otel , the Front Desk part is very important to control all the otel's functions. All the otel records are saved in the Front Desk department of an otel.Especially , in luxury and developed otels , there must be many functions to control the otel status. Like Rooms Status , Rezervations , Check-in , Check-out , Events in Otel , Reports and so on. To manage , record and control all these functions is very difficult in big otels. Therefore , the hotel rezervation programs are used.

An Otel Rezervation Program makes some general operations of an otel. And an otel rezervation program is the most important and most necessary part of an otel. This program provides to reach immediately to customer detailed information in otel , rezervations to customers and groups and agent who in otel and will come to otel , Rooms Status like full or empty rooms , clean or dirty rooms , The expences of the customers in otel and outside customers , cashier details like cashier reports , events in otel , services in otel and so on.

Nilkan Soft 1.0 is an otel rezervation program and written on Delphi Programming Language. The records are saved in tables which prepared in Paradox 7.0. Paradox 7.0 is a useful database manage program of Delphi . All the components of interfaces are Delphi's own components.

Chapter One describes the general Delphi's functions. Besides chapter one explain based program of Delphi , Pascal .

Chapter Two describes the visual programming in Delphi and explains the differences between other programs. And , the development of delphi programming language .

Chapter Three describes the Delphi's Database operations with examples and explains Delphi's own Database Programs.

Chapter Four describes Nilkan Soft 1.0 Hotel Rezervation Program. In this chapter Nilkan Soft is explained clearly and described also the development of the program.

Chapter-1
FOUNDATIONS

1.1 The Delphi IDE

The Delphi IDE includes large and small changes that will really improve a programmer's productivity. Among the key features are the introduction of the Object TreeView for every designer, an improved Object Inspector, extended code completion, and loadable views, including diagrams and HTML.

Most of the features are quite easy to grasp, but it's worth examining them with some care so that you can start using Delphi at its full potential right away. You can see an overall image of Delphi IDE, highlighting some of the new features, in Figure 1.1.

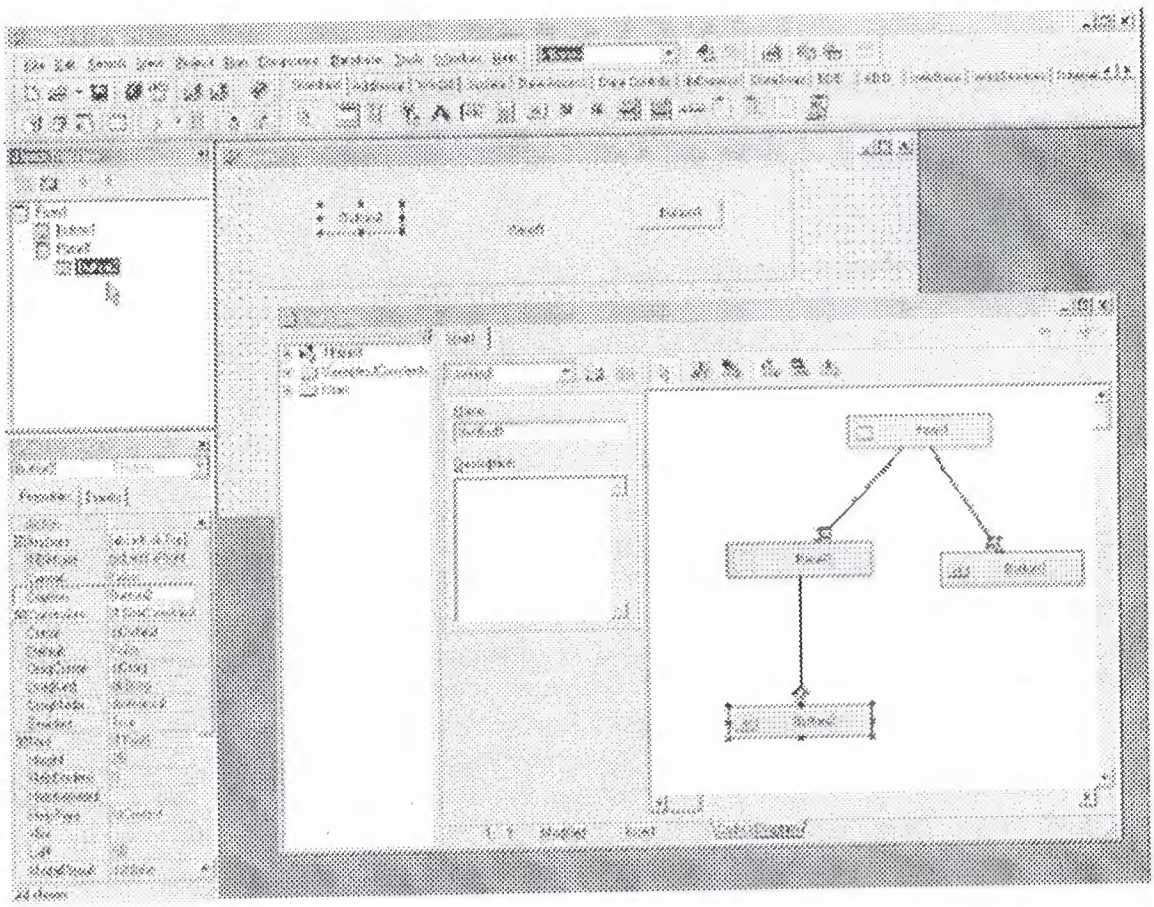


FIGURE 1.1 :The Delphi IDE: Notice the Object TreeView and the Diagram view.

1.1.1 Delphi IDE

1.1.1.1 The Object TreeView

Delphi introduced a TreeView for data modules, where you could see the relations among nonvisual components, such as datasets, fields, actions, and so on. Delphi extends the idea by providing an Object TreeView for every designer, including plain forms. The Object TreeView is placed by default above the Object Inspector; use the View . Object TreeView command in case it is hidden. 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. The TreeView is synchronized with the Object Inspector and Form Designer, so as you select an item and change the focus in any one of these three tools, the focus changes in the other two tools.

1.1.1.2 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.

The most frequently used view is the Diagram page, which was already available in Delphi data modules, although it was less powerful. Another set of views is available in Web applications, including an HTML Script view, an HTML Result preview.

1.1.1.3 An IDE for Two Libraries

Another very important change we just want to introduce here is the fact that Delphi, for the first time, allows you to use to different component libraries, VCL (Visual Components Library) and CLX (Component Library for Cross-Platform). When you create a new project, you simply choose which of the two libraries you want to use, starting with the File . New . Application command for a classic VCL-based Windows program and with the File . New . CLX Application command for a new CLX-based portable application. Creating a new project or opening an existing one, the Component Palette is rearranged to show only the controls related to the current library (although most of them are actually shared). You can use Delphi to build applications you can compile right away for Linux using Kylix. The effect of this change on the IDE is really quite large, as many things "under the hood" had to be reengineered. Only programmers using the ToolsAPI and other advanced

elements will notice all these internal differences, as they are mostly transparent to most users.

1.1.1.4 Smaller Enhancements

There are small (but still quite important)changes in the Delphi IDE. Here is a list of these changes:

- There is a new Window menu in the IDE. This menu lists the open windows, something you could obtain in the past using the Alt+0 keys. This is really very handy, as windows often end up behind others and are hard to find.

- The File menu doesn't include specific items for creating new forms or applications.These commands have been increased in number and grouped under the File . New secondary menu. The Other command of this menu opens the New Item dialog box (the Object Repository) as the File . New command did in the past.

- The Component Palette local menu has a submenu listing all of the palette pages in alphabetic order. You can use it to change the active page, particularly when it is not visible on the screen.

1.1.2 The AppBrowser Editor

The editor included with Delphi hasn't changed recently, but it has many features that many Delphi programmers don't know and use. It's worth briefly examining this tool. The Delphi editor allows you to work on several files at once, using a "notebook with tabs" metaphor, and you can also open multiple editor windows. You can jump from one page of the editor to the next by pressing Ctrl+Tab (or Shift+Ctrl+Tab to move in the opposite direction).

Several options affect the editor, located in the new Editor Properties dialog box. You have to go to the Preferences page of the Environment Options dialog box, however, to set the editor's AutoSave feature, which saves the source code files each time you run the program (preventing data loss in case the program crashes badly).A tip to remember is that using the Cut and Paste commands is not the only way to move source code. You can also select and drag words, expressions, or entire lines of code. You can also copy text instead of moving it, by pressing the Ctrl key while dragging.

1.1.2.1 The Code Explorer

The Code Explorer window, which is generally docked on the side of the editor, simply lists all of the types, variables, and routines defined in a unit, plus other units appearing in uses statements. For complex types, such as classes, the Code Explorer can list detailed information, including a list of fields, properties, and methods. All the information is updated as soon as you start typing in the editor.

You can use the Code Explorer to navigate in the editor. If you double-click one of the entries in the Code Explorer, the editor jumps to the corresponding declaration.

1.1.2.2 Browsing in the Editor

Another feature of the AppBrowser editor is *Tooltip symbol insight*. Move the mouse over a symbol in the editor, and a Tooltip will show you where the identifier is declared. This feature can be particularly important for tracking identifiers, classes, and functions within an application you are writing, and also for referring to the source code of the Visual Component Library (VCL).

1.1.2.3 Class Completion

The third important feature of Delphi's AppBrowser editor is *class completion*, activated by pressing the Ctrl+Shift+C key combination. Adding an event handler to an application is a fast operation, as Delphi automatically adds the declaration of a new method to handle the event in the class and provides you with the skeleton of the method in the implementation portion of the unit. This is part of Delphi's support for visual programming.

1.1.2.4 Code Insight

Besides the Code Explorer, class completion, and the navigational features, the Delphi editor supports the *code insight* technology. Collectively, the code insight techniques are based on a constant background parsing, both of the source code you write and of the source code of the system units your source code refers to.

1.1.3 The Form Designer

Another Delphi window you'll interact with very often is the Form Designer, a visual tool for placing components on forms. In the Form Designer, you 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, you can use the Esc key to select the parent control of the current one. You 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.

Among its other features, the Form Designer offers several Tooltip hints:

- As you move the pointer over a component, the hint shows you the name and type of the component. Delphi offers extended hints, with details on the control position, size, tab order, and more. This is an addition to the Show Component Captions environment setting, which I keep active.
- As you resize a control, the hint shows the current size (the Width and Height properties). Of course, this feature is available only for controls, not for nonvisual components (which are indicated in the Form Designer by icons).
- As you move a component, the hint indicates the current position (the Left and Top properties).

Finally, you can save DFM (Delphi Form Module) files in plain text instead of the traditional binary resource format. You can toggle this option for an individual form with the Form Designer's shortcut menu, or you can set a default value for newly created forms in the Designer page of the Environment Options dialog box. In the same page, you can also specify whether the secondary forms of a program will be automatically created at startup, a decision you can always reverse for each individual form (using the Forms page of the Project Options dialog box).

1.1.4 Additional and External Delphi Tools

Besides the IDE, when you install Delphi you get other, external tools. Some of them, such as the Database Desktop, the Package Collection Editor (PCE.exe), and the Image Editor (ImagEdit.exe), are available from Tools menu of the IDE. In addition, the Enterprise edition has a link to the SQL Monitor (SqlMon.exe).

Other tools that are not directly accessible from the IDE include many command-line tools you can find in the bin directory of Delphi. For example, there is a command-line Delphi compiler (DCC.exe), a Borland resource compiler (BRC32.exe and BRCC32.exe), and an executable viewer (TDump.exe).

Finally, some of the sample programs that ship with Delphi are actually useful tools that you can compile and keep at hand.

Web App Debugger (WebAppDbg.exe) is the debugging Web server introduced in Delphi. It is used to keep track of the requests send to your applications and debug them.

XML Mapper (XmlMapper.exe), again new in Delphi , is a tool for creating XML transformations to be applied to the format produced by the ClientDataSet component.

External Translation Manager (etm60.exe) is the stand-alone version of the Integrated Translation Manager. This external tool can be given to external translators.

Borland Registry Cleanup Utility (DRegClean.exe) helps you remove all of the Registry entries added by Delphi to a computer.

TeamSource is an advanced version-control system provided with Delphi, starting with version 5. The tool is very similar to its past incarnation and is installed separately from Delphi.

WinSight (Ws.exe) is a Windows "message spy" program available in the bin directory.

Database Explorer can be activated from the Delphi IDE or as a stand-alone tool, using the DBExplor.exe program of the bin directory.

OpenHelp (oh.exe) is the tool you can use to manage the structure of Delphi's own Help files, integrating third-party files into the help system.

Convert (Convert.exe) is a command-line tool you can use to convert DFM files into the equivalent textual description and vice versa.

Turbo Grep (Grep.exe) is a command-line search utility, much faster than the embedded Find In Files mechanism but not so easy to use.

Turbo Register Server (TRegSvr.exe) is a tool you can use to register ActiveX libraries and COM servers. The source code of this tool is available under \Demos\ActiveX\ TRegSvr.

Resource Explorer is a powerful resource viewer (but not a full-blown resource editor) you can find under \Demos\ResXplor.

Resource Workshop This is an old 16-bit resource editor that can also manage Win32 resource files. It was formerly included in Borland C++ and Pascal compilers for Windows and was much better than the standard Microsoft resource editors then available. Although its user interface hasn't been updated and it doesn't handle long filenames, this tool can still be very useful for building custom or special resources. It also lets you explore the resources of existing executable files.

1.1.5 The Files Produced by the System

Delphi produces various files for each project, and you should know what they are and how they are named. Basically, two elements have an impact on how files are named: the names you give to a project and its units, and the predefined file extensions used by Delphi. Table 1.1 lists the extensions of the files you'll find in the directory where a Delphi project resides. The table also shows when or under what circumstances these files are created and their importance for future compilations.

TABLE 1.1: Delphi Project File Extensions

Extension	File Type and Description	Creation Time	Required to Compile?
.BMP, .ICO, .CUR	Bitmap, icon, and cursor files: standard Windows files used to store bitmapped images.	Development: Image Editor	Usually not, but they might be needed at run time and for further editing.
.BPG	Borland Project Group: the files used by the new multiple-target Project Manager. It is a sort of makefile.	Development	Required to recompile all the projects of the group at once.
.BPL	Borland Package Library: a DLL including VCL components to be used by the Delphi environment at design time or by applications at run time. (These files used a .DPL extension in Delphi 3.)	Compilation: Linking	You'll distribute packages to other Delphi developers and, optionally, to end-users.
.CAB	The Microsoft Cabinet compressed-file format used for Web deployment by Delphi. A CAB file can store multiple compressed files.	Compilation	Distributed to users.
.CFG	Configuration file with project options. Similar to the DOF files.	Development	Required only if special compiler options have been set.
.DCP	Delphi Component Package: a file with symbol information for the code that was compiled into the package. It doesn't include compiled code, which is stored in DCU files.	Compilation	Required when you use packages. You'll distribute it only to other developers along with DPL files.
.DCU	Delphi Compiled Unit: the result of the compilation of a Pascal file.	Compilation	Only if the source code is not available. DCU files for the units you write are an intermediate step, so they make compilation faster.
.DDP	The new Delphi Diagram Portfolio, used by the Diagram view of the editor (was .DTI in Delphi 5)	Development	No. This file stores "design-time only" information, not required by the resulting program but very important for the programmer.

TABLE 1.1 continued: Delphi Project File Extensions

Extension	File Type and Description	Creation Time	Required to Compile?
.DFM	Delphi Form File: a binary file with the description of the properties of a form (or a data module) and of the components it contains.	Development	Yes. Every form is stored in both a PAS and a DFM file.
.~DF	Backup of Delphi Form File (DFM).	Development	No. This file is produced when you save a new version of the unit related to the form and the form file along with it.
.DFN	Support file for the Integrated Translation Environment (there is one DFN file for each form and each target language).	Development (ITE)	Yes (for ITE). These files contain the translated strings that you edit in the Translation Manager.
.DLL	Dynamic Link Library: another version of an executable file.	Compilation: Linking	See .EXE.
.DOF	Delphi Option File: a text file with the current settings for the project options.	Development	Required only if special compiler options have been set.
.DPK	Delphi Package: the project source code file of a package.	Development	Yes.
.DPR	Delphi Project file. (This file actually contains Pascal source code.)	Development	Yes.
.~DP	Backup of the Delphi Project file (.DPR).	Development	No. This file is generated automatically when you save a new version of a project file.
.DSK	Desktop file: contains information about the position of the Delphi windows, the files open in the editor, and other Desktop settings.	Development	No. You should actually delete it if you copy the project to a new directory.

TABLE 1.1 continued: Delphi Project File Extensions

Extension	File Type and Description	Creation Time	Required to Compile?
.DSM	Delphi Symbol Module: stores all the browser symbol information.	Compilation (but only if the Save Symbols option is set)	No. Object Browser uses this file, instead of the data in memory, when you cannot recompile a project.
.EXE	Executable file: the Windows application you've produced.	Compilation: Linking	No. This is the file you'll distribute. It includes all of the compiled units, forms, and resources.
.HTM	Or .HTML, for Hypertext Markup Language: the file format used for Internet Web pages.	Web deployment of an ActiveForm	No. This is not involved in the project compilation.
.LIC	The license files related to an OCX file.	ActiveX Wizard and other tools	No. It is required to use the control in another development environment.
.OBJ	Object (compiled) file, typical of the C/C++ world.	Intermediate compilation step, generally not used in Delphi	It might be required to merge Delphi with C++ compiled code in a single project.
.OCX	OLE Control Extension: a special version of a DLL, containing ActiveX controls or forms.	Compilation: Linking	See .EXE.
.PAS	Pascal file: the source code of a Pascal unit, either a unit related to a form or a stand-alone unit.	Development	Yes.
.~PA	Backup of the Pascal file (.PAS).	Development	No. This file is generated automatically by Delphi when you save a new version of the source code.
.RES, .RC	Resource file: the binary file associated with the project and usually containing its icon. You can add other files of this type to a project. When you create custom resource files you might use also the textual format, .RC.	Development Options dialog box. The ITE (Integrated Translation Environment) generates resource files with special comments.	Yes. The main RES file of an application is rebuilt by Delphi according to the information in the Application page of the Project Options dialog box.

TABLE 1.1 continued: Delphi Project File Extensions

Extension	File Type and Description	Creation Time	Required to Compile?
.RPS	Translation Repository (part of the Integrated Translation Environment).	Development (ITE)	No. Required to manage the translations.
.TLB	Type Library: a file built automatically or by the Type Library Editor for OLE server applications.	Development	This is a file other OLE programs might need.
TODO	To-do list file, holding the items related to the entire project.	Development	No. This file hosts notes for the programmers.
.UDL	Microsoft Data Link.	Development	Used by ADO to refer to a data provider. Similar to an alias in the BDE world (see Chapter 12).

Besides the files generated during the development of a project in Delphi, there are many others generated and used by the IDE itself. In Table 1.2, a short list of extensions worth knowing about. Most of these files are in proprietary and undocumented formats, so there is little you can do with them.

TABLE 1.2: Selected Delphi IDE Customization File Extensions

Extension	File Type
.DCI	Delphi code templates
.DRO	Delphi's Object Repository (The repository should be modified with the Tools ➪ Repository command.)
.DMT	Delphi menu templates
.DBI	Database Explorer information
.DEM	Delphi edit mask (files with country-specific formats for edit masks)
.DCT	Delphi component templates
.DST	Desktop settings file (one for each desktop setting you've defined)

Looking at Source Code Files

The fundamental Delphi files are Pascal source code files, which are plain ASCII text files. The bold, italic, and colored text you see in the editor depends on syntax highlighting, but it isn't saved with the file. It is worth noting that there is one single file for the whole code of the form, not just small code fragments.

For a form, the Pascal file contains the form class declaration and the source code of the event handlers. The values of the properties you set in the Object Inspector are stored in a separate form description file (with a .DFM extension). The only exception is the Name property, which is used in the form declaration to refer to the components of the form.

1.1.6 The Object Repository

Delphi has menu commands you can use to create a new form, a new application, a new data module, a new component, and so on. These commands are located in the File . New menu and in other pull-down menus. What happens if you simply select File . New . Other? Delphi opens the Object Repository, which is used to create new elements of any kind: forms, applications, data modules, thread objects, libraries, components, automation objects, and more.

1.2 The Object Pascal Language: Classes and Objects

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).

You can write Delphi applications even without knowing the details of Object Pascal. As you create a new form, add new components, and handle events, Delphi prepares most of the related code for you automatically. But knowing the details of the language and its implementation will help you to understand precisely what Delphi is doing and to master the language completely.

1.2.1 The Pascal Language

The Object Pascal language used by Delphi is an OOP extension of the classic Pascal language, which Borland pushed forward for many years with its Turbo Pascal compilers. The syntax of the Pascal language is known to be quite verbose and more readable than, for example, the C language. Its OOP extension follows the same approach, delivering the same power of the recent breed of OOP languages, from Java to C#.

There is only the object-oriented extensions of the Pascal language available in Delphi. However, highlight recent additions Borland has done to the core language. These features have been introduced in Delphi and are, at least partially, related to the Linux version of Delphi.

Delphi provides a few predefined conditional symbols, including compiler version, the operating system, the GUI environment, and so on. I've listed the most important ones in Table 2.1.

TABLE 2.1: Commonly Used Predefined Conditional Symbols

Symbol	Description
VER140	Compiling with Delphi 6, which is the 14.0 version of the Borland Pascal compiler; Delphi 5 used VER130, with lower numbers for past versions.
MSWINDOWS	Compiling on the Windows platform (new in Delphi 6).
LINUX	Compiling on the Linux platform. On Kylix, there are also the LINUX32, POSIX, and ELF predefined symbols.
WIN32	Compiling only on the 32-bit Windows platform. This symbol was introduced in Delphi 2 to distinguish from 16-bit Windows compilations (Delphi 1 defined the WINDOWS symbol). You should use WIN32 only to mark code specifically for Win32, not Win16 or future Win64 platforms (for which the WIN64 symbol has been reserved). Use MSWINDOWS, instead, to distinguish between Windows and other operating systems.
CONSOLE	Compiling a console application, and not a GUI one. This symbol is meaningful only under Windows, as all Linux applications are console applications.
BCB	Defined when the C++Builder IDE invokes the Pascal compiler.
ConditionalExpressions	Indicates that the $IF directive is available. It is defined in Kylix and Delphi 6, but not in earlier versions.

New Hint Directives

Supporting multiple operating systems within the same source code base implies a number of compatibility issues. Besides a modified run-time library and a wholly new component library .Delphi includes special directives Borland uses to mark special portions of the code. As they introduced the idea of custom warnings and messages (described in the previous section), they've added a few special predefined ones.

1.2.2 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.

1.2.2.1 Classes, Objects, and Visual Programming

Even if you simply create a new application with a form and place a button over it to execute some code when the button is pressed, you are building an object-oriented application. In fact, the form is an object of a new class (by default TForm1, which inherits from the base TForm class provided by Borland), and the button is an instance of the TButton class, provided by Borland.

Not using objects and classes with Delphi would probably be more difficult than using them, as you have to give up all of the design-time tools for visual programming. In any case, the real challenge is using OOP properly, something I'll try to teach you in this chapter (and in the rest of the book), along with an introduction to the key elements of the Object Pascal language.

1.2.2.2 Overloaded Methods

Object Pascal supports overloaded functions and methods: you 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:

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

1.2.2.3 Class Methods and Class Data

When you define a field in a class, you 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. You 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.

1.2.3 Encapsulation

A class can have any amount of data and any number of methods. However, for a good objectoriented approach, data should be hidden, or *encapsulated*, inside the class using it.

When you access a date, for example, it makes no sense to change the value of the day by itself. In fact, changing the value of the day might result in an invalid date, such as February 30. Using methods to access the internal representation of an object limits the risk of generating erroneous situations, as the methods can check whether the date is valid and refuse to modify the new value if it is not. Encapsulation is important because it allows the class writer to modify the internal representation in a future version.

1.2.3.1 Encapsulation and Units

A unit in Object Pascal is a secondary source-code file, with the main source-code file being represented by the project source code. Every unit has two main sections, called interface and implementation, as well as two optional ones for initialization and finalization code.

In short, every identifier (type, routine, variable, and so on) that you declare in the interface portion of a unit becomes visible to any other unit of the program, provided there is a uses statement referring back to the unit that defines the identifier. All the routines and methods you declare in the interface portion of the unit must later be fully defined in the implemented portion of the same unit. In the interface section of a unit, however, you cannot write any actual statements to execute.

1.2.3.2 Private, Protected, and Public

For class-based encapsulation, the Object Pascal language has three access specifiers: private, protected, and public. A fourth, published, controls RTTI and design time information .Here are the three *classic* access specifiers:

- . 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.
- . The protected directive is used to indicate methods and fields with limited visibility. Only the current class and its subclasses can access protected elements. We'll discuss this keyword again in the "Protected Fields and Encapsulation" section.
- . 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.

1.2.4 Constructors

To allocate the memory for the object, we call the Create method. This is a *constructor,* a special method that you can apply to a class to allocate memory for an instance of that class. The instance is returned by the constructor and can be assigned to a variable for storing the object and using it later on. The default TObject.Create constructor initializes all the data of the new instance to zero.

1.2.4.1 Overloaded Constructors

Overloading is particularly relevant for constructors, because we can add to a class multiple constructors and call them all Create, which makes them easy to remember.

NOTE Historically, overloading was added to C++ to allow the use of multiple constructors that have the same name (the name of the class). In Object Pascal, this feature was considered unnecessary because multiple constructors can have different specific names. The increased integration of Delphi with C++Builder has motivated Borland to make this feature available in both languages, starting with Delphi 4. Technically, when C++Builder constructs an instance of a Delphi VCL class, it looks for a Delphi constructor named Create and nothing but Create. If the Delphi class has constructors by other names, they cannot be used from C++Builder code. Therefore, when creating classes and components you intend to share with C++Builder programmers, you should be careful to name all your constructors Create and distinguish between them by their parameter lists (using overload). Delphi does not require this, but it is required for C++Builder to use your Delphi classes.

1.2.4.2 Destructors

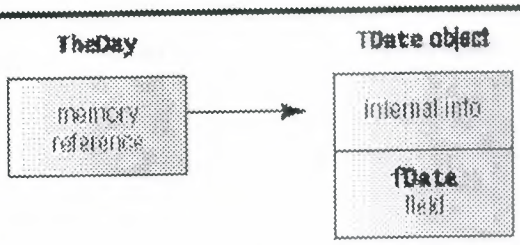
In the same way that a class can have a custom constructor, it can have a custom destructor, a method declared with the destructor keyword and called Destroy, which can perform some resource cleanup before an object is destroyed. Just as a constructor call allocates memory for the object, a destructor call frees the memory.

1.2.5 Delphi's Object Reference Model

In some OOP languages, declaring a variable of a class type creates an instance of that class. Object Pascal, instead, is based on an *object reference model.* The idea is that a variable of a class type, such as the TheDay variable in the preceding ViewDate example, does not hold the value of the object. Rather, it contains a reference, or a *pointer,* to indicate the memory location where the object has been stored. You can see this structure depicted in Figure 2.5

FIGURE 2.5:

A representation of the structure of an object in memory with a variable referring to it



The only problem with this approach is that when you declare a variable, you don't create an object in memory; you only reserve the memory location for a reference to an object. Object instances must be created manually, at least for the objects of the classes you define. Instances of the components you place on a form are built automatically by Delphi.

You've seen how to create an instance of an object by applying a constructor to its class. Once you have created an object and you've finished using it, you need to dispose of it (to avoid filling up memory you don't need any more, which causes what is known as a *memory leak*). This can be accomplished by calling the Free method. As long as you create objects when you need them and free them when you're finished with them, the object reference model works without a glitch. The object reference model has many consequences on assigning object and on managing memory.

1.2.5.1 Assigning Objects

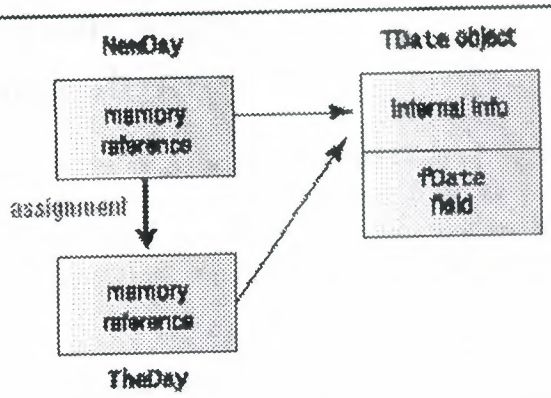
If a variable holding an object only contains a reference to the object in memory, what happens if you copy the value of that variable? Suppose we write the BtnTodayClick method of the ViewDate example in the following way:

```
procedure TDateForm.BtnTodayClick(Sender: TObject);
var
NewDay: TDate;
begin
NewDay := TDate.Create;
TheDay := NewDay;
LabelDate.Caption := TheDay.GetText;
end;
```

This code copies the memory address of the NewDay object to the TheDay variable (as shown in Figure 2.6); it doesn't copy the data of an object into the other. In this particular circumstance, this is not a very good approach, as we keep allocating memory for a new object every time the button is pressed, but we never release the memory of the object the TheDay variable was previously pointing to. This specific issue can be solved by freeing the old object, as in the following code (which is also simplified, without the use of an explicit variable for the newly created object):

FIGURE 2.6:

A representation of the operation of assigning an object reference to another one. This is different from copying the actual content of an object to another.



1.2.5.2 Objects and Memory

Memory management in Delphi is subject to three rules: Every object must be created before it can be used; every object must be destroyed after it has been used; and every object must be destroyed only once. Whether you have to do these operations in your code, or you can let Delphi handle memory management for you, depends on the model you choose among the different approaches provided by Delphi.

Delphi supports three types of memory management for dynamic elements (that is, elements not in the stack and the global memory area):

- Every time you create an object explicitly, in the code of your application, you should also free it. If you fail to do so, the memory used by that object won't be released for other objects until the program terminates.
- When you create a component, you can specify an owner component, passing the owner to the component constructor. The owner component (often a form) becomes responsible for destroying all the objects it owns. In other words, when you free the form, it frees all the components it owns. So, if you create a component and give it an owner, you don't have to remember to destroy it. This is the standard behavior of the components you create at design time by placing them on a form or data module.
- When you allocate memory for strings, dynamic arrays, and objects referenced by interface variables, Delphi automatically frees the memory when the reference goes out of scope. You don't need to free a string: when it becomes unreachable, its memory is released.

1.3 The Object Pascal Language: Inheritance and Polymorphism

After the introduction to classes and objects we've seen over the last chapter, let's move on to another key element of the language, *inheritance*. Deriving a class from an existing one is the real revolutionary idea of object-oriented programming, and it goes along with polymorphism, virtual functions, abstract functions, and many other topics discussed in this chapter.

We'll focus also on interfaces, another intriguing idea of the most recent OOP languages, and we'll cover a few more elements of Object Pascal, such as exception handling and class references. Together with the previous chapter, this will provide an almost complete roundup of the language.

1.3.1 Inheriting from Existing Types

We often need to use a slightly different version of an existing class that we have written or that someone has given to us. For example, you might need to add a new method or slightly change an existing one. You can do this easily by modifying the original code, unless you want to be able to use the two different versions of the class in different circumstances. Also, if the class was originally written by someone else (including Borland), you might want to keep your changes separate.

A typical alternative is to make a copy of the original type definition, change its code to support the new features, and give a new name to the resulting class. This might work, but it also might create problems: In duplicating the code you also duplicate the bugs; and if you want to add a new feature, you'll need to add it two or more times, depending on the number of copies of the original code you've made. This approach results in two completely different data types, so the compiler cannot help you take advantage of the similarities between the two types.

1.3.2 Late Binding and Polymorphism

Pascal functions and procedures are usually based on *static* or *early binding*. This means that a method call is resolved by the compiler and linker, which replace the request with a call to the specific memory location where the function or procedure resides. (This is known as the *address* of the function.) OOP languages allow the use of another form of binding, known as *dynamic* or *late binding*. In this case, the actual address of the method to be called is determined at run time based on the type of the instance used to make the call.

The advantage of this technique is known as *polymorphism*. Polymorphism means you can write a call to a method, applying it to a variable, but which method Delphi actually calls depends on the type of the object the variable relates to. Delphi cannot determine until run time the actual class of the object the variable refers to, because of the type-compatibility rule discussed in the previous section.

1.3.3 Type-Safe Down-Casting

The Object Pascal type-compatibility rule for descendant classes allows you to use a descendant class where an ancestor class is expected.The reverse is not possible.Now suppose that the TDog class has an Eat method, which is not present in the Tanimal class. If the variable MyAnimal refers to a dog, it should be possible to call the function. But if you try, and the variable is referring to another class, the result is an error. By making an explicit typecast, we could cause a nasty run-time error (or worse, a subtle memory overwrite problem), because the compiler cannot determine whether the type of the object is correct and the methods we are calling actually exist.

1.3.4 Using Interfaces

When you define an abstract class to represent the base class of a hierarchy, you can come to a point in which the abstract class is so abstract that it only lists a series of virtual functions without providing any actual implementation. This kind of *purely abstract class* can also be defined using a specific technique, an interface. For this reason, we refer to these classes as *interfaces*.

Technically, an interface is not a class, although it may resemble one. Interfaces are not classes, because they are considered a totally separate element with distinctive features:

- Interface type objects are reference-counted and automatically destroyed when there are no more references to the object. This mechanism is similar to how Delphi manages long strings and makes memory management almost automatic.
- A class can inherit from a single base class, but it can implement multiple interfaces.
- As all classes descend from TObject, all interfaces descend from IInterface, forming a totally separate hierarchy.

1.3.5 Working with Exceptions

Another key feature of Object Pascal is the support for *exceptions*. The idea of exceptions is to make programs more robust by adding the capability of handling software or hardware errors in a uniform way. A program can survive such errors or terminate gracefully, allowing the user to save data before exiting. Exceptions allow you to separate the error-handling code from your normal code, instead of intertwining the two. You end up writing code that is more compact and less cluttered by maintenance chores unrelated to the actual programming objective.

Another benefit is that exceptions define a uniform and universal error-reporting mechanism, which is also used by Delphi components. At run time, Delphi raises exceptions when something goes wrong (in the run-time code, in a component, in the operating system). From the point of the code in which it is raised, the exception is passed to its calling code, and so on. Ultimately, if no part of your code handles the exception, Delphi handles it, by displaying a standard error message and trying to continue the program, by handing the next system message
or user request.

Debugging and Exceptions

When you start a program from the Delphi environment (for example, by pressing the F9 key), you'll generally run it within the debugger. When an exception is encountered, the debugger will stop the program by default. This is normally what you want, of course, because you'll know where the exception took place and can see the call of the handler step-by-step. You can also use the Stack Trace feature of Delphi to see the sequence of function and method calls, which caused the program to raise an exception.

In the case of the Exception1 test program, however, this behavior will confuse the program's execution. In fact, even if the code is prepared to properly handle the exception, the debugger will stop the program execution at the source code line closest to where the exception was raised. Then, moving step-by-step through the code, you can see how it is handled. If you just want to let the program run when the exception is properly handled, run the program from Windows Explorer, or temporarily disable the Stop on Delphi Exceptions options in the Language Exceptions page of the Debugger Options dialog box (activated by the Tools . Debugger Options command), shown in the Language Exceptions page of the Debugger Options dialog box .

1.4 The Run-Time Library

Delphi uses Object Pascal as its programming language and favors an object-orientedapproach, tied with a visual development style. This is where Delphi shines, and we will cover component-based and visual development, however, the fact that a lot of ready-to-use features of Delphi come from its run-time library, or RTL for short. This is a large collection of functions you can use to perform simple tasks, as well as some complex ones, within your Pascal code. (I use "Pascal" here, because the run-time library mainly contains procedures and functions and not classes and objects.)

There is actually a second reason to devote this chapter of the book to the run-time library: Delphi 6 sees a large number of enhancements to this area. There are new groups of functions, functions have been moved to new units, and other elements have changed, creating a few incompatibilities with existing code. So even if you've used past versions of Delphi and feel confident with the RTL, you should still read at least portions of this chapter.

1.4.1 The Units of the RTL

In Delphi the RTL (run-time library) has a new structure and several new units. The reason for adding new units is that many new functions were added. In most cases, you'll find the existing functions in the units where they used to be, but the new functions will appear in specific units. For example, new functions related to dates are now in the DateUtils unit, but existing date functions have not been moved away from SysUtils in order to avoid incompatibilities with existing code.

The exception to this rule relates to some of the variant support functions, which were moved out of the System unit to avoid unwanted linkage of specific Windows libraries, even in programs that didn't use those features. These variant functions are now part of the new Variants unit, described later in the chapter.

1.4.1.1 The System and SysInit Units

System is the core unit of the RTL and is automatically included in any compilation (considering an automatic and implicit uses statement referring to it). Actually, if you try adding the unit to the uses statement of a program, you'll get the compile-time error: [Error] Identifier redeclared: System .The System unit includes, among other things:

- The TObject class, the base class of any class defined in the Object Pascal language, including all the classes of the VCL.

- The IUnknown and IDispatch interfaces as well as the simple implementation class TInterfacedObject. There are also the new IInterface and IInvokable interfaces. IInterface was added to underscore the point that the interface type in Delphi's Object Pascal language definition is in no way dependent on the Windows operating system (and never has been). IInvokable was added to support SOAP-based invocation.

- Some variant support code, including the variant type constants, the TVarData record type and the new TVariantManager type, a large number of variant conversion routines, and also variant records and dynamic arrays support. This area sees a lot of changes compared to Delphi. The basic information on variants is provided in *Essential Pascal* , while an introduction to custom variants is available later in this chapter.

- Many base data types, including pointer and array types and the TDateTime type we've already described in the last chapter.

- Memory allocation routines, such as GetMem and FreeMem, and the actual memory manager, defined by the TMemoryManager record and accessed by the GetMemoryManager and SetMemoryManager functions. For information, the GetHeapStatus function returns a THeapStatus data structure. Two new global variables (AllocMemCount and AllocMemSize) hold the number and total size of allocated memory blocks. There is more on memory and the use of these functions.

- Package and module support code, including the PackageInfo pointer type, the GetPackageInfoTable global function, and the EnumModules procedure.

- A rather long list of global variables, including the Windows application instance Main- Instance; IsLibrary, indicating whether the executable file is a library or a stand-alone program; IsConsole, indicating console applications; IsMultiThread, indicating whether there are secondary threads; and the command-line string CmdLine. (The unit includes also the ParamCount and ParamStr for an easy access to command-line parameters.) Some of these variables are specific to the Windows platform.

- Thread-support code, with the BeginThread and EndThread functions; file support records and file-related routines; wide string and OLE string conversion routines; and many other low-level and system routines (including a number of automatic conversion functions).

The companion unit of System, called SysInit, includes the system initialization code, with functions you'll seldom use directly. This is another unit that is always implicitly included, as it is used by the System unit.

1.4.1.2 The SysUtils and SysConst Units

The SysConst unit defines a few constant strings used by the other RTL units for displaying messages. These strings are declared with the resourcestring keyword and saved in the program resources. As other resources, they can be translated by means of the Integrated Translation Manager or the External Translation Manager.

The SysUtils unit is a collection of system utility functions of various types. Different from other RTL units, it is in large part an operating system–dependent unit. The SysUtils unit has no specific focus, but it includes a bit of everything, from string management to locale and multibyte-characters support, from the Exception class and several other derived exception classes to a plethora of string-formatting constants and routines.

Some of the features of SysUtils are used every day by every programmer as the IntToStr or Format string-formatting functions; other features are lesser known, as they are the Windows version information global variables. These indicate the Windows platform (Window 9*x* or NT/2000), the operating system version and build number, and the eventual service pack installed on NT.

1.5 Core Library Classes

1.5.1 The RTL Package, VCL, and CLX

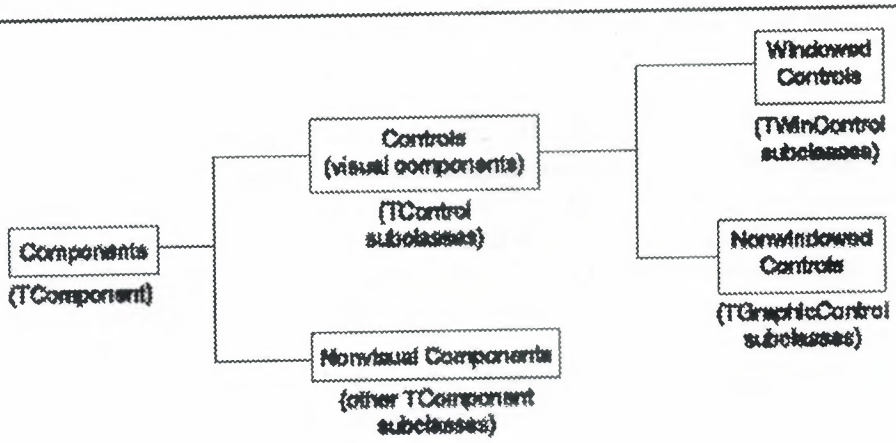
Until version, Delphi's class library was known as VCL, which stands for Visual Components Library. Kylix, the Delphi version for Linux, introduced a new component library, called CLX (pronounced "clicks" and standing for Component Library for X-Platform or Cross Platform). Delphi 6 includes both the VCL and CLX libraries. For visual components, the two class libraries are alternative one to the other. However, the core classes and the database and Internet portions of the two libraries are basically shared.

VCL was considered as a single large library, although programmers used to refer to different parts of it (components, controls, nonvisual components, data sets, data-aware controls, Internet components, and so on). CLX introduces a distinction in four parts: BaseCLX, VisualCLX, DataCLX, and NetCLX. Only in VisualCLX does the library use a totally different approach between the two platforms, with the rest of the code being inherently portable to Linux. In the following section, I discuss the sections of these two libraries, while the rest of the chapter focuses on the common core classes.

1.5.1.1 Traditional Sections of VCL

Delphi programmers use to refer to different sections of VCL with names Borland originally suggested in its documentation, and names that became common afterwards for different groups of components.
Technically, components are subclasses of the TComponent class, which is one of the root classes of the hierarchy, as you can see in Figure 5.1. Actually the Tcomponent class inherits from the TPersistent class; the role of these two classes will be explained in the next section.

FIGURE 5.1:
A graphical representation of the main groups of components of VCL.



Controls have a position and a size on the screen and show up in the form at design time in the same position they'll have at run time. Controls have two different subspecifications, window-based or graphical.

Nonvisual components are all the components that are not controls—all the classes that descend from TComponent but not from TControl. At design time, a nonvisual component appears on the form as an icon (optionally with a caption below it). At run time, some of these components may be visible (for example, the standard dialog boxes), and others are always invisible (for example, the database table component).

1.5.1.2 The Structure of CLX

This is the traditional subdivision of VCL, which is very common for Delphi programmers. Even with the introduction of CLX and some new naming schemes, the traditional names will probably survive and merge into Delphi programmers' jargon.

Borland now refers to different portions of the CLX library using one terminology under Linux and a slightly different (and less clear) naming structure in Delphi. This is the subdivision of the Linux-compatible library:

BaseCLX forms the core of the class library, the topmost classes (such as TComponent), and several general utility classes (including lists, containers, collections, and streams). Compared to the corresponding classes of VCL, BaseCLX is largely unchanged and is

highly portable between the Windows and Linux platforms. This chapter is largely devoted to exploring BaseCLX and the common VCL core classes.

VisualCLX is the collection of visual components, generally indicated as controls. This is the portion of the library that is more tightly related to the operating system: VisualCLX is implemented on top of the Qt library, available both on Windows and on Linux. Using VisualCLX allows for full portability of the visual portion of your application between Delphi on Windows and Kylix on Linux. However, most of the VisualCLX components have corresponding VCL controls, so that you can also easily move your code from one library to the other.

DataCLX comprises all the database-related components of the library. Actually, DataCLX is the front end of the new dbExpress database engine included in Delphi 6 and Kylix. Delphi includes also the traditional BDE front end, dbGo, and InterBase Express (IBX). If we can consider all these components as part of DataCLX, only the dbExpress front end and IBX are portable between Windows and Linux. DataCLX includes also the ClientDataSet component, now indicated as MyBase, and other related classes.

NetCLX includes the Internet-related components, from the WebBroker framework, to the HTML producer components, from Indy (Internet Direct) to Internet Express, from the new Site Express to XML support. This part of the library is, again, highly portable between Windows and Linux.

1.5.1.3 VCL-Specific Sections of the Library

The preceding areas of the library are available, with the differences I've mentioned, on both Delphi and Kylix. In Delphi, however, there are other sections of VCL, which for one reason or another are specific to Windows only:

- The Delphi ActiveX (DAX) framework provides support for COM, OLE Automation, ActiveX, and other COM-related technologies.
- The Decision Cube components provide OLAP support but have ties with the BDE and haven't been updated recently.

Finally, the default Delphi installation includes some third-party components, such as TeeChart for business graphics and QuickReport for reporting. These components will be mentioned in the book but are not discussed in detail.

1.5.2 Events in Delphi

When a user does something with a component, such as clicking it, the component generates an event. Other events are generated by the system, in response to a method call or a change to one of that component's properties (or even a different component's). For

example, if you set the focus on a component, the component currently having the focus loses it, triggering the corresponding event.

Technically, most Delphi events are triggered when a corresponding operating system message is received, although the events do not match the messages on a one-to-one basis. Delphi events tend to be higher-level than operating system messages, and Delphi provides a number of extra inter-component messages.

From a theoretical point of view, an event is the result of a request sent to a component or control, which can respond to the message. Following this approach, to handle the click event of a button, we would need to subclass the TButton class and add the new event handler code inside the new class.

Method Pointers

Events rely on a specific feature of the Object Pascal language: *method pointers*. A method pointer type is like a procedural type, but one that refers to a method. Technically, a method pointer type is a procedural type that has an implicit Self parameter. In other words, a variable of a procedural type stores the address of a function to call, provided it has a given set of parameters. A method pointer variable stores two addresses: the address of the method code and the address of an object instance (data). The address of the object instance will show up as Self inside the method body when the method code is called using this method pointer.

1.5.3 Summarizing the Core VCL and BaseCLX Units

This unit is certainly important, but it is not the only core unit of the library (although there aren't many others). In this section, We are providing an overview of these units and their content.

1.5.3.1 The Classes Unit

The Classes unit is at the heart of both VCL and CLX libraries, and though it sees many internal changes from the last version of Delphi, there is little new for the average users. (Most changes are related to modified IDE integration and are meant for expert component writers.)

Here is a list of what you can find in the Classes unit, a unit that every Delphi programmer should spend some time with:

- Many enumerated types, the standard method pointer types (including TNotifyEvent), and many exception classes.

- Core library classes, including TPersistent and TComponent but also TBasicAction and TBasicActionLink.

- List classes, including TList, TThreadList (a thread-safe version of the list), TinterfaceList (a list of interfaces, used internally), TCollection, TCollectionItem, TownedCollection (which is simply a collection with an owner), TStrings, and TStringList.

- All the stream classes I discussed in the previous section but won't list here again. There are also the TFiler, TReader, and TWriter classes and a TParser class used internally for DFM parsing.

- Utility classes, such as TBits for binary manipulation and a few utility routines (for example, point and rectangle constructors, and string list manipulation routines such as LineStart and ExtractStrings). There are also many registration classes, to notify the system of the existence of components, classes, special utility functions you can replace, and much more.

- The TDataModule class, a simple object container alternative to a form. Data modules can contain only nonvisual components and are generally used in database and Web applications.

- New interface-related classes, such as TInterfacedPersistent, aimed at providing further support for interfaces. This particular class allows Delphi code to hold onto a reference to a TPersistent object or any descendent implementing interfaces, and is a core element of the new support for interfaced objects in the Object Inspector.

- The new TRecall class, used to maintain a temporary copy of an object, particularly useful for graphical-based resources.

- The new TClassFinder class used for finding a registered class instead of the Find-Class method.

- The TThread class, which provides the core to operating system–independent support for multithreaded applications.

1.5.3.2 Other Core Units

Other units that are part of the RTL package are not directly used by typical Delphi programmers as often as Classes. Here is a list:

- The TypInfo unit includes support for Accessing RTTI information for published properties, as we've seen in the section "Accessing Properties by Name."

- The SyncObjs unit contains a few generic classes for thread synchronization. Of course, the RTL package also includes the units with functions and procedures discussed in the preceding chapter, such as Math, SysUtils, Variants, VarUtils, StrUtils, DateUtils, and so on.

Chapter-2
VISUAL PROGRAMMING

2.1 Controls: VCL Versus VisualCLX

The use of components and the development of the user interface of applications. This is really what Delphi is about. Visual programming using components is the key feature of this development environment.

Delphi comes with a large number of ready-to-use components.Examining each of its properties and methods; if you need this information, you can find it in the Help system.

We'll start with a comparison of the VCL and VisualCLX libraries available in Delphi and a coverage of the core classes (particularly TControl). Then we'll try to list all the various visual components you have, because choosing the right basic controls is often a way to get into a project faster.

2.1.1 VCL versus VisualCLX

Delphi introduces the new 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.

VisualCLX, the visual portion of CLX, is a wrapper of the Qt (pronounced "cute") library. It includes wrappers of the native Qt widgets, which range from basic to advanced controls, very similar to Windows' own standard and common controls. It includes also painting support using another, similar, TCanvas class. Qt is a C++ class library, developed by Trolltech (www.trolltech.com), a Norwegian company with a strong relationship with Borland. On Linux, Qt is one of the de facto standard user-interface libraries and is the

basis of the KDE desktop environment. On Windows, Qt provides an alternative to the use of the native APIs: In fact, unlike VCL, which provides a wrapper to the native controls, Qt provides an alternate implementation to those controls.

This allows programs to be truly portable, as there are no hidden differences created by the operating system (and that the operating system vendor can introduce behind the scenes). It also allows us to avoid an extra layer; CLX on top of Qt on top of Windows native controls suggests three layers, but in fact there are two layers in each solution (CLX controls on top of Qt, VCL controls on top of Windows).

Technically, there are huge differences behind the scenes between a native Windows application built with VCL and a portable Qt program developed with VisualCLX. Suffice to say that at the low level, Windows uses API function calls and messages to communicate with controls, while Qt uses class methods and direct method callbacks and has no internal messages.

Technically, the Qt classes offer a high-level object-oriented architecture, while the Windows API is still bound to its C legacy and a message-based system dated 1985 (when Windows was released). VCL offers an object-oriented abstraction on top of a low-level API, while Visual- CLX remaps an already high-level interface into a more *familiar* class library.

DFM and XFM

As you 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 you want to convert a DFM file into an XFM file, you 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.

Table 6.1 is a comparison of the names of the visual VCL and CLX units, excluding the database portion and some rarely referenced units:

TABLE 6.1: Names of Equivalent VCL and CLX Units

VCL	CLX
ActnList	QActnList
Buttons	QButtons
Clipbrd	QClipbrd
ComCtrls	QComCtrls
Consts	QConsts
Controls	QControls
Dialogs	QDialogs
ExtCtrls	QExtCtrls
Forms	QForms
Graphics	QGraphics
Grids	QGrids
ImgList	QImgList
Menus	QMenus
Printers	QPrinters
Search	QSearch
StdCtrls	QStdCtrls

2.1.2 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 TWidget- Control and TGraphicControl.

2.1.3 Working with Menus

Working with menus and menu items is generally quite simple. This section offers only some very brief notes and a few more advanced examples. The first thing to keep in mind about menu items is that they can serve different purposes:

Commands are menu items used to execute an action.

State-setters are menu items used to toggle an option on and off, to change the state of a particular element. These commands usually have a check mark on the left to indicate they are active. In Delphi you can automatically obtain this behavior using the handy AutoCheck property.

Radio items have a round check mark and are grouped to represent alternative selections, like radio buttons. To obtain radio menu items, simply set the RadioItem property to True and set the GroupIndex property for the alternative menu items to the same value.

Dialog menu items cause a dialog box to appear and are usually indicated by an ellipsis (three dots) after the text. As you enter new elements in the Menu Designer, Delphi creates a new component for each menu item and lists it in the Object Inspector (although nothing is added to the form). To name each component, Delphi uses the caption you enter and appends a number (so that *Open* becomes Open1). Because Delphi removes spaces and other special characters in the caption when it creates the name, and the menu item separators are set up using a hyphen as caption, these items would have an empty name. For this reason Delphi adds the letter *N* to the name, appending the number and generating items called N1, N2, and so on.

Accelerator Keys

Since Delphi 5, you don't need to enter the & character in the Caption of a menu item; it provides an automatic accelerator key if you omit one. Delphi's automatic accelerator-key system can also figure out if you have entered conflicting accelerator keys and fix them on-the-fly. This doesn't mean you should stop adding custom accelerator keys with the & character, because the automatic system simply uses the first available letter, and it doesn't follow the default standards. You might also find better mnemonic keys than those chosen by the automatic system.

This feature is controlled by the AutoHotkeys property, which is available in the main menu component and in each of the pull-down menus and menu items. In the main menu, this property defaults to maAutomatic, while in the pull-downs and menu items it defaults to maParent, so that the value you set for the main menu component will be used automatically by all the subitems, unless they have a specific value of maAutomatic or maManual. The engine behind this system is the RethinkHotkeys method of the TMenuItem class, and the companion InternalRethinkHotkeys. There is also a RethinkLines method, which checks whether a pull-down has two consecutive separators or begins or ends with a separator. In all these cases, the separator is automatically removed.

2.1.4 Owner-Draw Controls and Styles

Let's return briefly to menu graphics. Besides using an ImageList to add glyphs to the menu items, you can turn a menu into a completely graphical element, using the owner-draw technique. The same technique also works for other controls, such as list boxes. In Windows, the system is usually responsible for painting buttons, list boxes, edit boxes, menu items, and similar elements. Basically, these controls know how to paint themselves. As an alternative, however, the system allows the owner of these controls, generally a form, to paint them. This technique, available for buttons, list boxes, combo boxes, and menu items, is called *owner-draw*.

In VCL, the situation is slightly more complex. The components can take care of painting themselves in this case (as in the TBitBtn class for bitmap buttons) and possibly activate corresponding events. The system sends the request for painting to the owner (usually the form), and the form forwards the event back to the proper control, firing its event handlers. In CLX, some of the controls, such as ListBoxes and ComboBoxes, surface events very similar to Windows owner-draw, but menus lack them. The native approach of Qt is to use styles to determine the graphical behavior of all of the controls in the system, of a specific application, or of a given control. I'll introduce styles shortly, later in this section.

CLX Styles

In Windows, the system has full control of the user interface of the controls, unless the program takes over using owner-draw or other advanced techniques. In Qt (and in Linux in general), the user chooses the user interface style of the controls. A system will generally offer a few basic styles, such as the Windows look-and-feel, the Motif one, and others. A user can add also install new styles in the system and make them available to applications.

2.2 Advanced VCL Controls

In the preceding chapter, I discussed the core concepts of the TControl class and its derived classes in the VCL and VisualCLX libraries. After that, I provided a sort of rapid tour of the key controls you can use to build a user interface, including editing components, lists, range selectors, and more.

This provides more details on some of these components (such as the ListView and TreeView) and then discusses other controls used to define the overall design of a form, such as the PageControl, TabControl, and Splitter. The chapter also presents examples of splitting forms and resizing controls dynamically. These topics are not particularly complex, but it is worth examining their key concepts briefly.

2.2.1 ListView and TreeView Controls

The standard list box and combo box components are still very common, but they are often replaced by the more powerful ListView and TreeView controls. Again, these two controls are part of the Win32 common controls, stored in the ComCtl32.DLL library. Similar controls are available in Qt and VisualCLX.

2.2.2 Multiple-Page Forms

When you have a lot of information and controls to display in a dialog box or a form, you can use multiple pages. The metaphor is that of a notebook: Using tabs, a user can select one of the possible pages. There are two controls you can use to build a multiple-page application in Delphi:

- You can use the PageControl component, which has tabs on one of the sides and multiple pages (similar to panels) covering the rest of its surface. As there is one page per tab, you can simply place components on each page to obtain the proper effect both at design time and at run time.

- You can use the TabControl, which has only the tab portion but offers no pages to host the information. In this case, you'll want to use one or more components to mimic the *page change* operation.

2.2.3 Form-Splitting Techniques

There are several ways to implement form-splitting techniques in Delphi, but the simplest approach is to use the Splitter component, found in the Additional page of the Component Palette. To make it more effective, the splitter can be used in combination with the Constraints property of the controls it relates to.

2.2.4 Control Anchors

Here we've described how you can use alignment and splitters to create nice, flexible user interfaces, that adapt to the current size of the form, giving users maximum freedom. Delphi also supports right and bottom anchors. Before this feature was introduced in Delphi 4, every control placed on a form had coordinates relative to the top and bottom, unless it was aligned to the bottom or right sides. Aligning is good for some controls but not all of them, particularly buttons.

By using anchors, you can make the position of a control relative to any side of the form. For example, to have a button anchored to the bottom-right corner of the form, place the button in the required position and set its Anchors property to [akRight, akBottom]. When the form size changes, the distance of the button from the anchored sides is kept fixed. In other words, if you set these two anchors and remove the two defaults, the button will remain in the bottom-right corner.

On the other hand, if you place a large component such as a Memo or a ListBox in the middle of a form, you can set its Anchors property to include all four sides. This way the control will behave as an aligned control, growing and shrinking with the size of the form, but there will be some margin between it and the form sides.

2.2.5 The ToolBar Control

In early versions of Delphi, toolbars had to be created using panels and speed buttons. Starting with version 3, Delphi introduced a specific ToolBar component, which encapsulates the corresponding Win32 common control or the corresponding Qt widget in VisualCLX. This component provides a toolbar, with its own buttons, and it has many advanced capabilities. To use this component, you place it on a form and then use the component editor (the shortcut menu activated by a right mouse button click) to create a few buttons and separators.

2.3 Building the User Interface

Modern Windows applications usually have multiple ways of giving a command, including menu items, toolbar buttons, shortcut menus, and so on. To separate the actual commands a user can give from their multiple representations in the user interface, Delphi has the idea of actions. In Delphi 6 this architecture has been largely extended to make the construction of the user interface on top of actions totally visual. You can now also easily let the user of your programs customize this interface, as happens in many professional programs. This chapter focuses on actions, action lists and action managers, and the related components. It also covers a few related topics, such as toolbar container controls and toolbar docking, and docking in general.

2.3.1 The ActionList Component

Delphi's event architecture is very open: You can write a single event handler and connect it to the OnClick events of a toolbar button and a menu. You can also connect the same event handler to different buttons or menu items, as the event handler can use the Sender parameter to refer to the object that fired the event. It's a little more difficult to synchronize the status of toolbar buttons and menu items. If you have a menu item and a toolbar button that both toggle the same option, every time the option is toggled, you must both add the check mark to the menu item and change the status of the button to show it pressed.

Predefined Actions in Delphi

With the action list editor, you can create a brand new action or choose one of the existing actions registered in the system. These are listed in a secondary dialog box. There are many predefined actions, which can be divided into logical groups:

File actions include open, save as, open with, run, print setup, and exit.
Edit actions are illustrated in the next example. They include cut, copy, paste, select all, undo, and delete.

RichEdit actions complement the edit actions for RichEdit controls and include bold, italic, underline, strikeout, bullets, and various alignment actions.
MDI window actions as we examine the Multiple Document Interface approach. They include all the most common MDI operations: arrange, cascade, close, tile (horizontally or vertically), and minimize all.
Dataset actions There are many dataset actions, representing all the main operations you can perform on a dataset.
Help actions allow you to activate the contents page or index of the Help file attached to the application.
Search actions include find, find first, find next, and replace.
Tab and Page control actions include previous page and next page navigation.
Dialog actions activate color, font, open, save, and print dialogs.
List actions include clear, copy, move, delete, and select all. These actions let you interact with a list control. Another group of actions, including static list, virtual list, and some support classes, allow the definition of lists that can be connected to a user interface. More on this topic is in the section "Using List Actions" toward the end of this chapter.
Web actions include browse URL, download URL, and send mail actions.
Tools actions include only the dialog to customize the action bars.

2.3.2 Toolbar Containers

Most modern applications have multiple toolbars, generally hosted by a specific container. Microsoft Internet Explorer, the various standard business applications, and the Delphi IDE all use this general approach. However, they each implement this differently. Delphi has two ready-to-use toolbar containers, the CoolBar and the ControlBar components. They have differences in their user interface, but the biggest one is that the CoolBar is a Win32 common control, part of the operating system, while the ControlBar is a VCL-based component. Both components can host toolbar controls as well as some extra elements such as combo boxes and other controls. Actually, a toolbar can also replace the menu of an application.

2.3.3 Delphi's Docking Support

Another feature added in Delphi 4 was support for *dockable* toolbars and controls. In other words, you can create a toolbar and move it to any of the sides of a form, or even move it freely on the screen, undocking it. However, setting up a program properly to obtain this effect is not as easy as it sounds.

First of all, Delphi's docking support is connected with container controls, not with forms. A panel, a ControlBar, and other containers (technically, any control derived from TWinControl) can be set up as dock targets by enabling their DockSite property. You can also set the Auto- Size property of these containers, so that they'll show up only if they actually hold a control. To be able to drag a control (an object of any TControl-derived class) into the dock site, simply set its DragKind property to dkDock and its DragMode

property to dmAutomatic. This way, the control can be dragged away from its current position into a new docking container.
To undock a component and move it to a special form, you can set its FloatingDockSiteClass property to TCustomDockForm (to use a predefined stand-alone form with a small caption). All the docking and undocking operations can be tracked by using special events of the component being dragged (OnStartDock and OnEndDock) and the component that will receive the docked control (OnDragOver and OnDragDrop). These docking events are very similar to the dragging events available in earlier versions of Delphi.

2.3.4 The ActionManager Architecture

We have seen that actions and the ActionManager component can play a central role in the development of Delphi applications, since they allow a much better separation of the user interface from the actual code of the application. The user interface, in fact, can now easily change without impacting the code too much. The drawback of this approach is that a programmer has more work to do. To have a new menu item, you need to add the corresponding action first, than move to the menu, add the menu item, and connect it to the action. To solve this issue, and to provider developers and end users with some advanced features, Delphi 6 introduces a brand new architecture, based on the ActionManager component, which largely extends the role of actions. The ActionManager, in fact, has a collection of actions but also a collection of toolbars and menus tied to them. The development of these toolbars and menus is completely visual: you drag actions from a special component editor of the ActionManager to the toolbars to have the buttons you need. Moreover, you can let the end user of your programs do the same operation, and rearrange their own toolbars and menus starting with the actions you provide them.

2.4 Working with Forms

If you've read the previous chapters, you should now be able to use Delphi's visual components to create the user interface of your applications. So let's turn our attention to another central element of development in Delphi: forms. We have used forms since the initial chapters, but I've never described in detail what you can do with a form, which properties you can use, or which methods of the TForm class are particularly interesting.

2.4.1 The TformClass

Forms in Delphi are defined by the TForm class, included in the Forms unit of VCL. Of course, there is now a second definition of forms inside VisualCLX. Although I'll mainly refer to the VCL class in this chapter, I'll also try to highlight differences with the cross-platform version provided in CLX.

The TForm class is part of the windowed-controls hierarchy, which starts with the TwinControl (or TWidgetControl) class. Actually, TForm inherits from the *almost complete* TCustomForm, which in turn inherits from TScrollingWinControl (or TScrollingWidget). Having all of the features of their many base classes, forms have a long series of methods, properties, and events. For this reason, I won't try to list them here, but I'd rather present some interesting techniques related to forms throughout this chapter. I'll start by presenting a technique for *not* defining the form of a program at design time, using the TForm class directly, and then explore a few interesting properties of the form class.

2.4.2 Direct Form Input

Having discussed some special capabilities of forms, we'll now move to a very important topic: user input in a form. If you decide to make limited use of components, you might write complex programs as well, receiving input from the mouse and the keyboard.

2.4.2.1 Supervising Keyboard Input

Generally, forms don't handle keyboard input directly. If a user has to type something, your form should include an edit component or one of the other input components. If you want to handle keyboard shortcuts, you can use those connected with menus (possibly using a hidden pop-up menu).

At other times, however, you might want to handle keyboard input in particular ways for a specific purpose. What you can do in these cases is turn on the KeyPreview property of the form. Then, even if you have some input controls, the form's OnKeyPress event will always be activated for any keyboard-input operation. The keyboard input will then reach the destination component, unless you stop it in the form by setting the character value to zero (not the character *0*, but the value 0 of the character set, indicated as #0).

2.4.2.2 Getting Mouse Input

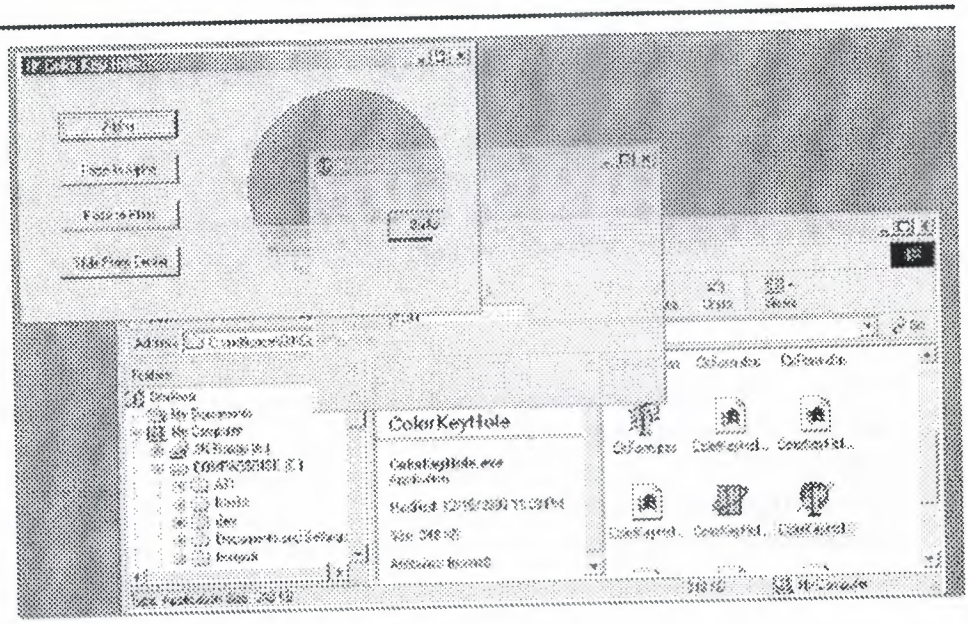
When a user clicks one of the mouse buttons over a form (or over a component, by the way), Windows sends the application some messages. Delphi defines some events you can use to write code that responds to these messages. The two basic events are OnMouseDown, received when a mouse button is clicked, and OnMouseUp, received when the button is released. Another fundamental system message is related to mouse movement; the event is OnMouseMove. Although it should be easy to understand the meaning of the three messages—down, up, and move— the question that might arise is, how do they relate to the OnClick event we have often used up to now?

2.4.3 Unusual Techniques: Alpha Blending, Color Key, and the Animate API

One of the few new features of Delphi related to forms is support for some new Windows APIs regarding the way forms are displayed (not available under Qt/CLX). For a form, *alpha blending* allows you to merge the content of a form with what's behind it on the screen, something you'll rarely need, at least in a business application. The technique is certainly more interesting when applied to bitmap (with the new AlphaBlend and AlphaDIBBlend API functions) than to a form itself. In any case, by setting the AlphaBlend property of a form to True and giving to the AlphaBlendValue property a value lower than 255, you'll be able to see, in transparency, what's behind the form. The lower the AlphaBlendValue, the more the form will *fade*.

You can see an example of alpha blending in Figure 9.7, taken from the CkKeyHole example

FIGURE 9.7: The output of the CkKeyHole, showing the effect of the new TransparentColor and AlphaBlend properties, and also the AnimateWindow API.



This is not the only new Delphi feature in the area of what we can only call *unusual*. The second is the new TransparentColor property, which allows you to indicate a transparent color, which will be replaced by the background, creating a sort of hole in a form. The transparent color is indicated by the TransparentColorValue property.

Finally, you can use a native Windows technique, animated display, which is not directly supported by Delphi (beyond the display of hints).

2.4.4 Position, Size, Scrolling, and Scaling

Once you have designed a form in Delphi, you run the program, and you expect the form to show up exactly as you prepared it. However, a user of your application might have a different screen resolution or might want to resize the form (if this is possible, depending on the border style), eventually affecting the user interface. We've already discussed (mainly in Chapter 7) some techniques related to controls, such as alignment and anchors. Here I want to specifically address elements related to the form as a whole.

2.4.4.1 The Form Position

There are a few properties you can use to set the position of a form. The Position property indicates how Delphi determines the initial position of the form. The default poDesigned value indicates that the form will appear where you designed it and where you use the positional (Left and Top) and size (Width and Height) properties of the form.

2.4.4.2 The Size of a Form and Its Client Area

At design time, there are two ways to set the size of a form: by setting the value of the Width and Height properties or by dragging its borders. At run time, if the form has a resizable border, the user can resize it (producing the OnResize event, where you can perform custom actions to adapt the user interface to the new size of the form).

2.4.4.3 Form Constraints

When you choose a resizable border for a form, users can generally resize the form as they like and also maximize it to full screen. Windows informs you that the form's size has changed with the wm_Size message, which generates the OnResize event. OnResize takes place after the size of the form has already been changed. Modifying the size again in this event (if the user has reduced or enlarged the form too much) would be silly. A preventive approach is better suited to this problem.

2.4.4.4 Scrolling a Form

When you build a simple application, a single form might hold all of the components you need.As the application grows, however, you may need to squeeze in the components, increase the size of the form, or add new forms. If you reduce the space occupied by the components, you might add some capability to resize them at run time, possibly splitting the form into different areas. If you choose to increase the size of the form, you might use scroll bars to let the user move around in a form that is bigger than the screen (or at least bigger than its visible portion on the screen).

2.4.4.5 Scaling Forms

When you create a form with multiple components, you can select a fixed size border or let the user resize the form and automatically add scroll bars to reach the components falling outside the visible portion of the form, as we've just seen. This might also happen because a user of your application has a display driver with a much smaller number of pixels than yours. Instead of simply reducing the form size and scrolling the content, you might want to reduce the size of each of the components at the same time. This automatically happens also if the user has a system font with a different pixel-per-inch ratio than the one you used for development. To address these problems, Delphi has some nice scaling features, but they aren't fully intuitive.

2.4.5 Creating and Closing Forms

Up to now we have ignored the issue of form creation. We know that when the form is created, we receive the OnCreate event and can change or test some of the initial form's properties or fields. The statement responsible for creating the form is in this project's source file:

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

To skip the automatic form creation, you can either modify this code or use the Forms page of the Project Options dialog box .In this dialog box, you can decide whether the form should be automatically created. If you disable the automatic creation, the project's initialization code becomes the following:

```
begin
Applications.Initialize;
Application.Run;
end.
```

Old and New Creation Orders

Now the question is whether these custom operations are executed before or after the OnCreate event is fired. The answer depends on the value of the OldCreateOrder property of the form, introduced in Delphi 4 for backward compatibility with earlier versions of Delphi. By default, for a new project, all of the code in the constructor is executed before the OnCreate event handler. In fact, this event handler is not activated by the base class constructor but by its AfterConstruction method, a sort of constructor introduced for compatibility with C++Builder.

To study the creation order and the potential problems, you can examine the CreatOrd program. This program has an OnCreate event handler, which creates a list box control dynamically. The constructor of the form can access this list box or not, depending on the value of the OldCreateOrder property.

2.4.6 Dialog Boxes and Other Secondary Forms

When you write a program, there is really no big difference between a dialog box and another secondary form, aside from the border, the border icons, and similar user-interface elements you can customize.

What users associate with a dialog box is the concept of a *modal window*—a window that takes the focus and must be closed before the user can move back to the main window. This is true for message boxes and usually for dialog boxes, as well. However, you can also have nonmodal —or *modeless*—dialog boxes. So if you think that dialog boxes are just modal forms, you are on the right track, but your description is not precise. In Delphi (as in Windows), you can have modeless dialog boxes and modal forms. We have to consider two different elements:

- The form's border and its user interface determine whether it looks like a dialog box.
- The use of two different methods (Show or ShowModal) to display the secondary form determines its behavior (modeless or modal).

2.4.7 Windows Common Dialogs

We have already used some of these dialog boxes in several examples in the previous chapters, so you are probably familiar with them. Basically, you need to put the corresponding component on a form, set some of its properties, run the dialog box (with the Execute method, returning a Boolean value), and retrieve the properties that have been set while running it. To help you experiment with these dialog boxes, We've built the CommDlg test program. What we want to do is simply highlight some key and nonobvious features of the common dialog boxes, and let you study the source code of the example for the details:

- The Open Dialog Component can be customized by setting different file extensions filters, using the Filter property, which has a handy editor and can be assigned directly with a string like Text File (*.txt)|*.txt. Another handy feature is to let the dialog check whether the extension of the selected file matches the default extension, by checking the ofExtensionDifferent flag of the Options property after executing the dialog. Finally, this dialog allows multiple selections by setting its

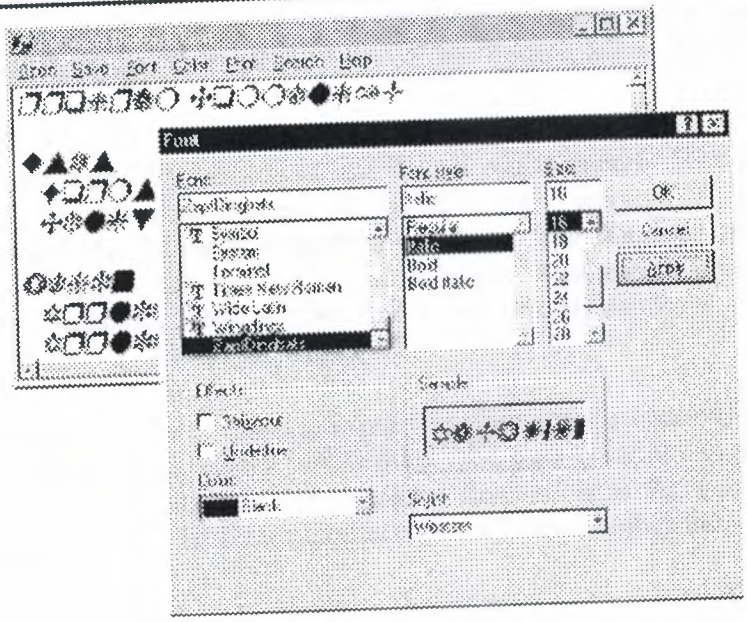
ofAllowMultiSelect option. In this case you can get the list of the selected files by looking at the Files string list property.

- The SaveDialog component is used in similar ways and has similar properties, although you cannot select multiple files, of course.

- The OpenPictureDialog and SavePictureDialog components provide similar features but have a customized form, which shows a preview of an image. Of course, it makes sense to use them only for opening or saving graphical files.

- The FontDialog component can be used to show and select from all types of fonts, fonts useable on both the screen and a selected printer (WYSIWYG), or only TrueType fonts.

FIGURE 9.15:
The Font selection dialog box with an Apply button



- The ColorDialog component is used with different options, to show the dialog fully open at first or to prevent it from opening fully. These settings are the cdFullOpen or cdPreventFullOpen values of the Options property.

- The Find and Replace dialog boxes are truly modeless dialogs, but you have to implement the find and replace functionality yourself, as I've partially done in the CommDlg example. The custom code is connected to the buttons of the two dialog boxes by providing the OnFind and OnReplace events.

2.4.8 A Parade of Message Boxes

The Delphi message boxes and input boxes are another set of predefined dialog boxes. There are many Delphi procedures and functions you can use to display simple dialog boxes:

- The MessageDlg function shows a customizable message box, with one or more buttons and usually a bitmap. The MessageDlgPos function is similar to the MessageDlg function, but the message box is displayed in a given position, not in the center of the screen.

- The ShowMessage procedure displays a simpler message box, with the application name as the caption and just an OK button. The ShowMessagePos procedure does the same, but you also indicate the position of the message box. The ShowMessageFmt procedure is a variation of ShowMessage, which has the same parameters as the Format function. It corresponds to calling Format inside a call to ShowMessage.

- The MessageBox method of the Application object allows you to specify both the message and the caption; you can also provide various buttons and features. This is a simple and direct encapsulation of the MessageBox function of the Windows API, which passes as a main window parameter the handle of the Application object. This handle is required to make the message box behave like a modal window.

- The InputBox function asks the user to input a string. You provide the caption, the query, and a default string. The InputQuery function asks the user to input a string, too. The only difference between this and the InputBox function is in the syntax. The InputQuery function has a Boolean return value that indicates whether the user has clicked OK or Cancel.

2.4.9 About Boxes and Splash Screens

Applications usually have an About box, where you can display information, such as the version of the product, a copyright notice, and so on. The simplest way to build an About box is to use the MessageDlg function. With this method, you can show only a limited amount of text and no special graphics.

Therefore, the usual method for creating an About box is to use a dialog box, such s the one generated with one of the Delphi default templates. In this about box you might want to add some code to display system information, such as the version of Windows or the amount of free memory, or some user information, such as the registered user name.

2.5 The Architecture of Delphi Applications

Although together we've built Delphi applications since the beginning of the book, we've never really focused on the structure and the architecture of an application built with Delphi's class library. For example, there hasn't been much coverage about the global Application object, about techniques for keeping tracks of the forms we've created, about the flow of messages in the system, and other such elements.

In the last chapter you saw how to create applications with multiple forms and dialog boxes, but we haven't discussed how these forms can be related one to the other, how can you share similar features of forms, and how you can operate on multiple similar forms in a coherent way. All of this is the ambitious goal of this chapter, which covers both basic and advanced techniques, including visual form inheritance, the use of frames, and MDI development, but also the use of interfaces for building complex hierarchies of form classes.

2.5.1 The ApplicationObject

We've already mentioned the Application global object on multiple occasions, but as in this chapter we are focusing on the 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 you 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 you to assign handlers at design time, the advantage of this component is that it allows for multiple handlers. If you 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 you 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 exception handler we used in Chapter 3, 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 your program are behind those of other programs, licking one window in your 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. 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 you don't need to write one in Delphi because the Application object provides a default one. If this is the main role of the Application object, it manages few other interesting areas as well:

- . Hints
- . The help system, which in Delphi 6 includes the ability to define the type of help viewer (something not covered in detail in this book)
- . Application activation, minimize, and restore
- . A global exceptions handler, as discussed in Chapter 3 in the ErrorLog example
- . General application information, including the MainForm, executable file name and path (ExeName), the Icon, and the Title displayed in the Windows taskbar and when you scan the running applications with the Alt+Tab keys

2.5.2 Events, Messages, and Multitasking in Windows

To understand how Windows applications work internally, we need to spend a minute discussing how multitasking is supported in this environment. We also need to understand the role of timers (and the Timer component) and of background (or *idle*) computing. In short, we need to delve deeper into the event-driven structure of Windows and its multitasking support. Because this is a book about *Delphi* programming,

2.5.2.1 Event-Driven Programming

The basic idea behind event-driven programming is that specific events determine the control flow of the application. A program spends most of its time waiting for these events

and provides code to respond to them. For example, when a user clicks one of the mouse buttons, an event occurs.
A message describing this event is sent to the window currently under the mouse cursor. The program code that responds to events for that window will receive the event, process it, and respond accordingly. When the program has finished responding to the event, it returns to a waiting or "idle" state.

As this explanation shows, events are serialized; each event is handled only after the previous one is completed. When an application is executing event-handling code (that is, when it is not waiting for an event), other events for that application have to wait in a message queue reserved for that application (unless the application uses multiple threads). When an application has responded to a message and returned to a waiting state, it becomes the last in the list of programs waiting to handle additional messages. In every version of Win32 (9*x*, NT, Me, and 2000), after a fixed amount of time has elapsed, the system interrupts the current application and immediately gives control to the next one in the list. The first program is resumed only after each application has had a turn. This is called preemptive multitasking.

So, an application performing a time-consuming operation in an event handler doesn't prevent the system from working properly, but is generally unable even to repaint its own windows properly, with a very nasty effect. If you've never experienced this problem, try for yourself: Write a time-consuming loop executed when a button is pressed, and try to move the form or move another window on top of it. The effect is really annoying. Now try adding the call Application.ProcessMessages within the loop, and you'll see that the operation becomes much slower, but the form will be immediately refreshed. If an application has responded to its events and is waiting for its turn to process messages, it has no chance to regain control until it receives another message (unless it uses multithreading). This is a reason to use timers, a system component that will send a message to your application every time a time interval elapses.

One final note—when you think about events, remember that input events (using the mouse or the keyboard) account for only a small percentage of the total message flow in a Windows application. Most of the messages are the system's internal messages or messages exchanged between different controls and windows. Even a familiar input operation such as clicking a mouse button can result in a huge number of messages, most of which are internal Windows messages. You can test this yourself by using the WinSight utility included in Delphi. In WinSight, choose to view the Message Trace, and select the messages for all of the windows. Select Start, and then perform some normal operations with the mouse. You'll see hundreds of messages in a few seconds.

2.5.2.2 Windows Message Delivery

Before looking at some real examples, we need to consider another key element of message handling. Windows has two different ways to send a message to a window:

- The PostMessage API function is used to place a message in the application's message queue. The message will be handled only when the application has a

chance to access its message queue (that is, when it receives control from the system), and only after earlier messages have been processed.
This is an asynchronous call, since you do not know when the message will actually be received.

- The SendMessage API function is used to execute message-handler code immediately. SendMessage bypasses the application's message queue and sends the message directly to a target window or control. This is a synchronous call. This function even has a return value, which is passed back by the message-handling code. Calling SendMessage is no different than directly calling another method or function of the program. The difference between these two ways of sending messages is similar to that between mailing a letter, which will reach its destination sooner or later, and ending a fax, which goes

immediately to the recipient. Although you will rarely need to use these low-level unctions in Delphi this description should help you determine which one to use if you do need to write this type of code.

2.5.2.3 Background Processing and Multitasking

Suppose that you need to implement a time-consuming algorithm. If you write the algorithm as a response to an event, your application will be stopped completely during all the time it takes to process that algorithm. To let the user know that something is being processed, you can display the hourglass cursor, but this is not a user-friendly solution. Win32 allows other programs to continue their execution, but the program in question will freeze; it won't even update its own user interface if a repaint is requested. In fact, while the algorithm is executing, the application won't be able to receive and process any other messages, including the paint messages.

The simplest solution to this problem is to call the ProcessMessages method of the Application object many times within the algorithm, usually inside an internal loop. This call stops the execution, allows the program to receive and handle a message, and then resumes execution. The problem with this approach, however, is that while the program is paused to accept messages, the user is free to do any operation and might again click the button or press the keystrokes that started the algorithm. To fix this, you can disable the buttons and commands you don't want the user to select, and you can display the hourglass cursor (which technically doesn't prevent a mouse click event, but it does suggest that the user should wait before doing any other operation). An alternative solution is to split the algorithm into smaller pieces and execute each of them in turn, letting the application respond to pending messages in between processing the pieces. We can use a timer to let the system notify us once a time interval has elapsed. Although you can use timers to implement some form of background computing, this is far from a good solution. A slightly better technique would be to execute each step of the program when the Application object receives the OnIdle event. The difference between calling ProcessMessages and using the

OnIdle events is that by calling ProcessMessages, you will give your code more processing time than with the OnIdle approach.
Calling ProcessMessages is a way to let the system perform other operations while your program is computing; using the OnIdle event is a way to let your application perform background tasks when it doesn't have pending requests from the user.

2.5.3 Checking for a Previous Instance of an Application

One form of multitasking is the execution of two or more instances of the same application. Any application can generally be executed by a user in more than one instance, and it needs to be able to check for a previous instance already running, in order to disable this default behavior and allow for one instance at most. This section demonstrates several ways of implementing such a check, allowing me to discuss some interesting Windows programming techniques.

2.5.4 Creating MDI 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.

MDI in Windows: A Technical Overview

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:

- The main window of the application acts as a frame or a container.
- 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.
- There are multiple child windows, of the same kind 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 frame window.

2.5.5 Frame and Child Windows in Delphi

Delphi makes the development of MDI applications easy, even without using the MDI Application template available in Delphi (see the Applications page of the File . New dialog box). You only need to build at least two forms, one with the FormStyle property set to fsMDIForm and the other with the same property set to fsMDIChild. Set these two properties in a simple program and run it, and you'll see the two forms nested in the typical MDI style.

Generally, however, the child form is not created at startup, and you 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.

2.5.6 MDI Applications with Different Child Windows

A common approach in complex MDI applications is to include child windows of different kinds (that is, based on different child forms). I will build a new example, called MdiMulti, to highlight some problems you may encounter with this approach. This example has two different types of child forms. The first type will host a circle drawn in the position of the last mouse click, while the second will contain a bouncing square. Another feature I'll add to the main form is a custom background obtained by painting a tiled image in it.

2.5.6.1 Child Forms and Merging Menus

The first type of child form can display a circle in the position where the user clicked one of the mouse buttons. Figure 10.5 shows an example of the output of the MdiMulti program. The program includes a Circle menu, which allows the user to change the color of the surface of the circle as well as the color and size of its border. What is interesting here is that to program the child form, we do not need to consider the existence of other forms or of the frame window. We simply write the code of the form, and that's all. The only special care required is for the menus of the two forms.

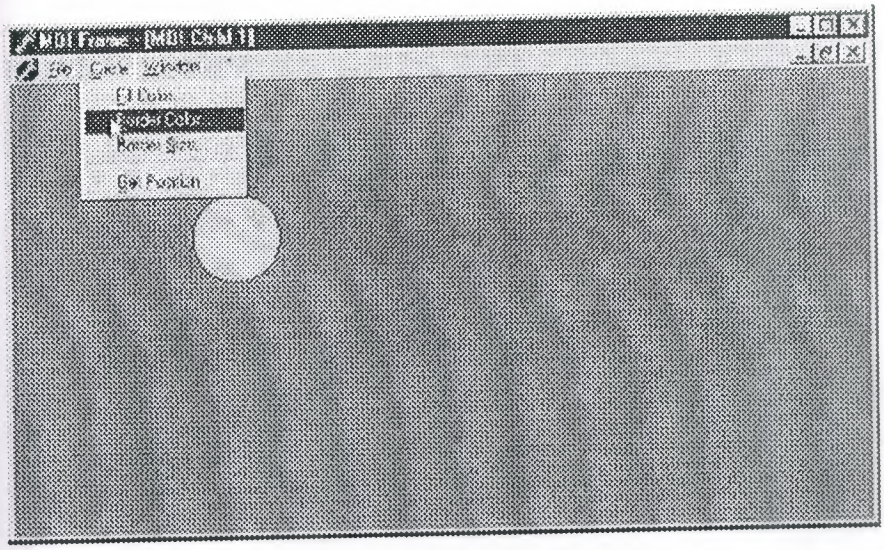
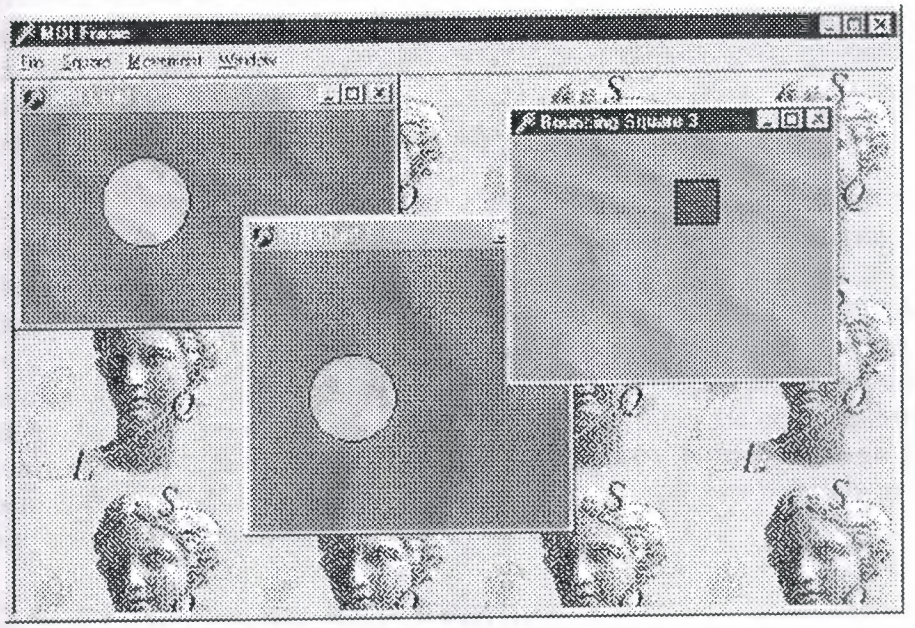


FIGURE10.5:
The output of the MdiMulti example, with a child window that displays circles

2.5.6.2 The Main Form

Now we need to integrate the two child forms into an MDI application. The File pull-down menu here has two separate New menu items, which are used to create a child window of either kind. The code uses a single child window counter. As an alternative, you could use two different counters for the two kinds of child windows. The Window menu uses the predefined MDI actions.

As soon as a form of this kind is displayed on the screen, its menu bar is automatically merged with the main menu bar. When you select a child form of one of the two kinds, the menu bar changes accordingly. Once all the child windows are closed, the original menu bar of the main form is reset. By using the proper menu group indexes, we let Delphi accomplish everything automatically, as you can see in Figure 10.6.



2.5.7 Visual Form Inheritance

When you need to build two or more similar forms, possibly with different event handlers, you can use dynamic techniques, hide or create new components at run time, change event handlers, and use if or case statements. Or you can apply the object-oriented techniques, thanks to visual form inheritance. In short, instead of creating a form based on TForm, you can inherit a form from an existing one, adding new components or altering the properties of the existing ones. But what is the real advantage of visual form inheritance?

Well, this mostly depends on the kind of application you are building. If it has multiple forms, some of which are very similar to each other or simply include common elements, then you can place the common components and the common event handlers in the base form and add the specific behavior and components to the subclasses. For example, if you prepare a standard parent form with a toolbar, a logo, default sizing and closing code, and the handlers of some Windows messages, you can then use it as the parent class for each of the forms of an application. You can also use visual form inheritance to customize an application for different clients, without duplicating any source code or form definition code; just inherit the specific versions for a client from the standard forms. Remember that the main advantage of visual inheritance is that you can later change the original form and automatically update all the derived forms. This is a well-known advantage of inheritance in object-oriented programming languages. But there is a beneficial side effect: polymorphism. You can add a virtual method in a base form and override it in a subclassed form. Then you can refer to both forms and call this method for each of them.

Inheriting from a Base Form

The rules governing visual form inheritance are quite simple, once you have a clear idea of what inheritance is. Basically, a subclass form has the same components as the parent form as well as some new components. You cannot remove a component of the base class, although (if it is a visual control) you can make it invisible. What's important is that you can easily change properties of the components you inherit.

Notice that if you change a property of a component in the inherited form, any modification of the same property in the parent form will have no effect. Changing other properties of the component will affect the inherited versions, as well. You can resynchronize the two property values by using the Revert to Inherited local menu command of the Object Inspector. The same thing is accomplished by setting the two properties to the same value and recompiling the code. After modifying multiple properties, you can resynchronize them all to the base version by applying the Revert to Inherited command of the component's local menu. Besides inheriting components, the new form inherits all the methods of the base form, including the event handlers. You can add new handlers in the inherited form and also override existing handlers.

2.5.8 Understanding Frames

Chapter 1 briefly discussed frames, which were introduced in Delphi 5. We've seen that you can create a new frame, place some components in it, write some event handlers for the components, and then add the frame to a form. In other words, a frame is similar to a form, but it defines only a portion of a window, not a complete window. This is certainly not a feature worth a new construct. The totally new element of frames is that you can create multiple instances of a frame at design time, and you can modify the class and the instance at the same time. This makes frames an effective tool for creating customizable composite controls at design time, something close to a visual component-building tool.

In visual form inheritance you can work on both a base form and a derived form at design time, and any changes you make to the base form are propagated to the derived one, unless this overrides some property or event. With frames, you work on a class (as usual in Delphi), but the difference is that you can also customize one or more instances of the class at design time. When you work on a form, you cannot change a property of the TForm1 class for the Form1 object at design time. With frames, you can.

Once you realize you are working with a class and one or more of its instances at design time, there is nothing more to understand about frames. In practice, frames are useful when you want to use the same group of components in multiple forms within an application. In this case, in fact, you can customize each of the instances at design time. Wasn't this already possible with component templates? It was, but component templates were based on the concept of copying and pasting some components and their code. There was no way to change the original definition of the template and see the effect in every place it was used. That is what happens with frames (and in a different way with visual form inheritance); changes to the original version (the class) are reflected in the copies (the instances).

Frames and Pages

When you have a dialog box with many pages full of controls, the code underlying the form becomes very complex because all the controls and methods are declared in a single form. Also, creating all these components (and initializing them) might result in a delay in the display of the dialog box. Frames actually don't reduce the construction and initialization time of equivalently loaded forms; quite the contrary, as loading frames is more complicated for the streaming system than loading simple components. However, using frames you can load only the visible pages of a multipage dialog box, reducing the *initial* load time, which is what the user perceives.

Frames can solve both of these issues. First, you can easily divide the code of a single complex form into one frame per page. The form will simply host all of the frames in a PageControl. This certainly helps you to have simpler and more focused units and makes it simpler to reuse a specific page in a different dialog box or application. Reusing a single page of a Page- Control without using a frame or an embedded form, in fact, is far from simple. As an example of this approach I've built the FramePag example, which has some frames placed inside the three pages of a PageControl, as you can see in Figure 10.12. All of the frames are aligned to the client area, using the entire surface of the tab sheet (the page) hosting them. Actually two of the pages have the same frame, but the two instances of the frame have some differences at design time. The frame, called Frame3 in the example, has a list box that is populated with a text file at startup, and has buttons to modify the items in the list and saves them to a file. The filename is placed inside a label, so that you can easily select a file for the frame at design time by changing the Caption of the label.

2.5.9 Base Forms and Interfaces

We have seen that when you need two similar forms inside an application, you can use visual form inheritance to inherit one from the other or both of them from a common ancestor. The advantage of visual form inheritance is that you can use it to inherit the visual definition, the DFM. However, this is not always requested.
At times, you might want several forms to exhibit a common behavior, or respond to the same commands, without having any shared component or user interface elements. Using visual form inheritance with a base form that has no extra components makes little sense to me. I rather prefer defining my own custom form class, inherited from TForm, and then manually editing the form class declarations to inherit from this custom base form class instead of the standard one. If all you need is to define some shared methods, or override TForm virtual methods in a consistent way, defining custom form classes can be a very good idea.

2.6 Creating Components

While most Delphi programmers are probably familiar with using existing components, at times it can also be useful to write our own components or to customize existing ones. One of the most interesting aspects of Delphi is that creating components is simple. For this reason, even though this book is intended for Delphi application programmers and not Delphi tool writers, this chapter will cover the topic of creating components and introduce Delphi add-ins, such as component and property editors.

2.6.1 Extending the Delphi Library

Delphi components are classes, and the Visual Components Library (VCL) is the collection of all the classes defining Delphi components. Each time you add a new package with some components to Delphi, you actually extend VCL with a new class. This new class will be derived from one of the existing component-related classes or the generic TComponent class, adding new capabilities to those it inherits.

You can derive a new component from an existing component or from an *abstract component class*—one that does not correspond to a usable component. The VCL hierarchy includes many of these intermediate classes (often indicated with the TCustom prefix in their name) to let you choose a default behavior for your new component and to change its properties.

Component Packages

Components are added to component packages. Each component package is basically a DLL (a dynamic link library) with a BPL extension (which stands for Borland Package Library). Packages come in two flavors: design-time packages used by the Delphi IDE and run-time packages optionally used by applications. The design-only or run-only package option determines the package's type. When you attempt to install a package, the IDE checks whether it has the design-only or run-only flags, and decides whether to let the user install the package and whether it should be added to the list of run-time packages. Since there are two nonexclusive options, each with two possible states, there are four different kinds of component packages—two main variations and two special cases:

- Design-only component packages can be installed in the Delphi environment. These packages usually contain the design-time parts of a component, such as its property editors and the registration code. Often they can also contain the components themselves, although this is not the most professional approach. The code of the components of a design-only package is usually statically linked into the executable

file, using the code of the corresponding Delphi Compiled Unit (DCU) files. Keep in mind, however, that it is also technically possible to use a design-only package as a run-time package.

- Run-only component packages are used by Delphi applications at run time. They cannot be installed in the Delphi environment, but they are automatically added to the list of run-time packages when they are required by a design-only package you install. Runonly packages usually contain the code of the component classes, but no design-timesupport (this is done to minimize the size of the component libraries you ship along with your executable file). Run-only packages are important because they can be freely distributed along with applications, but other Delphi programmers won't be able to install them in the environment to build new programs.
- Plain component packages (having neither the design-only nor the run-only option set) cannot be installed and will not be added to the list of run-time packages automatically. This might make sense for utility packages used by other packages, but they are certainly rare.
- Packages with both flags set can be installed and are automatically added to the list of run-time packages. Usually these packages contain components requiring little or no design-time support (apart from the limited component registration code). Keep in mind, however, that users of applications built with these packages can use them for their own development.

2.6.2 A Complex Graphical Component

The graphical component I want to build is an arrow component. You can use such a component to indicate a flow of information, or an action, for example. This component is quite complex, so I'll show you the various steps instead of looking directly at the complete source code.

The component I've added to the MdPack package on the CD is only the final version of this process, which will demonstrate several important concepts:

- The definition of new enumerated properties, based on custom enumerated data types.
- The use of properties of TPersistent-derived classes, such as TPen and TBrush, and the issues related to their creation and destruction, and to handling their OnChange events internally in our component.
- The implementation of the Paint method of the component, which provides its user interface and should be generic enough to accommodate all the possible values of the various properties, including its Width and Height. The Paint method plays a substantial role in this graphical component.
- The definition of a custom event handler for the component, responding to user input (in this case, a double-click on the point of the arrow). This will require direct handling of Windows messages and the use of the Windows API for graphic regions.
- The registration of properties in Object Inspector categories and the definition of a custom category.

2.6.3 Customizing Windows Controls

One of the most common ways of customizing existing components is to add some predefined behavior to their event handlers. Every time you need to attach the same event handler to components of different forms, you should consider adding the code of the event right into a subclass of the component. An obvious example is that of edit boxes accepting only numeric input. Instead of attaching to each of them a common OnChar event handler, we can define a simple new component. This component, however, won't handle the event; events are for component users only. Instead, the component can either handle the Windows message directly or override a method, as described in the next two sections.

2.6.3.1 Overriding Message Handlers: The Numeric Edit Box

To customize an edit box component to restrict the input it will accept, all you need to do is handle the wm_CharWindows messages that occur when the user presses any but a few specific keys (namely, the numeric characters).

2.6.3.2 Overriding Dynamic Methods: The Sound Button

Our next component, TMdSoundButton, plays one sound when you press the button and another sound when you release it. The user specifies each sound by modifying two String properties that name the appropriate WAV files for the respective sounds. Once again, we need to intercept and modify some system messages (wm_LButtonDown and wm_LButtonUp), but instead of handling the messages by writing a new message-response method, we'll override the appropriate *second-level* handlers.

2.6.3.3 Handling Internal Messages: The Active Button

The Windows interface is evolving toward a new standard, including components that become highlighted as the mouse cursor moves over them. Delphi provides similar support in many of its built-in components, but what does it take to mimic this behavior for a simple button? This might seem a complex task to accomplish, but it is not.

2.6.3.4 Component Messages and Notifications

To build the ActiveButton component, I've used two internal Delphi component messages, as indicated by their *cm* prefix. These messages can be quite interesting, as the example highlights, but they are almost completely undocumented by Borland. There is also a second group of internal Delphi messages, indicated as component notifications and distinguished by their *cn* prefix. I don't have enough space here to discuss each of them or provide a detailed analysis; browse the VCL source code if you want to learn more.

2.6.4 A Nonvisual Dialog Component

The next component we'll examine is completely different from the ones we have seen up to now. After building window-based controls and simple graphic components, I'm now going to build a nonvisual component.

The basic idea is that forms are components. When you have built a form that might be particularly useful in multiple projects, you can add it to the Object Repository or make a component out of it. The second approach is more complex than the first, but it makes using the new form easier and allows you to distribute the form without its source code. As an example, I'll build a component based on a custom dialog box, trying to mimic as much as possible the behavior of standard Delphi dialog box components.

The first step in building a dialog box in a component is to write the code of the dialog box itself, using the standard Delphi approach. Just define a new form and work on it as usual. When a component is based on a form, you can almost visually design the component. Of course, once the dialog box has been built, you have to define a component around it in a nonvisual way.

2.6.5 Defining Custom Actions

Besides defining custom components, you can define and register new standard actions, which will be made available in the Action Editor of the Action List component. Creating new actions is not complex. You have to inherit from the TAction class and override some of the methods of the base class.

There are basically three methods to override. The HandlesTarget function returns whether the action object wants to handle the operation for the current target, which is by default the control with the focus. The UpdateTarget procedure can set the user interface of the controls connected with the action, eventually disabling the action if the operation is currently not available. Finally, you can implement the ExecuteTarget method to determine the actual code to execute, so that the user can simply select the action and doesn't have to implement it.

2.6.6 Writing Property Editors

Writing components is certainly an effective way to customize Delphi, helping developers to build applications faster without requiring a detailed knowledge of low-level techniques. The Delphi environment is also quite open to extensions. In particular, you can extend the Object Inspector by writing custom property editors and to extend the Form Designer by adding component editors.

2.6.7 Writing a Component Editor

Using property editors allows the developer to make a component more user-friendly. In fact, the Object Inspector represents one of the key pieces of the user interface of the Delphi environment, and Delphi developers use it quite often. However, there is a second approach you can adopt to customize how a component interacts with Delphi: write a custom component editor.

Just as property editors extend the Object Inspector, component editors extend the Form Designer. In fact, when you right-click within a form at design time, you see some default menu items, plus the items added by the component editor of the selected component. Examples of these menu items are those used to activate the Menu Designer, the Fields Editor, the Visual Query Builder, and other editors of the environment. At times, displaying these special editors becomes the default action of a component when it is double-clicked. Common uses of component editors include adding an About box with information about the developer of the component, adding the component name, and providing specific wizards to set up its properties.

2.7 Libraries and Packages

Windows executable files come in two flavors: *programs* and *dynamic link libraries* (DLLs). When you write a Delphi application, you typically generate a program file, an EXE. However, Delphi applications often use calls to functions stored in DLLs. Each time you call a Windows API function directly, you actually access a DLL. Delphi also allows programmers to use run-time DLLs for the component library. When you create a package, you basically create a DLL. Delphi can also generate plain dynamic link libraries. The New page of the Object Repository includes a DLL skeleton generator, which generates very few lines of source code.

It is very simple to generate a DLL in the Delphi environment. However, some problems arise from the nature of DLLs. Writing a DLL in Windows is not always as simple as it seems, because the DLL and the calling program need to agree on calling conventions, parameter types, and other details. This chapter covers the basics of DLL programming from the Delphi point of view and provides some simple examples of what you can place in a Delphi DLL. While discussing the examples, I'll also refer to other programming languages and environments, simply because one of the key reasons for writing a procedure in a DLL is to be able to call it from a program written in a different language.

The second part of the chapter will focus on a specific type of dynamic link library, the Delphi *package*. These packages are not as easy to use as they first seem, and it took Delphi programmers some time to figure out how to take advantage of them effectively. Here I'm going to share with you some of these interesting tips and techniques.

2.7.1 The Role of DLLs in Windows

Before delving into the development of DLLs in Delphi and other programming languages, I'll give you a short technical overview of DLLs in Windows, highlighting the key elements. We will start by looking at dynamic linking, then see how Windows uses DLLs, explore the differences between DLLs and executable files, and end with some general rules to follow when writing DLLs.

2.7.1.1 What Is Dynamic Linking?

First of all, you need to understand the difference between static and dynamic linking of functions or procedures. When a subroutine is not directly available in a source file, the compiler adds the subroutine to an internal table, which includes all external symbols. Of course, the compiler must have seen the declaration of the subroutine and know about its parameters and type, or it will issue an error.

After compilation of a normal—*static*—subroutine, the linker fetches the subroutine's compiled code from a Delphi compiled unit (or static library) and adds it to the executable. The resulting EXE file includes all the code of the program and of the units involved. The Delphi linker is smart enough to include only the minimum amount of code of the units used by the program and to link only the functions and methods that are actually used.

2.7.1.2 What Are DLLs For?

Now that you have a general idea of how DLLs work, we can focus on the reasons for using them in Windows:

- If different programs use the same DLL, the DLL is loaded in memory only once, thus saving system memory. DLLs are mapped into the private address space of each process (each running application), but their code is loaded in memory only once.
- You can provide a different version of a DLL, replacing the current one. If the subroutines in the DLL have the same parameters, you can run the program with the new version of the DLL without having to recompile it. If the DLL has new subroutines, it doesn't matter at all. Problems might arise only if a routine in the older version of the DLL is missing in the new one. Problems also arise if the new DLL does not implement the functions in a manner that is compatible with the operation of the old DLL.

These generic advantages apply in several cases. If you have a complex algorithm, or some complex forms required by several applications, you can store them in a DLL. This will let you reduce the executable's size and save some memory when you run several programs using those DLLs at the same time.

The second advantage is particularly applicable to complex applications. If you have a very big program that requires frequent updates and bug fixes, dividing it into several executables and DLLs allows you to distribute only the changed portions instead of one single large executable. This makes sense for Windows system libraries in particular: You generally don't need to recompile your code if Microsoft provides you an updated version of Windows system libraries—for example, in a new version of the operating system.

Another common technique is to use DLLs to store nothing except resources. You can build different versions of a DLL containing strings for different languages and then change the language at run time, or you can prepare a library of icons and bitmaps and then use them in different applications. The development of language-specific versions of a program is particularly important, and Delphi includes support for it through the Integrated Translation Environment (ITE) and the external environment, which are more advanced topics than I have room to go into.

Another key advantage is that DLLs are independent of the programming language. Most Windows programming environments, including most macro languages in end-user applications, allow a programmer to call a subroutine stored in a DLL. This means you can build a DLL in Delphi and call it from Visual Basic, Excel, and many other Windows applications.

Understanding System DLLs

The Windows system DLLs take advantage of all the key benefits of DLLs I've just highlighted. For this reason, it is worth examining them. First of all, Windows has many system DLLs. The three central portions of Windows—Kernel, User, and GDI—are implemented using DLLs (with 32-bit or 16-bit code depending on the OS version). Other system DLLs are operating-system extensions, such as the DLLs for common dialog boxes and controls, OLE, device drivers, fonts, ActiveX controls, and hundreds of others.

Dynamic system libraries are one of the technical foundations of the Windows operating systems. Since each application uses the system DLLs for anything from creating a window to producing output, every program is linked to those DLLs. When you change your printer, you do not need to rebuild your application or get a new version of the Windows GDI library, which manages the printer output. You only need to provide a specific driver, which is a DLL called by GDI to access your printer. Each printer type has its own driver DLL, which makes the system extremely flexible.

From a different point of view, version handling is important for the system itself. If you have an application compiled for Windows 95, you should be able to run it on Windows Me, Windows 2000, and (possibly) future versions of Windows, but the application might behave differently, as each version of Windows has different system code.

The system DLLs are also used as system-information archives. For example, the User DLL maintains a list of all the active windows in the system, and the GDI DLL holds the list of active pens, brushes, icons, bitmaps, and the like. The free memory area of these two

system DLLs is usually called "free system resources," and the fact that it is limited plays a very important role in Windows versions still relying on 16-bit code, such as the Windows 9*x* family. On NT platforms, GDI and User resources are limited only by available system memory.

2.7.2 Using Delphi Packages

In Delphi, component packages are an important type of DLL. Packages allow you to bundle a group of components and then link the components either statically (adding their compiled code to the executable file of your application) or dynamically (keeping the component code in a DLL, the run-time package that you'll distribute along with your program). In the last chapter, you saw how to build a package. Now I want to underline some advantages and disadvantages of the two forms of linking for a package.

Package Versioning

A very important and often misunderstood element is the distribution of updated packages. When you update a DLL, you can ship the new version, and the executable programs requiring this DLL will generally still work (unless you've removed existing exported functions or changed some of their parameters).

When you distribute a Delphi package, however, if you update the package and modify the interface portion of any unit of the package, you might need to recompile all the applications that use the package. This is required if you add methods or properties to a class, but not if you add new global symbols (or modify anything not used by client applications). There is no problem at all for changes affecting only the implementation section of the package's units. A DCU file in Delphi has a version tag based on its timestamp and a checksum computed from the interface portion of the unit.
When you change the interface portion of a unit, every other unit based on it should be recompiled. The compiler compares the timestamp and checksum of the unit of previous compilations with the new timestamp and checksum, and decides whether the dependent unit must be recompiled. This is why you have to recompile each unit when you get a new version of Delphi, which has modified system units. A package is a collection of units. In Delphi 3, a checksum of the package, obtained from the checksum of the units it contains and the checksum of the packages it requires, was added as an extra entry function to the package library, so that any executable based on an older version of the package would fail at startup.

Delphi 4 and following versions have relaxed the run-time constraints of the package. The design-time constraints on DCU files remain identical, though. The checksum of the packages is not checked anymore, so you can directly modify the units that are part of a package and deploy a new version of the package to be used with the existing executable file. Since methods are referenced by name, you cannot remove any existing method. You cannot even change its parameters, because of name-mangling techniques specifically added to the packages to protect against changes in parameters.

Removing a method referenced from the calling program will stop the program during the loading process. If you make other changes, however, the program might fail unexpectedly during its execution. For example, if you replace a component placed on a form compiled in a package with a similar component, the calling program might still able to access the one in that memory location, although it is now a different component!

If you decide to follow this treacherous road of changing the interface of units in a package without recompiling all the programs that use it, you should at least limit your changes. When you add new properties or nonvirtual methods to the form, you should be able to maintain full compatibility with existing programs already using the package. Also, adding fields and virtual methods might affect the internal structure of the class, leading to problems with existing programs that expect a different class data and virtual method table (VMT) layout. Of course, this applies to the binary compatibility between the EXE and the BPL (Borland Package Library).

2.7.3 Forms Inside Packages

We've already discussed (in Chapter 11, "Creating Components") the use of component packages in Delphi applications. As I'm discussing the use of packages and DLLs for partitioning an application, here I'll start discussing the development of packages holding forms.

We've seen earlier in this chapter that you can use forms inside DLLs, but this sometimes causes a few problems. If you are building both the library and the executable file in Delphi, using packages results in a much better and cleaner solution.

At first sight, you might believe that Delphi packages are a way to distribute components to be installed in the environment. Instead, you can use packages as a way to structure your code but, unlike DLLs, retain the full power of Delphi's OOP. Consider this: A package is a collection of compiled units and your program uses several units. The units the program refers to will be compiled inside the executable file, unless you ask Delphi to place them inside a package.

2.7.4 Packages Versus DLLs

In the preceding section, we've seen that using packages is a fine alternative to using DLLs for sharing compiled code among multiple Delphi applications or splitting a large executable into multiple (and partially independent) modules. As a summary, here are a few of the differences between the two approaches:

- DLLs are collections of functions; packages can easily "export" classes and objects.

- Dynamically loading a DLL implies losing any safety in the function call, in case you pass the wrong parameters. Dynamically loading packages requires some extra coding as well, but is definitely simpler and safer, particularly if you use interfaces.
- Packages force you to use the VCL run-time package for the application, although even when using DLLs, run-time packages help solve quite a few difficulties (as we'll discuss shortly).
- DLLs can be used across programming languages and development environments, but packages are limited to Delphi and C++Builder. If you need libraries in a Delphi-only environment, packages are the native solution and should generally be preferred. Using DLLs also accounts for a few extra troubles that you can partially solve by letting the DLLs share run-time packages. The following sections discuss the problem briefly, as this is not the recommended approach anyway.

Executables and DLLs Sharing the VCL Packages

In the FormDLL example, we faced a problem: When you place forms inside a DLL, you don't get the proper behavior for the flat buttons even if you synchronize the two application objects. Moreover, both the executable file and the DLL contain the compiled code of the VCL library, leading to useless duplication. As discussed earlier, the simplest solution to this issue is to use a package instead of a DLL.

Another solution is to keep the DLL in its format, but let it use run-time packages, so that no global objects will be duplicated between the executable and the library. In this case there will be only one Application object, shared by the program and the DLL, instead of two separate objects, so we don't need the synchronization code any more. Another simplification to the program comes from the fact that the modeless form inside the DLL can communicate back to the main form by accessing the list of the forms (available to the shared global Screen object) or simply using the Application.MainForm property. This is what I've done in the FormDllP example on the CD.

With this approach, you face the risk of having the main form and the form in the DLL not synchronized at all, with two entries in the Taskbar; also, this code still has all the other problems of the first version of the FormDLL example. The problem lies in the fact that when you run the program, the DLL is initialized before the application, so it is the DLL that initializes the Forms unit of the VCL. Within a DLL, the VCL creates the Application object but doesn't create the corresponding window.

There are two radically different approaches to this initialization issue: One is to change the initialization order by loading the DLL dynamically after the application has started; the second is to add some extra initialization code in the program. None of these techniques provides a better solution than using packages altogether!

Chapter-3
DATABASE PROGRAMMING

3.1 Delphi's Database Architecture

Delphi's support for database applications is one of the key features of the programming environment. Many programmers spend most of their time writing data-access code, which needs to be the most robust portion of a database application. This chapter provides an overview of Delphi's extensive support for database programming.

What you won't find here is a discussion of the theory of database design. I'm assuming that you already know the fundamentals of database design and have already designed the structure of a database. I won't delve into database-specific problems; my goal is to help you understand how Delphi supports database access.

I'll begin with an explanation of the alternatives Delphi offers in terms of data access, and then I'll provide an overview of the database components that are available in Delphi. This chapter includes an overview of the TDataSet class, an in-depth analysis of the TField components, and the use of data-aware controls. The following chapters will provide information on more advanced database programming topics, such as client/server programming, the use of dbGo, dbExpress, and InterBase Express.

3.1.1 Accessing a Database: BDE, dbExpress, and Other Alternatives

In the first few versions of Delphi, the only available technology to access database data was to use the Borland Database Engine (BDE). Starting with Delphi 3, the portion of VCL related to Database access has been restructured to open it up to multiple database access solutions. Delphi 5 saw the introduction of specific sets of components supporting Microsoft's ActiveX Data Objects (ADO) and InterBase Express (IBX). Delphi 6 adds to the picture dbExpress, which is a brand-new cross-platform and database-independent data-access technology provided by Borland with Kylix on Linux and Delphi 6 on Windows.

With all these alternatives, it is easy to get confused on which approach to use. In the following sections I've provided a short description of the key elements of these data-access technologies available in Delphi, trying to suggest in which case you'll want to use each of them.

3.1.1.1 Borland Database Engine (BDE)

The BDE originated with Paradox, well before Delphi existed, and was extended by Borland to support other local databases and many SQL servers. The BDE has direct access to dBASE, Paradox, ASCII, FoxPro, and Access tables. A series of drivers (called SQL Links and available only in Delphi Enterprise) allows access to some SQL servers, including Oracle, Sybase, Microsoft, Informix, InterBase, and DB2 servers. If you need access to a different database, the BDE can also interface with ODBC drivers.

The advantage of using a common database engine is that your application will be portable among different servers of the same category (porting from a local database to an SQL server is generally much more complex). The specific advantages of using the BDE are that this technology is very well integrated in Delphi; its elements are very well documented; and it is the only viable solution for accessing local files such as Paradox and dBase tables.

The disadvantages of this solution: Borland has stopped developing it (there will be no further updates); you'll have to install and configure it on the client computers; it is quite a "heavyweight" engine, with large installation files and memory requirements; and it is available only on Windows. If you have existing Delphi BDE applications accessing local files, there is no hurry to convert them and get rid of the BDE, unless you want to move your applications to Linux. If you are using an SQL server, migrating to another data-access technology will probably be easier.

BDE is still a good solution, if you balance advantages and disadvantages, but its long-term viability is certainly in doubt. I'll keep using the BDE for the simpler examples of this chapter, but only for the sake of simplicity. In any case, I'll try to stress the elements common to all the dataset components rather than focus on specific features of BDE or Paradox. The Delphi components related to the BDE are all hosted in the Data Access page of the Components palette. There are three dataset components, Table, Query, and StoredProc, plus the UpdateSQL used in connection with the Query component. The Database and Session components are used to set up the database connection. The BatchMove component is for copying data; the rarely used NestedTable component allows you to nest master-detail data in a sub-table; and the BDEClientDataSet component, introduced in Delphi 6, merges a ClientDataSet with a BDE-related data-access component.

3.1.1.2 ActiveX Data Objects (ADO)

ADO, which stands for ActiveX Data Objects, is Microsoft's high-level interface for database access. ADO is implemented on Microsoft's data-access OLE DB technology, which provides access to relational and non-relational databases as well as e-mail and file systems and custom business objects. ADO is an engine with features comparable to the BDE: database server independence supporting local and SQL servers alike, a really heavyweight engine, and a simplified configuration (because it is not centralized). Installation should in theory not be an issue, as the engine is part of recent versions of Windows.
However, the limited compatibility among versions of ADO will force you to upgrade your users' computers to the same version you've used for developing the program—and the sheer size of the MDAC (Microsoft Data Access Components) installation, which updates large portions of the operating system, makes this operation far from simple.

ADO offers some definite advantages if you plan on using Access or SQL Server, as Microsoft's drivers for their own databases are of better quality than the average OLE DB providers. For Access databases, specifically, using Delphi's ADO components is a good solution. But if you plan using other SQL servers, first check the availability of a good quality driver, as you might have some surprises. ADO is very powerful, but you have to learn living with it, as it really stands in the way between your program and the database, providing services but occasionally also issuing different commands than you are expecting. On the negative side, do not even think of using ADO if you plan future cross-platform development: this Microsoft-specific technology is not available on Linux or other operating systems. In short, use ADO if you plan working only on Windows, want to use Access or other Microsoft databases, or you find a good OLE DB provider for each of the

database servers you plan working with (at the moment, for example, this excludes InterBase and many other SQL servers).

ADO components (part of a package Borland called ADO Express in Delphi 5 and now calls dbGo in Delphi 6) are all grouped in the ADO page of the Components palette. The three core components are ADOConnection (for database connection), ADOCommand (for executing SQL commands), and ADODataSet (for executing requests that return a result set). There are also three compatibility components—ADOTable, ADOQuery, and ADOStoredProc—which you can use for porting BDE-based applications to ADO. Finally, there is the RDSConnection component, for accessing data in remote multitier applications.

3.1.1.3 The dbExpress Library

One of the relevant new features of Delphi 6 is the introduction of the dbExpress database library for the Windows platform. I say "library" because, unlike BDE and ADO, dbExpress uses a lightweight approach; and I underline "Windows" because the same library is available also for Linux in Borland Kylix.

Being light and portable are actually the two key characteristics of dbExpress and the reasons it has been introduced by Borland, along with the development of the Kylix project. There are certainly other database libraries you could use in the past and can still use with Delphi, but this new offering is worth a thought. Consider also it requires basically no configuration on the user machines.

Compared to other powerhouses, dbExpress is really limited in its capabilities. It can access only SQL servers (no local files); it has no caching capabilities and provides only unidirectional access to the data; it can natively work only with SQL queries and is unable of generating the corresponding SQL update statements.

At first sight, you might think that these limitations make the library pretty useless. On the contrary, these are *features* that make it interesting. Unidirectional datasets with no direct update are the norm if you need to produce reporting, including generating HTML pages showing the content of a database. If you want to build a user interface to edit the data, instead, consider that Delphi includes specific components (the ClientDataSet and Provider, in particular) that provide caching and query resolution. These components allow your dbExpress-based application to have much more control than you can have with a separate (and monolithic) database engine, which does extra things for you but often does it the way it wants, not the way you would like. Considering that Borland is pushing this library, and it is the only viable database-independent solution on Linux, I really urge you to consider it for new applications, and even to think about updating existing Delphi applications to this new architecture.

3.1.1.4 InterBase Express (IBX)

Delphi includes components for native access to Borland's own open-source (and free) Inter- Base server. Unlike BDE, ADO, and dbExpress, this is not a server-independent database engine, but a technology for accessing a specific database server. If you plan using only Inter- Base as your back-end RDBMS, using a specific set of components can give you more control of the server, provide the best performance, and allow you also to configure and maintain the server from within a custom client application.

3.1.1.5 The ClientDataSet Component

Finally, there is a component derived from TDataSet that has a peculiar behavior and can be combined with other data-access components. The ClientDataSet component, in fact, is a dataset accessing data kept in memory. The in-memory data can be totally temporary (lost as you exit the program), saved to a local file as a snapshot, and imported by another dataset using a Provider component. This last situation is certainly the most common: You can hook a ClientDataSet to any other local dataset, or use Borland's multitier support (discussed in Chapter 17, "Multitier Database Applications with DataSnap") to retrieve data from a dataset hosted by a different application, possibly running on a separate computer. The ClientDataSet component becomes particularly useful if the data-access components you are using provide limited or no caching. This is particularly true of the new dbExpress engine, but can equally help you when using the BDE or other native components. On the other hand, ADO already provides most of the services of the ClientDataSet component and using these two at the same time can be useful only in limited situations.

3.1.2 Classic BDE Components

Each of the database-access solutions discussed above has its own set of data-access, database connection, and extra utility components on a specific page of the Component palette. In Delphi 6, the classic BDE components have been moved to the new BDE page and include the Table, Query, and StoredProc components. The ADO, dbExpress, and InterBase Express components are each in specific pages, and all include specific dataset components and others that tend to mimic the BDE components, simplifying the porting of existing applications. The Data Access page of the Component palette in Delphi 6 includes only the Data Source component and others not specifically related with any single data access technology. Besides the data-access component of your choice, a Delphi visual application generally uses some data-aware controls (in the Data Controls page) and the DataSource component. Data-aware controls are visual components used to view and edit the data in a form and are extensions of standard components such as edit and list boxes,

radio buttons, images, and the grid. The DataSource component has the role of connector between the data-aware controls and a dataset component.

3.1.2.1 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 you use a Table component, you 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.

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.

3.1.2.2 Master/Detail Structures

Often you need to relate tables, which have a one-to-many relationship. This means that for a single record of the master table, there are many detailed records in a secondary table. A classic example is that of an invoice and the items of the invoice; another is a list of customers and the orders each customer has made. This is very common situation in database programming, and Delphi provides explicit support for it with the master/detail structure. We'll see this structure for BDE Table and Query components, but the same technique applies to almost all of the datasets available in Delphi.

Master/Detail with Tables

The simplest ways to create a master/detail structure in Delphi is to use the Database Form Wizard, selecting a master/detail form in the first page. To accomplish the same effect manually, place two table components in a form or data module, connect them with the same database, and connect each with a table. In the MastDet example, I've used the customer and orders tables of the DBDEMOS database, and I've used a data module. Now add a DataSource component for each table, and for the secondary table set a master source to the data source connected to the first table. Finally relate the secondary table to a field (called MasterField) of the main table, using the special property editor provided.

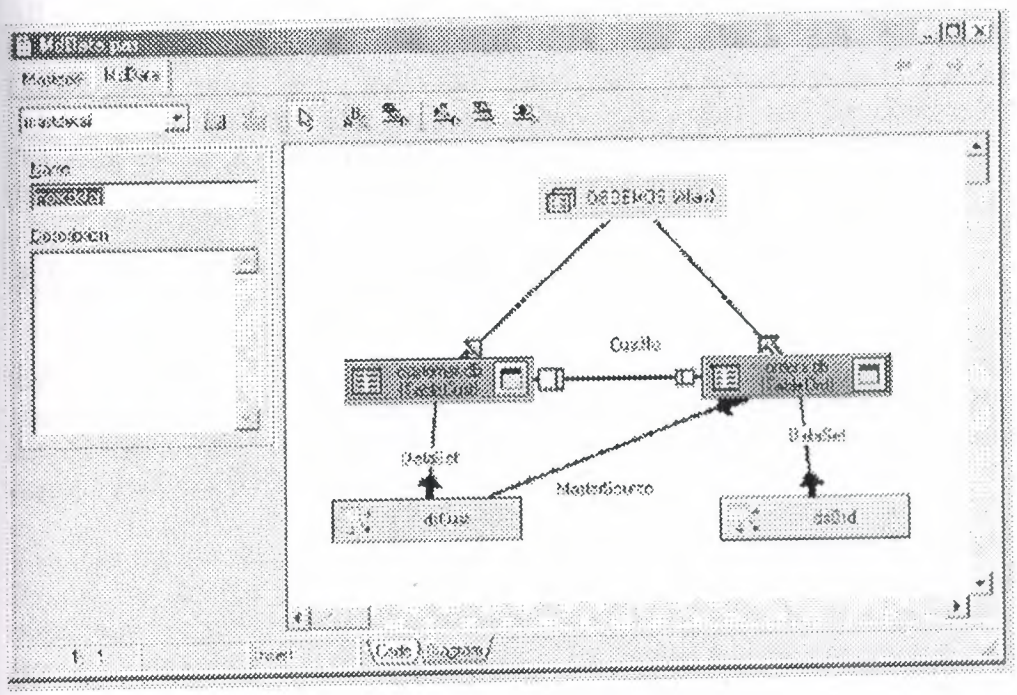
A Data Module for Data-Access Components

To build a Delphi database application, you can place data-access components and the dataaware controls in a form. This is handy for a simple program, but having the user interface and the data access and data model in a single, often large, unit is far from a good idea. For this reason, Delphi implements the idea of *data module*, a container of nonvisual components I already introduced in Chapter 1, "The Delphi 6 IDE."

At design time, a data module is similar to a form, but at run time it exists only in memory. The TDataModule class derives directly from TComponent, so it is completely unrelated to the Windows concept of a window (and is fully portable among different operating systems). Unlike a form, a data module has just a few properties and events. For this reason, it's useful to think of data modules as components and method containers.

Like a form or a frame, a data module has a designer. This means Delphi creates for a data module a specific Object Pascal unit for the definition of its class and a form definition file that lists its components and their properties. There are several reasons to use data modules. The simplest one is to share data-access components among multiple forms, as I'll demonstrate at the beginning of the next chapter. This technique works in conjunction with visual form linking, the ability to access components of another form or data module at design time (with the File .Use Unit command). The second reason is to separate the data from the user interface, improving the structure of an application. Data modules in Delphi even exist in versions specific for multitier applications (remote data modules) and server-side HTTP applications (Web data modules).

Finally, remember that you can use the Diagram page of the editor, introduced in Chapter 1, to see a graphical representation of the connections among the components of a data module, as you can see in this example for the MastDet application:



3.1.2.3 Other BDE Related Components

Along with Table, Query, StoredProc, and DataSource, other components are on the Data Access page of the Component palette, the BDE page. I'll cover these components in the next chapter, but here is a short summary:

- The Database component is used for transaction control, security, and connection control. It is generally used only to connect to remote databases in client/server applications or to avoid the overhead of connecting to the same database in several forms. The Database component is also used to set a local alias, one used only inside a program.

Once this local alias is set to a given path, the Table and Query components of the application can refer to the local database alias. This is much better than replicating the hard-coded path in each DataSet component of the program.

3.1.3 Using Data-Aware Controls

Once you've set up the proper data-access components, you can build a user interface to let a user view the data and eventually edit it. Delphi provides many components that resemble the usual Windows controls but are data-aware. For example, the DBEdit component is similar to the Edit component, and the DBCheckBox component corresponds to the CheckBox component. You can find all of these components in the Data Controls page of the Delphi Component palette.
All of these components are connected to a data source using the corresponding property, DataSource. Some of them relate to the entire dataset, such as the DBGrid and DBNavigator components, while the others refer to a specific field of the data source, as indicated by the DataField property. Once you select the DataSource property, the DataField property will have a list of values available in the drop-down combo box of the Object Inspector.

3.1.4 The DataSet Component

Instead of focusing right away on the use of a specific dataset, I prefer starting with a generic introduction of the features of the TDataSet class, which are shared by all inherited data-access classes. The DataSet component is a very complex one, so I won't list all of its capabilities but only discuss its core elements.

The idea behind this component is to provide access to a series of records that are read from some source of data, kept in internal buffers (for performance reasons), and eventually modified by a user, with the possibility of writing back changes to the persistent storage. This approach is generic enough to be applied to different types of data (even non-database data) but has a few rules. First, there can be only one active record at a time, so if

you need to access data in multiple records, you must move to each of them, read the data, then move again, and so on. You'll find an example of this and related techniques in the section about navigation.

Second, you can edit only the active record: you cannot modify a set of records at the same time, as you can in a relational database. Moreover, you can modify data in the active buffer only after you explicitly declare you want to do so, by giving the Edit command to the dataset. You can also use the Insert command to create a new blank record, and close both operations (insert or edit) by giving a Post command.

Other interesting elements of a dataset I will explore in the following sections are its status (and the status change events), navigation and record positions, and the role of the field objects. As a summary of the capabilities of the DataSet component, I've included the public methods of its class in Listing 13.1 (the code has been edited and commented for clarity). Not all of these methods are directly used everyday, but I decided to keep them all in the listing.

3.1.5 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. My personal 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 it will help me introduce some more database-related events (discussed in the sections "Database Events" and "Field Events").

The development of an application not based on data-aware controls can follow two different approaches. You can mimic the standard Delphi behavior in code, possibly departing from it in specific cases, or you can go for a much more customized approach. I'll demonstrate the first technique in the NonAware example and the latter in the SendToDb example.

3.1.6 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 you to place many data-aware controls in a small area of a form and automatically duplicate these controls for multiple records.

3.1.7 Handling Database Errors

Another important element of database programming is handling database errors in custom ways. Of course, you can let Delphi show an exception message each time a database error occurs, but you might want to try to correct the errors or simply show more details. There are basically three approaches you can use to handle database-related errors:

- You can wrap a try/except block around risky database operations, such as a call to the Open method of a 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.
- You can install a handler for the OnException event of the global Application object or use the ApplicationEvents component, as described in the next example.
- You can handle specific events of the datasets related to errors, as OnPostError, OnEditError, OnDeleteError, and 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 you see a list of errors, showing local BDE error codes and also the native error codes of the SQL server you are connected to. Besides the Message property, the EDBEngineError class has two more properties, ErrorCount and Errors.

3.2 Client/Server Programming

In the last chapter, we examined Delphi's support for database programming, using local files (particularly Paradox) in most of examples but not focusing on any specific database technology. This chapter moves on to the use of SQL server databases, focusing on client/server development with the BDE and the new dbExpress technology. A single chapter cannot cover this complex topic in detail, so I'll simply introduce it from the perspective of the Delphi developer and add some tips and hints. The next chapter extends our discussion of client/server programming, providing some real-world examples. I'll use InterBase in both chapters, because this is the RDBMS (relational database management system), or SQL server, that is included in the Enterprise edition of Delphi and because it is a free and open-source server. In a rapid application development tool such as Delphi, you can indeed take the same components and code developed for a local database application and use them in a client/server environment. However, this handy feature may prove to be dangerous to beginners, as a standard technique that works well for local access might become extremely inefficient in a client/server application.

3.2.1 An Overview of Client/Server Programming

The database applications in previous chapters used the BDE to access data stored in files either on the local machine or on a networked computer. In both cases we used a file server, whose only role was to store the file on a hard disk, because the database engine (the BDE) was running exclusively on the computer that also hosted the application. In this configuration, when we query one of the tables, its data is first copied into a local cache of the BDE and then processed.

```
select Max(Salary) from Employee
```

In case of a local table, this query would be processed by the local SQL engine of the BDE, and the entire dataset of the table would still have to be moved from the networked computer to the local one, with similarly poor performance. But if you use InterBase and let the server execute the SQL code, only the result set—a single number—will need to be transferred to the local computer.

If you want to store a large amount of data on a central computer and avoid moving the data to client computers for processing, the only solution is to let the central computer manipulate the data and send back to the client only a limited amount of information. This is the foundation of client/server programming.

In general, you'll use an existing program on the server (an RDBMS) and write a custom client application that connects to it. Sometimes, however, you might even want to write both a custom client and a custom server, as in three-tier applications. Delphi support for this type of program—what has been called the MIDAS architecture—is covered in Chapter 17, "Multitier Database Applications with DataSnap."

The *upsizing* of an application—that is, the transfer of data from local files to a SQL server database engine—is generally done for performance reasons and to allow for larger amounts of data. Going back to the previous example, in a client/server environment, the query used to select the maximum salary would be computed by the RDBMS, which would send back to the client computer only the final result, a single number! With a powerful server computer (such as a multiprocessor Sun SparcStation), the total time required to compute the result might be minimal.

However, there are also other reasons to choose a client/server architecture:

The amount of data A Paradox table cannot exceed 2 GB, but even around 300 MB you might start having serious speed problems, and errors in the indexes become more frequent.

The need for concurrent access to the data Paradox uses the Paradox.NET file to keep track of which user is accessing the various tables and records. The Paradox approach to handling multiple users is based on *pessimistic locking*. When a user starts an editing operation on a record, none of the other users can do the same (to avoid any update conflict), as we saw in the last chapter. In a system with tens of users, this might lead to serious problems, because a single user might block the work of many others. SQL server databases, by contrast, generally use *optimistic locking*, an approach that allows multiple

users to work on the same data and delays the concurrency control until the time the users send back some updates.

Protection and security An RDBMS usually has many more protection mechanisms than the simple password you can add to a Paradox table. When your application is based on files, a malicious or careless user might simply delete those vital files. When SQL servers are based on robust operating systems, instead, they provide multiple levels of protection, make backup easier, and often allow only the database administrator to modify the structure of the tables.
Programmability An RDBMS database can host business rules, in the form of stored procedures, triggers, table views, and other techniques we'll discuss in this and the next chapter. Choosing how to divide the application code between the client and the server is one of the main issues of client/server programming.
Transaction control Local files offer some support for transactions, but the transaction support provided by an RDBMS database is generally much greater. This is another important aspect of the overall robustness of the system.

3.2.2 From Local to Client/Server

Now we can start focusing on particular techniques useful for client/server programming. Keep in mind that the general goal is to distribute the workload properly between the client and the server and reduce the network bandwidth required to move information back and forth. The foundation of this approach is good database design, which involves both table structure and appropriate data validation and constraints, or business rules. Enforcing the validation of the data on the server is important, as the integrity of the database is one of the key aims of any program.
However, the client side should include data validation as well, to improve the user interface and make the input and the processing of the data more user-friendly. It makes little sense to let the user enter invalid data and then receive an error message from the server, when we can prevent the wrong input in the first place.

Unidirectional Cursors

In local databases, tables are sequential files whose order is either the physical order or is defined by an index. By contrast, SQL servers work on logical sets of data, not related to a physical order. A *relational* database server handles data according to the relational model, a mathematical model based on set theory.

What is important for the present discussion is that in a relational database, the records (sometimes called tuples) of a table are identified not by position but exclusively through a primary key, based on one or more fields. Once you've obtained a set of records, the server adds to each of them a reference to the following one, which makes it fast to move from a record to the following one but terribly slow to move back to the previous record. For this reason, it is common to say that an RDBMS uses a *unidirectional* cursor. Connecting such a table or query to a DBGrid control is practically impossible, as this would make it terribly

slow when browsing the grid backward.

The BDE helps a lot to handle unidirectional cursors, as it keeps in a local cache the records already loaded in the table. Thus, when we move to following records, they are requested from the SQL server; but when we go back, the BDE jumps in and provides the data. In other words, the BDE makes these cursors fully bidirectional, although this might use quite a lot of memory. When using dbExpress, which doesn't provide a similar caching system, a program needs to keep in memory the records it has already accessed. This can be easily accomplished by means of the ClientDataSet component.

3.2.3 Client/Server with the BDE

Now let's consider how Delphi fits into the client/server picture. How does it help us build client/server applications? As I've mentioned, you can still use all the components and techniques discussed in the Chapter 13, "Delphi's Database Architecture," although in some cases alternate approaches will help you leverage the power of the RDBMS your application is dealing with.

As a starting point, let's cover a few considerations on Delphi client/server development using the BDE and its components. After this I'll move to dbExpress, which in Delphi 6 is the recommended general solution for client/server development.

3.2.3.1 SQL Links

The BDE doesn't know how to handle the RDBMS; it uses some further drivers, called SQL Links, to perform this operation. As an alternative, the BDE can also interact with ODBC drivers. Borland provides native BDE drivers for InterBase, Oracle, Informix, Microsoft SQL Server, Sybase, and DB2.

If the BDE is still required on the local machines, it can actually be very efficient. For example, when you use the pass-through mode for queries, the BDE doesn't try to interpret the SQL code but passes it directly to the RDBMS server. This allows you to use a server's specific SQL commands and also to speed up the execution. The pass-through mode is activated using the BDE Administrator utility.

3.2.3.2 BDE Table and Query Components in Client/Server

In Delphi there are two BDE components you use to access an existing database table: Table and Query. When building client/server applications, programmers tend to use the Query component exclusively, but that is certainly not mandatory, and there are cases in which using the simpler Table component has no drawback.

3.2.3.3 Live Queries and Cached Updates

When working with local data, it is very common to use grids and other visual controls, edit the data, and send it back to the database. We've already seen that using a DBGrid might cause problems when working with an RDBMS, as moving on the grid might send numerous data requests to the server, creating a huge amount of network traffic.

When you use the Query component to connect to some data, you cannot edit the data unless its RequestLive property is set to True. If you are working with local tables, the query is always elaborated by the BDE with the Local SQL engine. The BDE will allow for a live query only if it is quite simple: All joins should be outer joins; there cannot be a distinct key; there can be no aggregation, no group by or having clause, no subqueries, and no order by unless supported by an index; and there are other rules you can find in Delphi's Help. If you are working with a SQL server, setting a live query will put the BDE in control of the query, instead of the server. When connected to a SQL server, a live query behaves like a Table component. (So it makes sense to use the table anyway, in these cases.)

3.2.3.4 Using Transactions

Whether you are working with a SQL server, you should use *transactions* to make your applications more robust. The idea of a transaction can be described as a series of operations to be considered as a single, "atomic" whole that cannot be split.
An example may help to clarify the concept. Suppose you have to raise the salary of each employee of a company by a fixed rate, as we did in the Total example of the preceding chapter.
Now if during the operation an error occurs, you might want to undo the previous changes.If you consider the operation "raise the salary of each employee" as a single transaction, it should either be completely done or completely ignored. Or consider the analogy with financial transactions—if only part of the operation is performed, because of an error, you might end up with a missed credit or with some extra money!

Working with database operations as transactions serves a useful purpose. You can start a transaction and do several operations that should all be considered parts of a single larger operation; then, at the end, you can either commit the changes or *roll back* the transaction, discarding all the operations done up to now. Typically, you might want to roll back a transaction if an error occurred during its operations.

Handling transactions in Delphi is quite simple. By default, each edit/post operation is considered a single *implicit* transaction, but you can alter this behavior by handling them explicitly.

3.2.3.5 Using SQL Monitor

Just as you need a debugger to test a Delphi application, you need some tools to test how a client/server application behaves and to speed it up if possible. In particular, it is very important to look at the information moving from the client to the server (the explicit SQL requests our program does and those added by the BDE) and from the server to the client (the actual data). This is what the SQL Monitor tool included in Delphi Enterprise is for.

3.2.4 The dbExpress Library

As I mentioned in Chapter 13, one of the most notable new features of Delphi 6 is the adoption of the dbExpress database access library, a new SQL server access layer introduced by Borland in its Kylix product for Linux, and now with Delphi 6 also for Windows. As I've already provided a general overview of dbExpress in the preceding chapter, let's focus right away on the technical details.

3.2.4.1 Working with Unidirectional Cursors

The motto of dbExpress could be "fetch but don't cache." The key difference between this library and BDE or ADO is that dbExpress can only execute SQL queries and fetch the results in a *unidirectional cursor*. In "unidirectional" database access, you can move from one record to the next, but you cannot get back to a previous record of the dataset. This is because the library doesn't store the data is has retrieved in a local cache, but only passes it from the database server to the calling application. Using a unidirectional cursor might sound like a limitation, and it really is! Besides having problems with the navigation, you cannot connect a database grid to a dataset like this. So what is a unidirectional dataset good for?

- You can use a unidirectional dataset for reporting purposes. In a printed report, but also an HTML page or an XML transformation, you move from record to record, produce the output, and that's it. No need to get back to past records and, in general, no interaction of the user with the data. Unidirectional datasets are probably the best option for Web and multitier architectures.
- You can use a unidirectional dataset to feed a local cache, such as the one provided by a ClientDataSet component. At this point, you can connect visual components to the inmemory dataset and operate on it with all the standard techniques, including the use of visual grids. You can freely navigate and edit the data in the in-memory cache, but also control it far better than with the BDE or ADO.

The important thing to notice is that, in these circumstances, avoiding the caching of the database engine actually saves time and memory. The library doesn't have to use extra memory for the cache and doesn't need to waste time storing data, duplicating information. Over the last couple of years, many programmers moved from BDE-based cached updates to the ClientDataSet component, which provides more flexibility in managing the content of the data and update information they keep in memory. However, using a ClientDataSet on top of the BDE (or ADO) exposes you to the risk of having two separate caches, actually wasting a lot of memory.

Another advantage of using the ClientDataSet component is that its cache supports editing operations, and the updates stored in this cache can be applied to the original database server by the DatasetProvider component. This component can generate the proper SQL update statements, and can do so in a more flexible way than the BDE (although ADO is quite powerful in this respect). In general, the provider can also use a dataset for the updates, but this isn't directly possible with the dbExpress dataset components.

3.2.4.2 Platforms and Databases

A key element of the dbExpress library is its availability for both Windows and Linux, in contrast to all the other database engines available for Delphi (BDE and ADO), which are available only for Windows. Notice, though, that some of the database-specific components, such as InterBase Express, are also available on multiple platforms.

When you use dbExpress, you are provided with a common framework, which is independent from the actual SQL database server you are planning to use. dbExpress comes with drivers for MySQL, InterBase, Oracle, and IBM DB2. These drivers are available as separate DLLs you have to deploy along with your program or as compiled units you can link into the executable file.

3.2.5 ClientDataSet and MyBase

The general idea of a client/server application implies that the computation workload is shared between two separate programs, the RDBMS and a client application.
Although it is very hard to strike a precise line between the two sides, it is certainly useful to do operations on the client. Most database engines (BDE, as we've seen in this chapter, and ADO, as we'll see in Chapter 16, "ActiveX Data Objects") can manipulate client-side data stored in a cache. Using the ClientDataSet component, you can do the same regardless of the database engine
you are using, which makes your program more flexible, particularly if you want to use dbExpress, which doesn't provide a similar feature natively.

A practical example will underline what I mean: Suppose you've written a SQL query to retrieve a rather large dataset, and a user wants to see the same data in a different order. You can certainly run a new query, with the proper order by clause, but this implies sending the same (possibly large) dataset once more from the server to the client. Since the client already has the data in memory, it would be more practical and generally faster to re-sort the data in memory and present the same data to the user with a different ordering.

Indexing is not all the ClientDataSet has to offer. When you have an index, you can define groups based on it, possibly with multiple levels of grouping. There is even specific support for determining the position of a record within a group (first, last, or middle position). Over groups or entire tables, you can define aggregates; that is, you can compute the sum or average value of a column for the entire table or the current group on-the-fly.

The data doesn't need to be posted to a physical server, because these aggregate operations take place in memory. You can even define new aggregate fields, to which you can directly connect data-aware controls. I'll explore these capabilities in the next section.

Another very interesting area of the ClientDataSet component is its ability to handle the updates log, undoing changes, looking at their list before committing them, and so on. I'll explore this next.

The ClientDataSet component supports many features, only some of which are related to the three-tier architecture (covered in Chapter 17). This component represents a database completely mapped in memory and can also be made persistent to a local file. Borland marketing has introduced the name MyBase to describe this feature of the ClientDataSet component, which was formerly called the briefcase model.

The important thing to keep in mind is that all of these features are available to any client/server and even local applications. The ClientDataSet component, in fact, can get its data from a remote connection, from a local dataset (as you must do with dbExpress), or from a local MyBase file. This is another huge area to explore, so I'll simply show you a couple of examples highlighting key features.

3.2.5.1 The Packets and the Cache

The ClientDataSet component reads data in packets made of the number of records indicated by the PacketRecords property.

The default value of this property is –1, which means that the provider will pull all the records at once (this is reasonable only for a small dataset). Alternatively, you can set this value to zero to ask the server for only the field descriptors and no actual data or use any positive value to specify an actual number.

If you retrieve only a partial dataset, as you browse past the end of local cache, if FetchOnDemand property is set to True (the default value), the ClientDataSet component will get more records from its source. This same property also controls whether BLOB fields and nested datasets of the current records are fetched automatically (these values might not be already part of the data packet, depending on the value of the Options of the dataset provider).

If you turn off this property, you'll need to manually fetch more records, by calling the GetNextPacket method, until the method returns zero. (You'll call FetchBlobs and FetchDetails for these other elements.)

3.2.5.2 Manipulating Updates

One of the core ideas behind the ClientDataSet component is that it is used as a local cache to collect some input from a user and then send a batch of update requests to the database.

The component has both a list of the changes to apply to the database server, stored in the same format used by the ClientDataSet (accessible though the Delta property), and a complete updates log that you can manipulate with a few methods (including an Undo capability).

3.2.5.3 MyBase (or the Briefcase Model)

The last capability of the ClientDataSet component I want to discuss in this chapter is its support for mapping memory data to local files, building stand-alone applications. The same technique can be applied in multitier applications to use the client program even when you're not physically connected to the application server. In this case, you can save all the data you expect to need in a local file for travel with a laptop (perhaps visiting client sites). You'll use the client program to access the local version of the data, edit the data normally, and when you reconnect, apply all the updates you've performed while disconnected.

3.3 InterBase and IBX

Client/server programming requires two sides: a client application that you probably want to build with Delphi, and a relational database management system (RDBMS), usually a "SQL server." In this chapter, I focus on one specific SQL server, InterBase. There are many reasons for this choice. InterBase is the SQL server developed by Borland; it is an open source project and can be obtained for free; and it has traditionally been bound with Delphi, which has specific dataset components for it.

For all of these reasons, InterBase should be a good choice for your Delphi client/server development, although there are many other equally powerful alternatives. I'll discuss Inter- Base from the Delphi perspective, without delving in to its internal architecture. A lot of the information presented also applies to other SQL servers, so even if you've decided not to use InterBase, you might still find it valuable.

3.3.1 Getting Started with InterBase

After installing InterBase, you'll be able to activate the server from the Windows Start menu, but if you plan on using it frequently, you should install it as a Windows service (of course, only if you have Windows NT/2000, as Windows 9*x*/Me doesn't have support for services). When the server is active, you'll see a corresponding icon in the Tray Icon area of the Windows Taskbar (unless you start it as a service). The menu connected with this icon allows you to see status information (see Figure 15.1) and do some very limited configuration.

3.3.1.1 Inside InterBase

Even though it has a limited market share, InterBase is a very powerful RDBMS. In this section I'll introduce the key technical features of InterBase, without getting into too much detail. This is a book on Delphi programming, in fact. Unfortunately, there is currently very little published about InterBase, although there are some ongoing efforts for an InterBase book and there is a wealth of information in the documentation accompanying the product and on a few Web sites devoted to the product.

InterBase was built from the beginning with a very modern and robust architecture. Its original author, Jim Starkey, invented an architecture for handling concurrency and transactions without imposing physical locks on portions of the tables, something other well-known database servers can hardly do even today. InterBase architecture is called Multi-Generational Architecture (MGA), and it handles concurrent access to the same data by multiple users, who can modify records without affecting what other concurrent users see in the database. This approach naturally maps to the *Repeatable Read* transaction isolation mode, in which a user within a transaction keeps seeing the same data, regardless of changes done and committed by other users. Technically, the server handles this by maintaining a different version of each accessed record for each open transaction. Even if this approach (also called *versioning*) can lead to larger memory consumption, it avoids almost any physical lock on the tables and makes the system much more robust in case of a crash. Also, MGA pushes toward a very clear programming model—Repeatable Read—which other well-known SQL servers don't even support without losing most of their performance. If Multi-Generational Architecture is at the heart of InterBase, the server has many other technical advantages:

- A limited footprint, which makes InterBase the ideal candidate for running directly on client computers, including portables. The disk space required by InterBase for a minimal installation is well below 10 MB, and its memory requirements are also incredibly limited.
- Good performance on large amounts of data.
- Availability on many different platforms (including 32-bit Windows, Solaris, and Linux), with totally compatible versions, which makes the server scalable from very small to huge systems without notable differences.
- A very good track record, as InterBase has been in use for 15 years with very few problems.
- A language very close to the SQL standard.
- Advanced programming capabilities, with positional triggers, selectable stored procedures, updateable views, exceptions, events, generators, and more.
- Simple installation and management, with limited administration headaches.

A Short History of InterBase

Jim Starkey wrote InterBase for his own Groton Database Systems company (hence the .gds extension still in use for InterBase files). The company was later bought by Ashton-

Tate, which was then acquired by Borland. Borland handled InterBase directly for a while, then created an InterBase subsidiary, which was later re-absorbed into the parent company.

Starting with Delphi 1, an evaluation copy of InterBase has been distributed along with the development tool, spreading the database server among developers. Although it doesn't have a large piece of the RDBMS market, which is dominated by a handful of players, InterBase has been chosen by a few very relevant organizations, from Ericsson to the U.S. Department of Defense, from stock exchanges to home banking systems.

More recent events include the announcement of InterBase 6 as an open source database (December 1999), the effective release of source code to the community (July 2000), and the release of the officially certified version of InterBase 6 by Borland (March 2001).

In between these events, there were announcements of the spin-off of a separate company to run the consulting and support business on top of the open source database. Contacts with a group of former InterBase developers and managers (who had left Borland) didn't lead to an agreement, but the group decided to go ahead even without Borland's help and formed IBPhoenix (www.ibphoenix.com) with the plan of supporting InterBase users.

At the same time, independent groups of InterBase experts formed the InterBase Developer Initiative (IBDI; www.interbase2000.org) and started the Firebird open source project to further extend InterBase. For this reason, SourceForge currently hosts two different versions of the project, InterBase itself run by Borland and the Firebird project run by this independent group. You see that the picture is rather complex, but this certainly isn't a problem for InterBase, as there are currently many organizations pushing it, along with Borland.

3.3.1.2 IBConsole

In past versions of InterBase, there were two main tools you could use to interact directly with the program: the Server Manager application, which could be used to administer both a local and a remote server; and Windows Interactive SQL (WISQL). Version 6 includes a much more powerful front-end application, called IBConsole. This is a full-fledged Windows program (built with Delphi) that allows you to administer, configure, test, and query an InterBase server, whether local or remote.

IBConsole is a simple and complete system for managing InterBase servers and their databases. You can use it to look into the details of the database structure, modify it, query the data (which can be useful to develop the queries you want to embed in your program), back up and restore the database, and perform all the other administrative tasks. As you can see in Figure 15.2, IBConsole allows you to manage multiple servers and databases, all listed in a single, handy configuration tree. You can ask for general information about the database and list its entities (tables, domains, stored procedures, triggers, and everything else), accessing the details of each. You can also create new databases and configure them, back up the files, update the definitions, check what's going on and who is currently connected, and so on.

The IBConsole application allows you to open multiple windows to look at detailed information, such as the tables window depicted in Figure 15.3. In this window, you can see lists of the key properties of each table (columns, triggers, constraints, and indexes), see the raw metadata (the SQL definition of the table), access permissions, have a look at the actual data, modify it, and study the dependencies of the table. Similar windows are available for each of the other entities you can define in a database.

Finally, IBConsole embeds an improved version of the original Windows Interactive SQL application (see Figure 15.4). You can directly type a SQL statement in the upper portion of the window (without any actual help from the tool, unfortunately) and then execute the SQL query. As a result, you'll see the data, but also the access *plan* used by the database (which an expert can use to determine the efficiency of the query) and some statistics on the actual operation performed by the server.

This is really a minimal description of IBConsole, which is a rather powerful tool and the only one included by Borland with the server besides command-line tools. IBConsole is probably not the most complete tool in its category, though. Quite a few third-party InterBase management applications are more powerful, although they are not all very stable or user-friendly. Some InterBase tools are shareware programs, while others are totally free. Two examples, out of many, are InterBase Workbench (www.interbaseworkbench.com) and IB_WISQL (done with and part of InterBase Objects, www.ibobjects.com).

3.3.2 Server-Side Programming

At the beginning of the previous chapter, I underlined the fact that one of the objectives of client/server programming—and one of its problems—is the division of the workload between the computers involved. When you activate SQL statements from the client, the burden falls on the server to do most of the work. However, you should try to use select statements that return a large result set, to avoid jamming the network.

Besides accepting DDL and DML requests, most RDBMS servers allow you to create routines directly on the server using the standard SQL commands plus their own server-specific extensions (which are generally not portable). These routines typically come in two forms, stored procedures and triggers.

3.3.2.1 Stored Procedures

Stored procedures are like the global functions of a Delphi unit and must be explicitly called by the client side. Stored procedures are generally used to define routines for data maintenance, to group sequences of operations you need in different circumstances, or to hold complex select statements.

Like Pascal procedures, stored procedures can have one or more typed parameters. Unlike Pascal procedures, they can have more than one return value. As an alternative to returning a value, a stored procedure can also return a result set, the result of an internal select statement or a custom fabricated one.

3.3.2.2 Triggers (and Generators)

Triggers behave more or less like Delphi events and are automatically activated when a given *event* occurs. Triggers can have specific code or call stored procedures; in both cases, the execution is done completely on the server. Triggers are used to keep data consistent, checking new data in more complex ways than a check constraint allows, and to automate the side effects of some input operations (such as creating a log of previous salary changes when the current salary is modified).

Triggers can be fired by the three basic data update operations: insert, update, and delete. When you create a trigger, you indicate whether it should fire before or after one of these three actions.

3.4 ActiveX Data Objects

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

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

ADO has great similarities with the BDE. They are, after all, designed to solve very similar problems. Both support navigation of datasets, manipulation of datasets, transaction processing, and cached updates (called *batch updates* in ADO), so the concepts and issues involved in using ADO are similar to those of the BDE. However, there are also differences. ADO is a more recent technology. This gives it an advantage over the BDE

because it is better suited to today's needs and doesn't need to carry so much deadwood around with it. Perhaps more importantly, ADO has a wider interpretation of "data." The BDE is used for accessing "rectangular" data—that is, data in rows and columns. This is ideal for accessing data from databases. ADO can be used for accessing this data but can also be used for accessing nonrectangular data, including directory structures, documents, Web sites, and e-mail.

In this chapter we will look at ADO and *dbGo*. (This set of Delphi components was called ADOExpress but has been renamed dbGo in Delphi 6, because Microsoft has objected to the use of the term *ADO* in product names.) It is possible to use ADO in Delphi without using dbGo. By importing the ADO type library, you can gain direct access to the ADO interfaces; this is, indeed, how Delphi programmers used ADO before the release of Delphi 5. However, this path bypasses Delphi's database infrastructure and ensures that you are unable to make use of other Delphi technologies such as DataSnap. Alternatively, you can turn to Delphi's active third-party market for other ADO component suites such as Adonis, AdoSolutio, Diamond ADO, and Kamiak.

This chapter uses dbGo for all of its examples, not only because it is readily available and supported but also because it is a very viable solution. Regardless of your final choice, you will find the information here useful.

3.4.1 Microsoft Data Access Components (MDAC)

ADO is part of a bigger picture called Microsoft Data Access Components (MDAC). MDAC is an umbrella for Microsoft's database technologies and includes ADO, OLE DB, ODBC, and RDS (Remote Data Services). Often you will hear people use the terms MDAC and ADO interchangeably (but incorrectly) because their version numbers and releases are now aligned. As ADO is only distributed as part of MDAC, we talk in terms of MDAC releases. The major releases of MDAC have been versions 1.5, 2.0, 2.1, 2.5, and 2.6. Microsoft releases MDAC independently and makes it available for free download and virtually free distribution (there are distribution requirements, but almost all Delphi developers will not have trouble meeting these requirements). MDAC is also distributed with most Microsoft products that have some kind of database content. This includes Windows 98, Windows 2000, Windows Millennium Edition, Microsoft Office, Internet Explorer, and SQL Server. In addition, Delphi 6 Enterprise ships with MDAC 2.5, and Delphi 5 Enterprise ships with MDAC 2.1.

There are two consequences of this level of availability. First, it is highly likely that your users will already have MDAC installed on their machines. Second, whatever version your users have, or you upgrade them to, it is also virtually certain that someone, either you, your users, or other application software, will upgrade their existing MDAC to whatever

the current release of MDAC is. There is no way you will be able to prevent this, as MDAC is installed with such commonly used software as Internet Explorer. Add to this the fact that Microsoft supports only the current release of MDAC and the release before it, and you are forced to arrive at a
conclusion:
Your applications must be designed to work with the current release of MDAC or the release before it. If you chart the releases of MDAC, you can expect to see a new version of MDAC every 10 months on average (Delphi itself has a new release every 14 months on average). As an ADO developer, you should regularly check the MDAC pages on Microsoft's site at www.microsoft.com/data. From there you can download the latest version of MDAC for free. At the time of writing, this is MDAC 2.6, which you can download from www.microsoft.com/ data/download_260rtm.htm (5.2 MB), but you should check for a more recent version first.

3.4.2 OLE DB Providers

OLE DB providers enable access to a source of data. They are ADO's equivalent to the BDE's drivers. However, although the BDE's Driver SDK has been available for many years, there are no third-party BDE drivers. This is not the case for OLE DB providers. MDAC includes many providers that I'll discuss, but many more are available from Microsoft and, more prolifically, the third-party market. It is no longer possible to reliably list all available OLE DB providers, because the list is so large and ever-changing, but here are some of the main vendors of these drivers:

Company	Web Site	OLE DB Provider
B2 Systems	www.b2systems.com	SQL Server, Oracle, Sybase, Informix, RDB, DB2, flat files
ISG	www.isgsoft.com	ISG Navigator (ISAM, DB2, IMS, Informix, Jasmine, Open Ingres, Oracle, SQL Server, Sybase, Adabas, RDB, RMS, VSAM)
Merant	www.merant.com	DB2, Informix, Lotus Notes, SQL Server, Ingres, Oracle, Sybase

You should add to this list almost all database vendors, as the majority now supply their own OLE DB providers. For example, Oracle supplies the ORAOLEDB provider. Notable omissions include InterBase; at the time of writing, Borland doesn't plan to write an OLE DB provider for InterBase. Your only solutions are to access InterBase either using the ODBC Driver, or through Jason Wharton's InterBase provider (www.ibobjects.com, although this is still in development) or Dmitry Kovalenko's IBProvider (www.lcpi.lipetsk.ru/prog/eng/ index.html).

MDAC OLE DB Providers

When you install MDAC, you automatically install the OLE DB providers shown in Table16.1:

TABLE 16.1: OLE DB Providers Included with MDAC

Driver	Provider	Description
MSDASQL	ODBC Drivers	ODBC drivers (default)
Microsoft.Jet.OLEDB.3.5	Jet 3.5	MS Access 97 databases only
Microsoft.Jet.OLEDB.4.0	Jet 4.0	MS Access databases et al.
SQLOLEDB	SQL Server	MS SQL Server databases
MSDAORA	Oracle	Oracle databases
MSOLAP	OLAP Services	Online Analytical Processing
SampProv	Sample provider	Example of an OLE DB provider for CSV files
MSDAOSP	Simple provider	For creating your own providers for simple text data

If you do not specify which OLE DB provider you are using, OLE DB defaults to the ODBC OLE DB Provider, which is used for backward compatibility with ODBC. As you learn more about ADO, you will discover the limitations of this provider. It's probable that you will eventually tire of these limitations, and I recommend that you look from the beginning for an OLE DB provider that is specific to your database rather than struggle with the ODBC OLE DB Provider.

The Jet OLE DB Providers support MS Access and other "desktop" databases. We will return to these providers later.

The SQL Server Provider supports SQL Server 7, SQL Server 2000, and Microsoft Database Engine (MSDE). MSDE is worth taking a moment's thought over. MSDE is SQL Server with most of the tools removed and some code added to deliberately degrade performance when there are more than 5 active connections. MSDE is important for two reasons. First, it is free. You can download it from Microsoft's Web site and, with very few restrictions, distribute it with your application. Second, MSDE is SQL Server. Of course you don't get the SQL Server tools, and it does deliberately degrade performance, but it is SQL Server. This means that it is perfect for use with low numbers of users. When the number of users increases and performance starts to suffer, your upgrade path to SQL Server is just a question of paying for SQL Server. Compatibility is virtually assured. You use the same OLE DB provider, and you use it in exactly the same way with exactly the same names and parameters. This is because MSDE is SQL Server. Because of these reasons and because Microsoft is moving their emphasis away from Access, MSDE is worth considering for future developments.

The OLE DB Provider For OLAP can be used directly but is more often used by ADO Multi-Dimensional (ADOMD). ADOMD is an additional ADO technology designed to provide Online Analytical Processing (OLAP). If you have used Delphi's Decision Cube, or Excel's Pivot Tables, or Access's Cross Tabs, then you have used some form of OLAP. In addition to these MDAC OLE DB providers, Microsoft supplies other OLE DB

providers with other products or with downloadable SDKs. The Active Directory Services OLE DB Provider is included with the ADSI SDK; the AS/400 And VSAM OLE DB Provider is included with SNA Server; and the Exchange OLE DB Provider is included with Microsoft Exchange 2000.

The OLE DB Provider For Indexing Service provides access to (and is part of) Microsoft Indexing Service, a Windows NT and 2000 mechanism that speeds up file searches by building catalogs of file information. Indexing Service is integrated into IIS and, consequently, is often used for indexing Web sites. Microsoft Indexing Service is also available for Windows NT 4 as part of the NT 4 Option Pack.

The OLE DB Provider For Internet Publishing is included with Internet Explorer 5, Windows 2000, and Office 2000 and allows developers to manipulate directories and files using HTTP. This is useful for maintaining Web sites that support either FrontPage Web Extender (WEC) or Web Distributed Authoring and Versioning (WebDAV). Still more OLE DB providers come in the form of *service providers*. As their name implies, OLE DB service providers provide a service to other OLE DB providers. Often these service providers will go unnoticed because they are invoked automatically as needed without programmer intervention. The Cursor Service, for example, is invoked when you create a clientside cursor, and the Persisted Recordset provider is invoked to save data locally.

3.4.3 dbGo

The set of components that make up dbGo (Table 16.2) should be easily recognizable by programmers familiar with the BDE, dbExpress, or IBExpress.

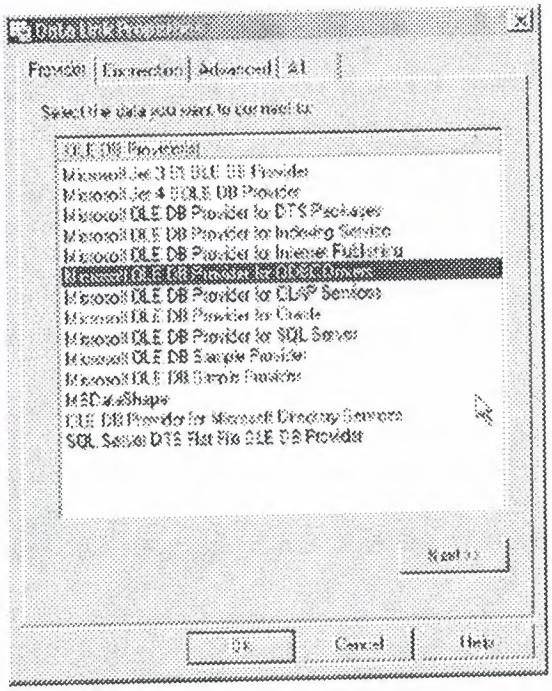
TABLE 16.2: dbGo Components

dbGo Component	Description	BDE Equivalent Component
TADOConnection	Connection to a database	TDatabase
TADOCommand	Executes an action SQL command	No equivalent
TADODataSet	All-purpose TDataSet	No equivalent
TADOTable	Encapsulation of a table	TTable
TADOQuery	Encapsulation of SQL SELECT	TQuery
TADOStoredProc	Encapsulation of a stored procedure	TStoredProc
TRDSConnection	Remote Data Services connection	No equivalent

The four dataset components (TADODataSet, TADOTable, TADOQuery, TADOStoredProc) are implemented almost entirely by their immediate ancestor TCustomADODataSet. This component provides the majority of dataset functionality, and its descendants are mostly thin wrappers that expose different features of the same component. As such, the components have a lot in common. In general, however, TADOTable, TADOQuery, and TADOStoredProc are viewed as "compatibility" components and are used to aid the transition of knowledge and code from their BDE counterparts. Be warned, though: These

compatibility components are similar to their counterparts but not exactly the same. You will find
differences in any application except the most trivial. TADODataSet is the component of choice partly because of its versatility but also because it is closer in appearance to the ADO Recordset interface upon which it is based. Throughout this chapter, we will use all of the TDataSet components to give you the experience of using each. Enough theory. Let's see some action. Drop a TADOTable onto a form. Look in the Object Inspector and you will not see any DatabaseName or AliasName properties. ADO doesn't use aliases, so there are no alias-related properties. Instead ADO runs on *connection strings*.

Connection strings are the lifeblood of ADO, and you should take time out to master this subject. You can type in a connection string by hand if you know what you are doing, but only programmers who think that VI is a great editor of our time enjoy doing this. For the rest of us, there is the connection string editor. In the Object Inspector, click the ellipses in the ConnectionString property. This invokes Delphi's connection string editor (Figure 16.1).



TADOConnection

When a TADOTable component is used in this way, it creates its own connection component behind the scenes in the same way that the BDE components create their temporary TDatabase component. You do not have to accept the default connection it creates, and you should not accept it. Instead, you should create your own connection in the form of a TADOConnection component.

The TADOConnection component is used for many of the same purposes as the BDE's TDatabase component. It allows you to customize the login procedure, control transactions, execute action commands directly, and reduce the number of connections in an application. Using a TADOConnection is easy. Place one on a form and set its ConnectionString property in the same way as for the TADOTable. Alternatively, you can double-click a TADOConnection to invoke the connection string editor directly. With the ConnectionString set to Northwind.mdb, you can disable the login dialog box by setting LoginPrompt to False. To make use of the new connection, set ADOTable1's Connection property to ADOConnection1. You will see ADOTable1's ConnectionString property reset because Connection and ConnectionString are mutually exclusive. One of the benefits of using a TADOConnection is that the connection string is centralized instead of scattered throughout many components. Another, more important, benefit is that all of the components that share the TADOConnection share a single connection. Without your own TADOConnection, each ADO dataset uses its own connection.

3.5 Multitier Database Applications with DataSnap

Large companies often have broader needs than applications using local database and SQL servers can meet. In the past few years, Borland Software Corporation has been addressing the needs of large corporations, and it even temporarily changed its own name to Inprise to underline this new enterprise focus. The name was changed back to Borland, but the focus on enterprise development remains.

Delphi is targeting many different technologies: three-tier architectures based on Windows NT and DCOM, CORBA architectures based on NT and Unix servers, TCP/IP and socket applications, and—most of all—SOAP- and XML-based Web services. This chapter focuses on database-oriented multitier architectures, while XML-oriented solutions will be discussed in Chapter 23, "XML and SOAP."

Even though I haven't yet discussed COM and sockets (covered in Chapters 19 to 21), in this chapter we'll build multitier architectures based on those technologies. As we'll use highlevel Delphi support, not knowing the details of some of the foundations should not create any problem. I'll concentrate more on the programming aspects of these architectures than on installation and configuration (the latter aspects are subject to change across different operating systems and are too complex to cover thoroughly).

Before proceeding, I should emphasize two important elements. First, the tools to support this kind of development are available only in the Enterprise version of Delphi; and second, you'll have to pay a license fee to Borland in order to deploy the necessary server-side software for DataSnap. This second requirement makes this architecture cost-effective mainly for large systems (that is, servers connected to dozens or even hundreds of clients). The license fee is only required for deployment of the server application and is a flat fee for

each server you deploy to (regardless of the number of clients that will connect). The license fee is not required for development or evaluation purposes.

3.5.1 One, Two, Three Levels

Initially, database PC applications were client-only solutions: the program and the database files were on the same computer. From there, adventuresome programmers moved the database files onto a network file server. The client computers still hosted the application software and the entire database engine, but the database files were now accessible to several users at the same time. You can still use this type of configuration with a Delphi application and Paradox files (or, of course, Paradox itself), but the approach was much more widespread just few years ago.

The next big transition was to client/server development, embraced by Delphi since its first version. In the client/server world, the client computer requests the data from a server computer, which hosts both the database files and a database engine to access them. This architecture downplays the role of the client, but it also reduces its requirements for processing power on the client machine. Depending on how the programmers implement client/server, the server can do most (if not all) of the data processing. In this way, a powerful server can provide data services to several less powerful clients.

Naturally, there are many other reasons for using centralized database servers, such as the concern for data security and integrity, simpler backup strategies, central management of data constraints, and so on. The database server is often called a SQL server, because SQL is the language most commonly used for making queries into the data, but it may also be called a DBMS (database management system), reflecting the fact that the server provides tools for managing the data, such as support for backup and replication.

Of course, some applications you build may not need the benefits of a full DBMS, so a simple client-only solution might be sufficient. On the other hand, you might need some of the robustness of a DBMS system, but on a single, isolated computer. In this case, you can use a local version of a SQL server, such as InterBase. Traditional client/server development is done with a two-tier architecture. However, if the DBMS is primarily performing data storage instead of data- and number-crunching, the client might contain both user interface code (formatting the output and input with customized reports, data-entry forms, query screens, and so on) and code related to managing the data (also known as *business rules*). In this case, it's generally a good idea to try to separate these two sections of the program and build a logical three-tier architecture. The term *logical* here means that there are still just two computers (that is, two physical tiers), but we've now partitioned the application into three distinct elements.

Delphi 2 introduced support for a logical three-tier architecture with data modules. As you'll recall, a *data module* is a nonvisual container for the data access components of an application, but it often includes several handlers for database-related events. You can share a single data module among several different forms and provide different user

interfaces for the same data; there might be one or more data-input forms, reports, master/detail forms, and various charting or dynamic output forms.

The logical three-tier approach solves many problems, but it also has a few drawbacks. First, you must replicate the data-management portion of the program on different client computers, which might hamper performance, but a bigger issue is the complexity this adds to code maintenance. Second, when multiple clients modify the same data, there's no simple way to handle the resulting update conflicts. Finally, for logical three-tier Delphi applications, you must install and configure the database engine (if any) and SQL server client library on every client computer.

The next logical step up from client/server is to move the data-module portion of the application to a separate server computer and design all the client programs to interact with it. This is exactly the purpose of remote data modules, which were introduced in Delphi 3. Remote data modules run on a server computer—generally called the application server. The application server in turn communicates with the DBMS (which can run on the application server or on another dedicated computer). Therefore, the client machines don't connect to the SQL server directly, but indirectly via the application server.

At this point there is a fundamental question: Do we still need to install the database access software? The traditional Delphi client/server architecture (even with three logical tiers) requires you to install the database access on each client, something quite troublesome when you must configure and maintain hundreds of machines. In the physical three-tier architecture, you need to install and configure the database access only on the application server, not on the client computers. Since the client programs have only user interface code and are extremely simple to install, they now fall into the category of so-called *thin clients*. To use marketing-speak, we might even call this a *zero-configuration thin-client architecture*. But let us focus on technical issues instead of marketing terminology.

3.5.2 Advanced DataSnap Features

There are many more features in DataSnap than I've covered up to now. Here is a quick tour of some of the more advanced features of the architecture, partially demonstrated by the AppSPlus and ThinPlus examples. Unfortunately, demonstrating every single idea would turn this chapter into an entire book (and not every Delphi programmer is interested in and can afford DataSnap), so I'll limit myself to an overview.

3.5.2.1 Master/Detail Relations

If your middle-tier application exports multiple datasets, you can retrieve them using multiple ClientDataSet components on the client side and connect them locally to form a master/detail structure. This will create quite a few problems for the detail dataset unless you retrieve all of the records locally.

This solution also makes it quite complex to apply the updates; you cannot usually cancel a master record until all related detail records have been removed, and you cannot add detail records until the new master record is properly in place. (Actually, different servers handle this differently, but in most cases where a foreign key is used, this is the standard behavior.) What you can do to solve this problem is to write complex code on the client side to update the records of the two tables according to the specific rules.

3.5.2.2 Using the Connection Broker

I've already mentioned that the ConnectionBroker component can be helpful in case you might want to change the physical connection used by many ClientDataSet components of a single program. In fact, by hooking each ClientDataSet to the ConnectionBroker, you can change the physical connection of them all simply by updating the physical connection of the broker.

3.5.2.3 More Provider Options

I've already mentioned the Options property of the DataSetProvider component, noting that it can be used to add the field properties to the data packet. There are several other options you can use to customize the data packet and the behavior of the client program. Here is a short list:

- You can minimize downloading BLOB data with poFetchBlobsOnDemand option. In this case, the client application can download BLOBs by specifying the FetchOnDemand property of the ClientDataSet to True or by calling the FetchBlobs method for specific records. Similarly, you can disable the automatic downloading of detail records by setting the poFetchDetailsOnDemand option. Again, the client can use the FetchOnDemand property or call the FetchDetails method.
- When you are using a master/detail relation, you can control cascades with either oftwo options. The poCascadeDeletes flag controls whether the provider should delete etail records before deleting a master record. You can set this option if the database server performs cascaded deletes for you as part of its referential integrity support. Similarly, you can set the poCascadeUpdates option when the update of key values of a master/detail relation can be performed automatically by the server.

- You can limit the operations on the client side. The most restrictive option, poReadOnly, disables any update. If you want to give the user a limited editing capability, use poDisableInserts, poDisableEdits, or poDisableDeletes.You can resend to the client a copy of the records the client has modified with poAutoRefresh, which is useful in case other users have simultaneously made other, nonconflicting changes. You can also send back to the client changes done in the BeforeUpdateRecordor .

- Finally, if you want the client to drive the operations, you can enable the poAllow-CommandText option. This lets you set the SQL query or table name of the middle tier from the client, using the GetRecords or Execute methods.

Chapter-4
NILKAN SOFT VERSION 1.0

4.1 Nilkan Soft 1.0 For The Users

In an otel , the Front Desk part is very important to control all the otel's functions. All the otel records are saved in the Front Desk department of an otel.Especially , in luxury and developed otels , there must be many functions to control the otel status. Like Rooms Status , Rezervations , Check-in , Check-out , Events in Otel , Reports and so on. To manage , record and control all these functions is very difficult in big otels. Therefore , the hotel rezervation programs are used.

An Otel Rezervation Program makes some general operations of an otel. And an otel rezervation program is the most important and most necessary part of an otel. This program provides to reach immediately to customer detailed information in otel , rezervations to customers and groups and agent who in otel and will come to otel , Rooms Status like full

or empty rooms , clean or dirty rooms , The expences of the customers in otel and outside customers , cashier details like cashier reports , events in otel , services in otel and so on.

Nilkan Soft 1.0 is an otel rezervation program and written on Delphi Programming Language. The records are saved in tables which prepared in Paradox 7.0. Paradox 7.0 is a useful database manage program of Delphi . All the components of interfaces are Delphi's own components.

The main menus in this program are Rezervation , Front Desk , Cashier , Rooms Management , Options , Menu and Help.(Shown in Figure1.1)

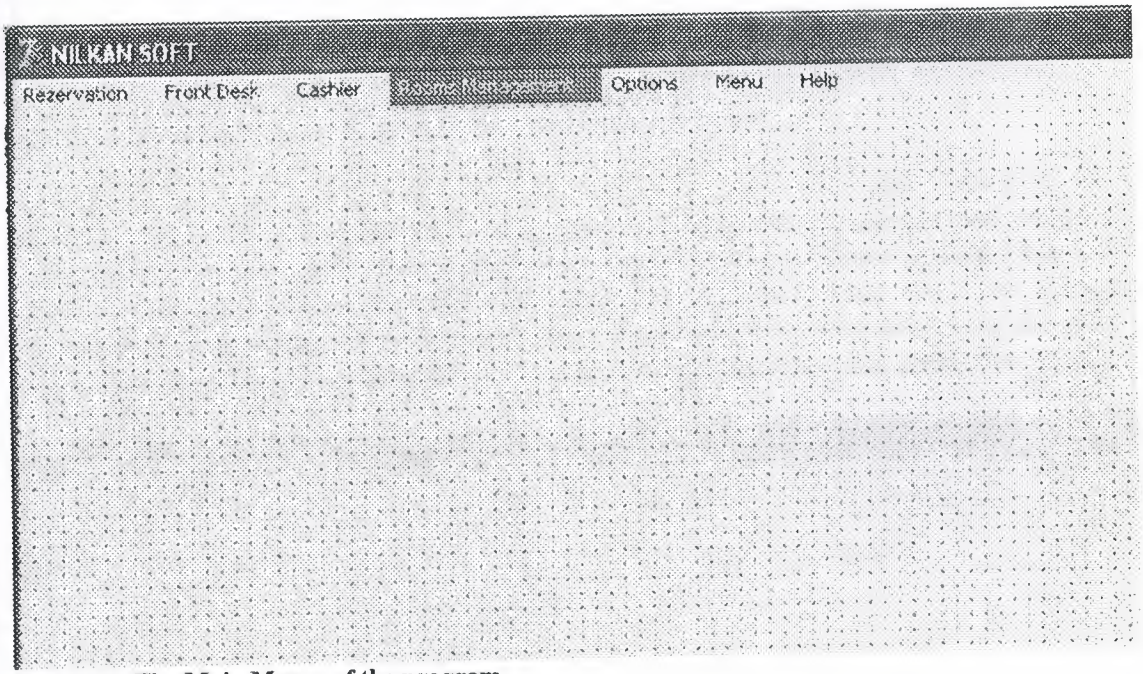


Figure1.1: The Main Menus of the program

In Nilkan Soft 1.0 there are many menus to control otel functions easily. These menus can be change according to otel's status.In this program the main menu is rezervation menu. In this menu there are individual , group and agent rezervations(Figure-1.2).

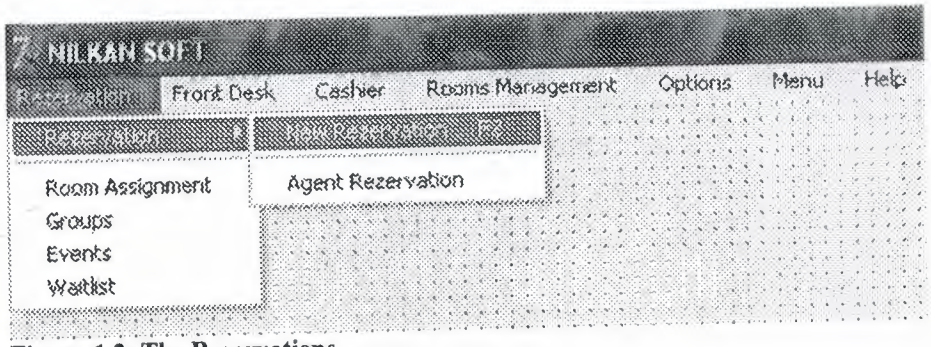


Figure-1.2: The Rezervations

If the user want to make a individual rezervation then will enter Rezervation main menu and then choose New Rezervation menu. Besides there is a shortcut of this menu which is F2. When the user choose this alternative , the program will open a new window to enter a member number to search in history database of the program whether there is an old record or not(Figure1.3).

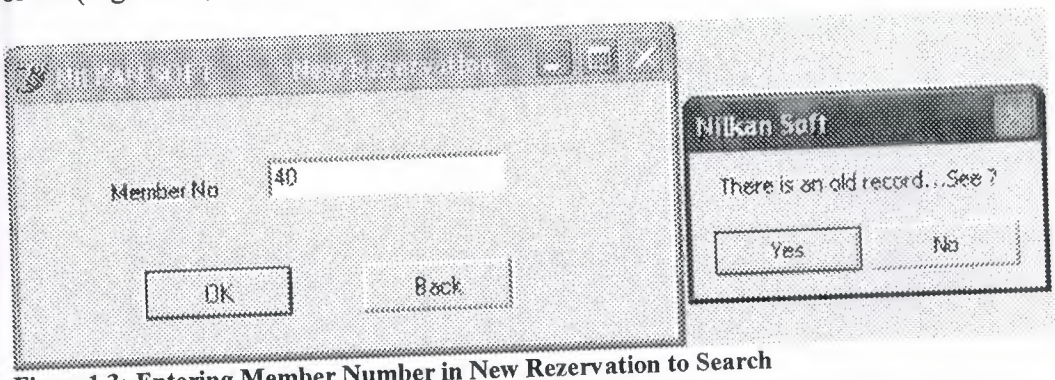
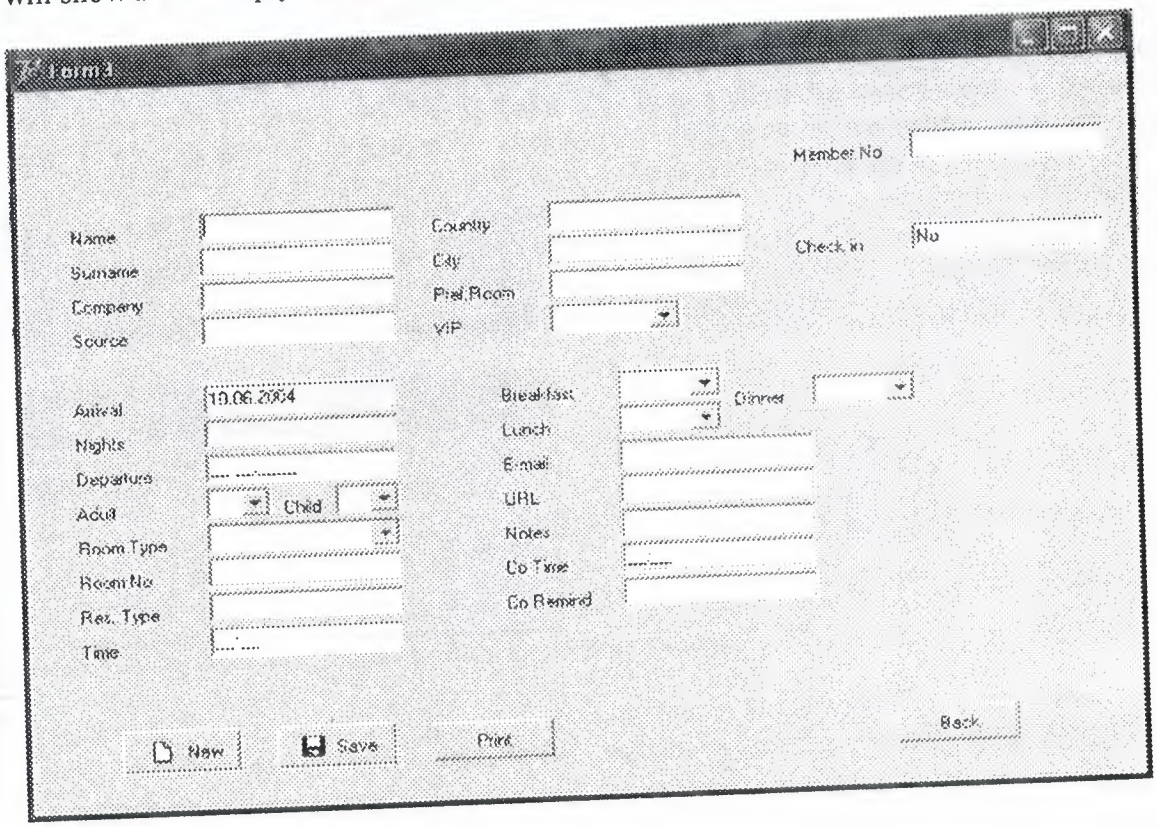


Figure1.3: Entering Member Number in New Rezervation to Search

If the system will find an old record then will give a message "There is an Old Record...See ?". If the user will choose ok then the program will open a new window and show the old record . If the user will choose Back then the program will go back to main menu. When the program search for old records , if doesn't find will show a message "There is no old record...New Entry ?" then if the user will choose ok then the program will show a new empty rezervation window else will go back to main menu(Figure1.4).



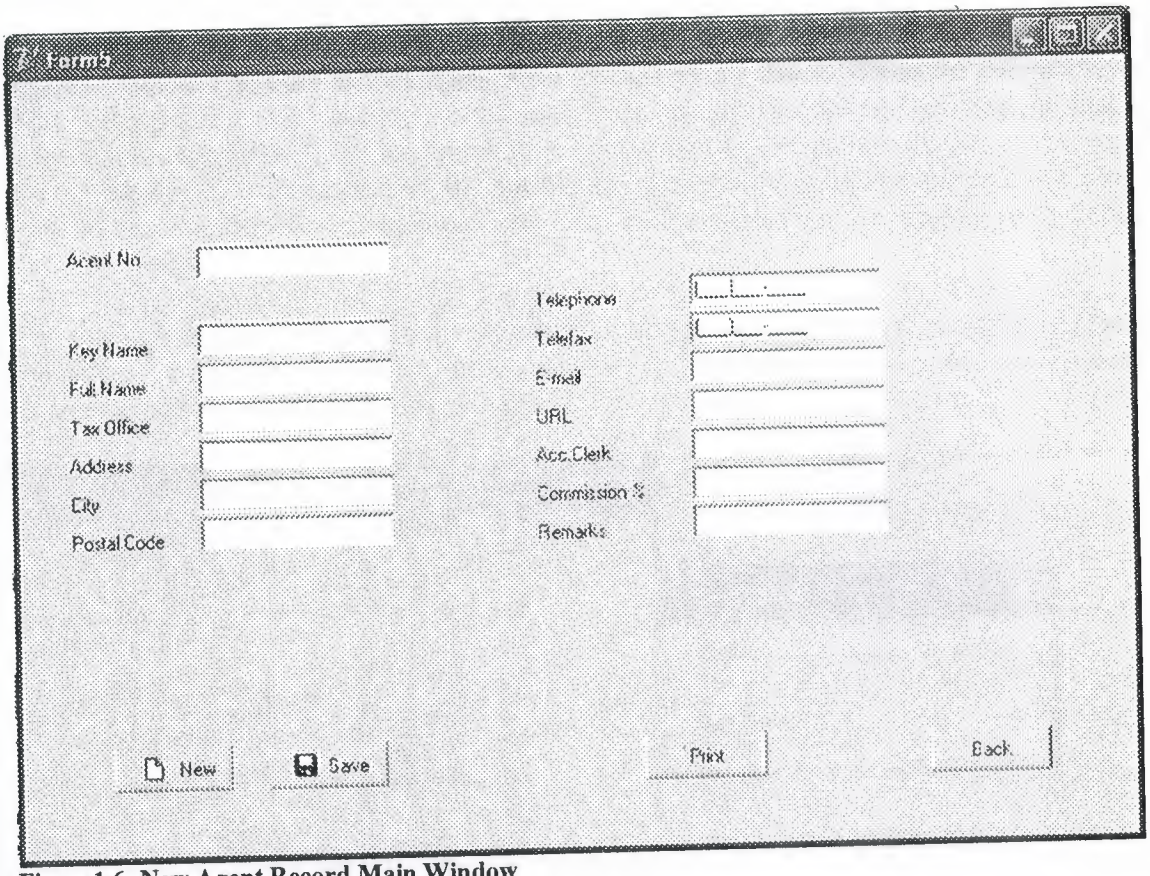


Figure1.6: New Agent Record Main Window

After the user make the reservation means that after click save button then on the agent reservation window there will be created a new button named as reservation . if the user want to reservations for entered agent then the user will click on this button . after the user click this button then the program will show a message "Do yo want to make reservation for this agent" .If the user choice ok then the program will give an input box to enter number of persons of this agent . after the user enter number of persons the program will open normal reservation window to enter each customers information and will repeat this operation until the given number(Figure1.7) .

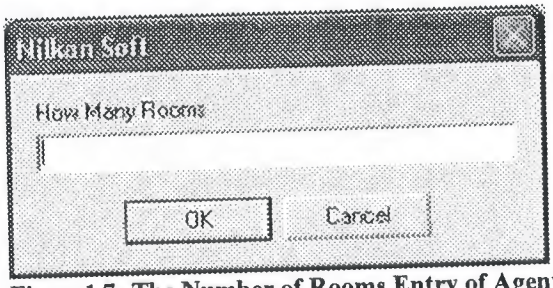


Figure1.7: The Number of Rooms Entry of Agent Rezervation

In new agent rezervation the user will enter the agent's agent number , key name , full name , tax office , adress , city , postal code , telephone , telefax , e-mail , URL , acc. Clerk , commission , remarks.

In this window there are four buttons. New , Save , Print , Back .When the user click new button the program will clear all the components on the window and provides to make a new agent rezervation .If the user click save button then the program will save all entries on the window to the database . If the user click print button then the program will print entered agent information . when the user click back button then the program will go back to main window .

The third reservation type is groups rezervation . the groups reservation provides to reserve the groups . when the user choices group reservation , the program will open a new window to enter group number(Figure1.8) .

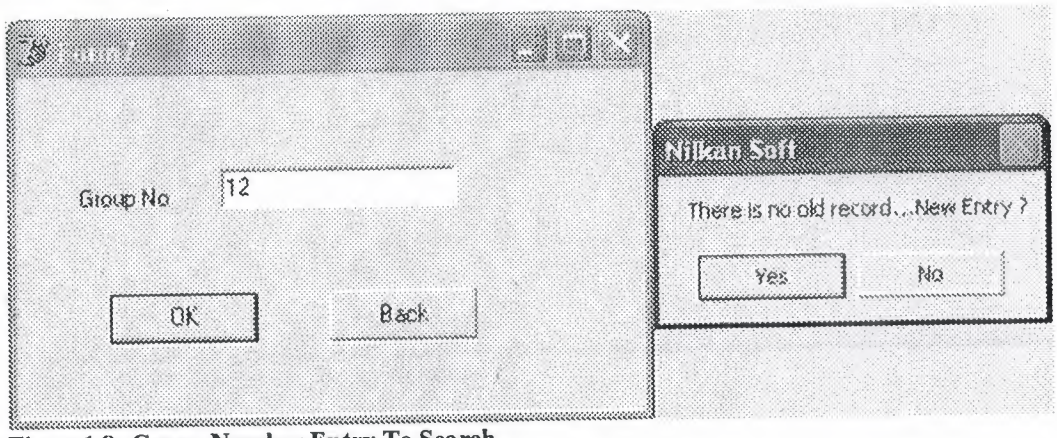


Figure1.8: Group Number Entry To Search

Then the program will search whether there is an old recorded group .If the program will not find the entered group number then will give a message "There is no old record...New entry ?" . If the user will choice ok then the program will open a new window to reserve the group . If the user will choice no then the program will go back to the main window. When the program search for old records , if find will show a message "There is an old record...See ?" then if the user will choice ok then the program will open group reservation window with old records . If the user will choose no then the program will show a new empty rezervation window(Figure1.9).

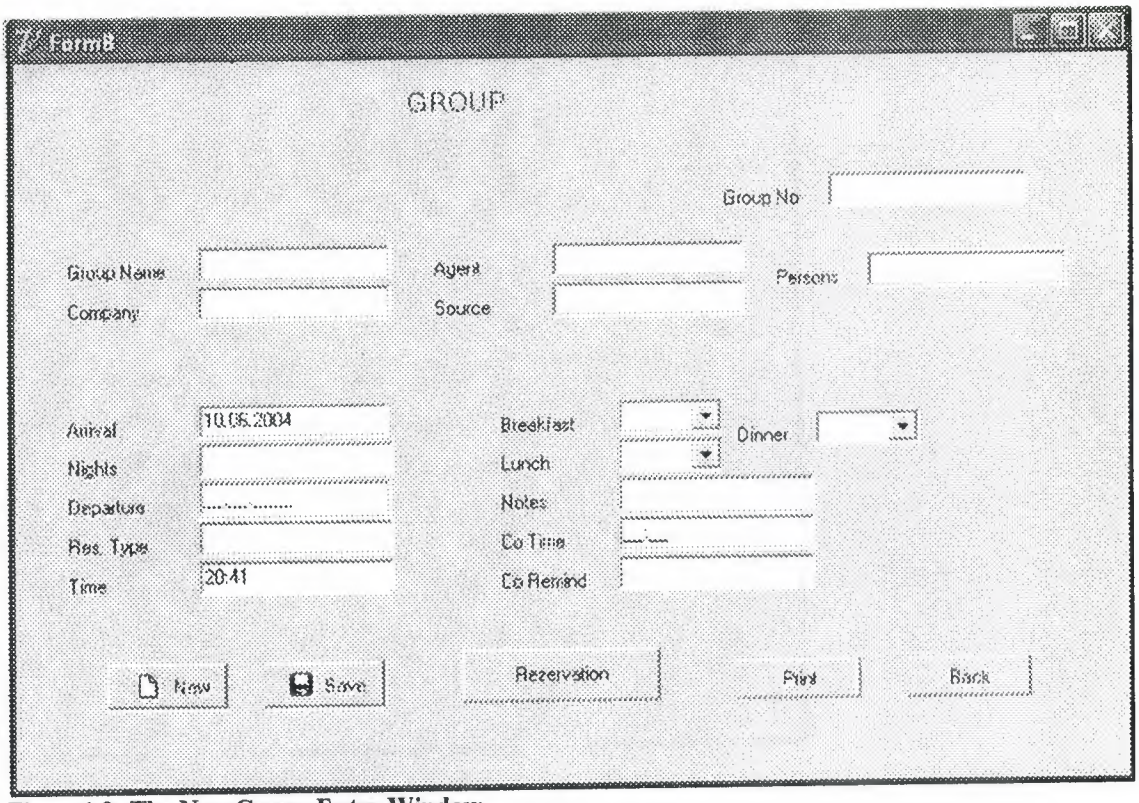


Figure1.9: The New Group Entry Window

In this window there are five buttons. New , Save , Print , Back , Reservation .When the user click new button the program will clear all the components on the window and provides to make a new group rezervation .If the user click save button then the program will save all entries on the window to the database . If the user click print button then the program will print entered group information . when the user click back button then the program will go back to main window . If the user will click reservation button then the program will take the number of persons and will open the normal reservation window to enter each customers information and will repeat this operation until the given number .

One of the reservation menus is room status . room status alternative provides a quick show the customer's room number according to room number or member number .when the user choice room status , the program will give a window to find the customers room number(Figure1.10) .

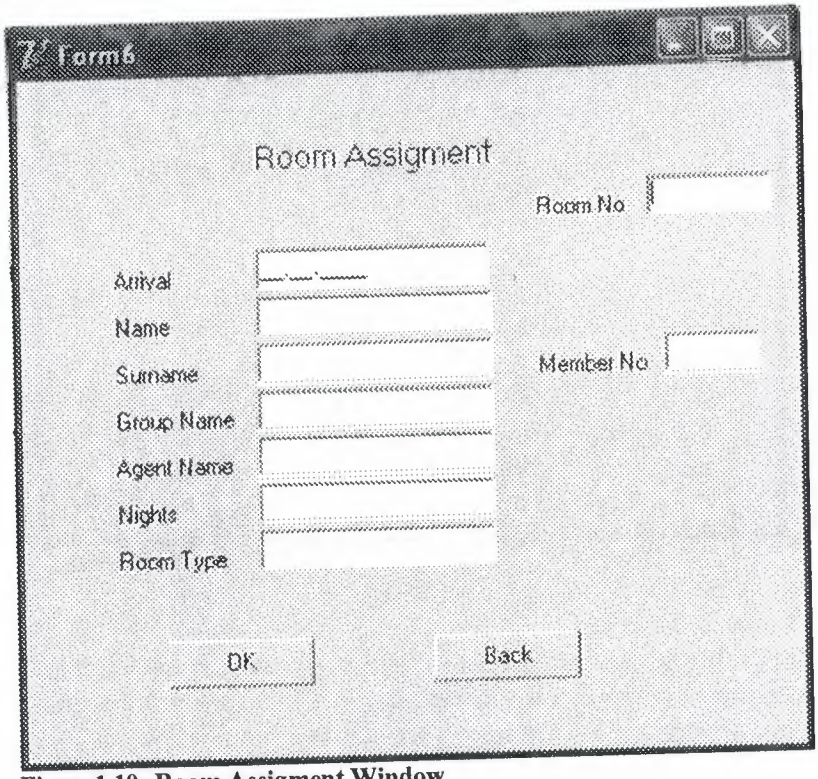


Figure1.10: Room Assigment Window

In this window there are two choices to search . One of them is room number and another one is member number . The user has to choose one of them of these alternatives . If the user will enter both of them then the program will accept room number . After the user will enter information then the program will search and if it find then will show the arrival time , name , surname , group name (if any) , agent name (if any) , days of stay in otel and room type of customer .

Another reservation choice is events . When the user click events then the program will open a new window(Figure1.11) to show, to add and to delete events . When the events window open , the user can see the events with its date and time . these are listed row by row . When the user delete one of them , he/she has to choice one row and then click delete button . then the program will show a message "are you sure to delete" . then the user will click ok.

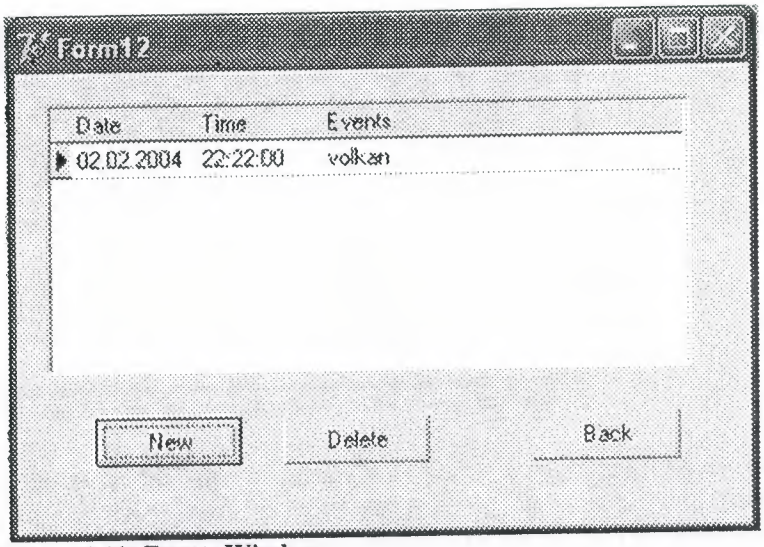


Figure1.11: Events Window

If the user wants to add a new event then has to click new button when the user click new button , the program will open a new window(Figure1.12) to enter the new event . on this window the user will enter the event's date , events time and the event then will click save button . after this operation the program will go back the events window and the user also can see the new entry .

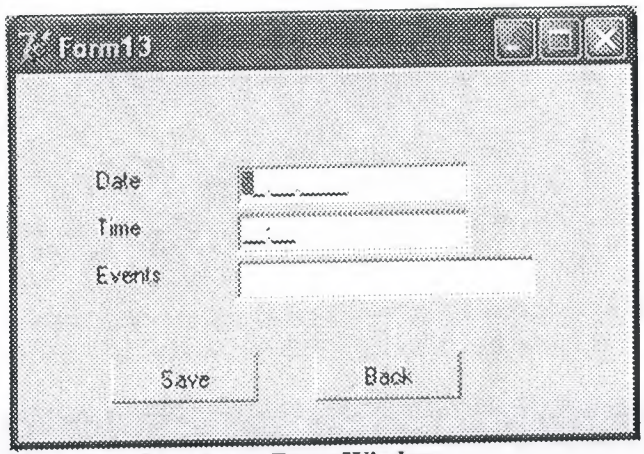


Figure1.12: New Event Entry Window

And the last rezervation choice is the Waitlist. Waitlist is list the waiting customers to reservation.This choice is used when the otel rezervations are full. When the otel is full , the customers are taken to the waitlist .

When the user choose waitlist option then the program will open a new window (Figure1.13) to take name and surname to search whether there is an old record in the waitlist table.

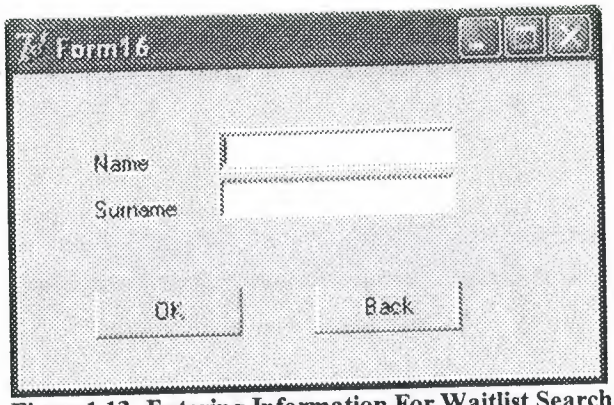


Figure1.13: Entering Information For Waitlist Search

If there is then the program will show a message "There is an old record..." . Then the program will show the old record . If there is no old record the program will show a message and will open a new window(Figure1.14) to enter new waitlist entry.

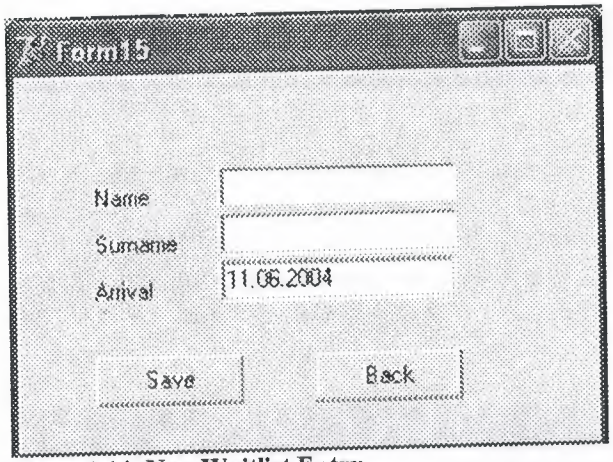


Figure1.14: New Waitlist Entry

Another main menu option is Front desk option(Figure1.15) . In Front Desk menu there are Arrival , In house , Profile , Accounts , Messages .

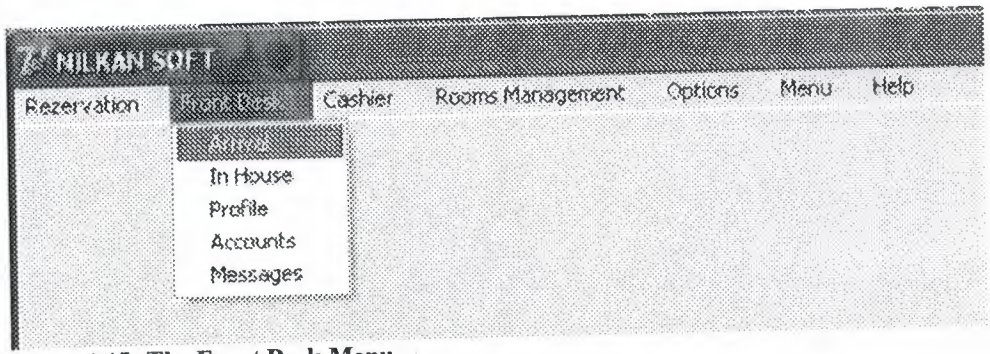


Figure1.15: The Front Desk Menu

After the customer's rezervation , the check-in operation has to made . This operation is determine actually the customer's arrival to the otel.For this check-in operation the Arrival menu is used.

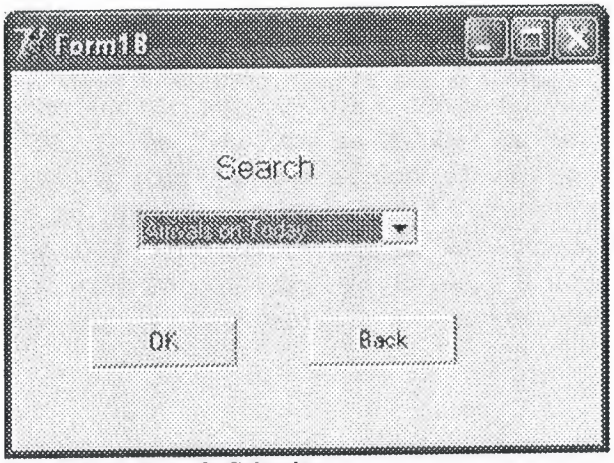


Figure1.16: Search Criterias

If the user choose Arrival menu to make check-in operation then the program will open a new window to search choices(Figure1.16).In this window , there are two choices , like "All Arrivals" and "Arrivals On Today".After the user choose one of the these alternative then the program open a new window again same as previous window and the choices are the "All Arrivals" , "Member No" , "Name" . The user of course choose one of these alternative and will continue by clicking the OK button.

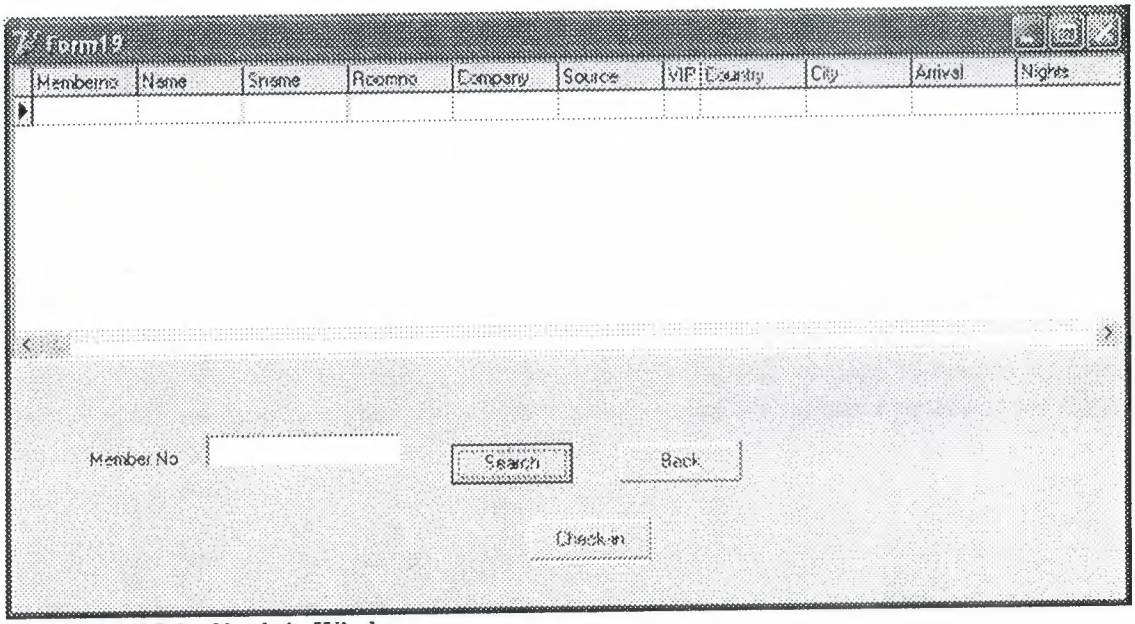


Figure1.17: Main Check-in Window

After this operations the program will open the main Check-in window(Figure1.17).In this window there are a table which shows the records , some buttons , a text reader with heading according to last window choice. For example if the user choose "Member No" there is a text reader with heading Member Number. In this window the user will enter the information about the user according to choice. After the user enter the data then the program will search whether there is a record in the rezervation records .

If the program will find a record then also will show this record's detail in the table which is on the main Check-in window. There is a button named as Check-in. When the pointer on the record , if the user wants the check-in operation of that customer then will click on that button. And then the program will show a message like "Are You Sure Want to Check-in" . After the user accept this message then the program will make the check-in operation. And there is a Back button on the window . The user can go back by using this button.

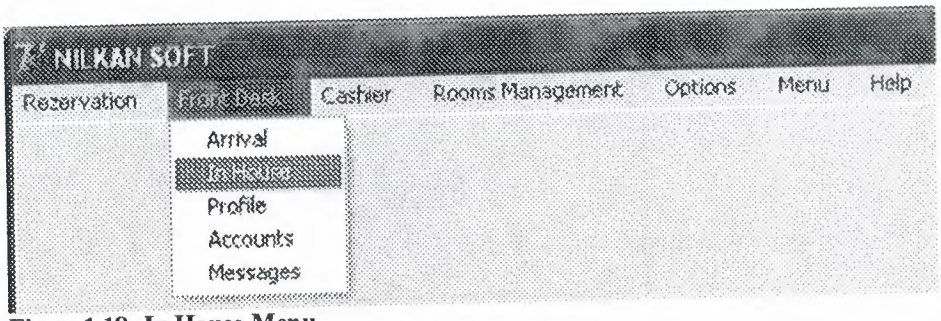


Figure1.18: In House Menu

Another Front Desk option is "In House" choice(Figure1.18). This alternative is used to see , control or update the checked-in customers. When the user choose this alternative then the program will open a new window to enter the customer's information according to choice(Figure1.19). These choices can be "Room Number" , "Member Number" , "Name" , "Show All" . According to choice the program will open a text reader to enter the data by user. After the user will enter the data then the program will search whether there is an record in the checked-in customers. If the program will find then will show record in the table on the window. After these operations the program will provide to user to update the record's information from the table. The user can make the update operations by clicking the update button.

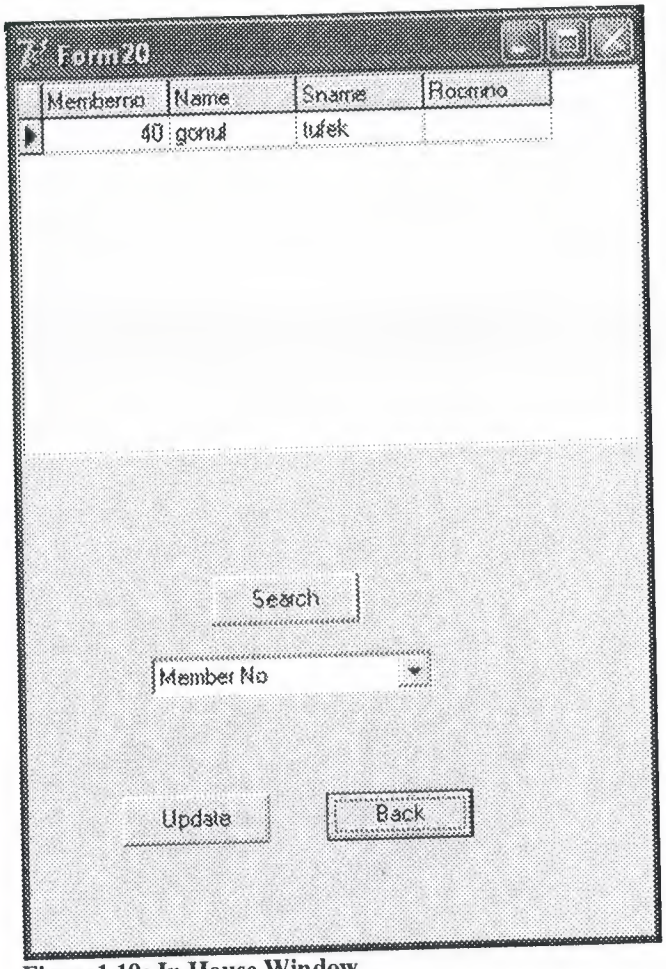


Figure1.19: In House Window

Of course one of the important part of an otel is the old costumers that is the checked-out customers. The program saves the all stay in otel customer's details. The user can see the old customers by using the profile option(Figure1.20).

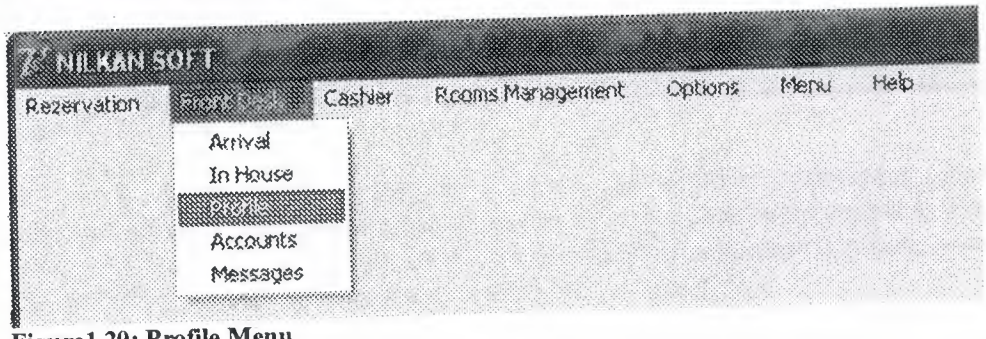


Figure1.20: Profile Menu

In the profile window there is a table to show the customer's details and some buttons to apply some changes on this table. These buttons are "Search" , "Update" , "Delete" , "Back" . After the user enter the this alternative then select a choice to enter the search data. These choices can be "Name" , "Group Name" , "Show All" (Figure1.21).

When the user enter the data and click the search button then the program will search the data according to choice in the history table of the otel. And if the program will find then show the record on the table on the Profile window. The user can update or delete this record by using buttons. The program provides to update operation on the table. Another button is the back button and this button allow to user to go back to the main menu.

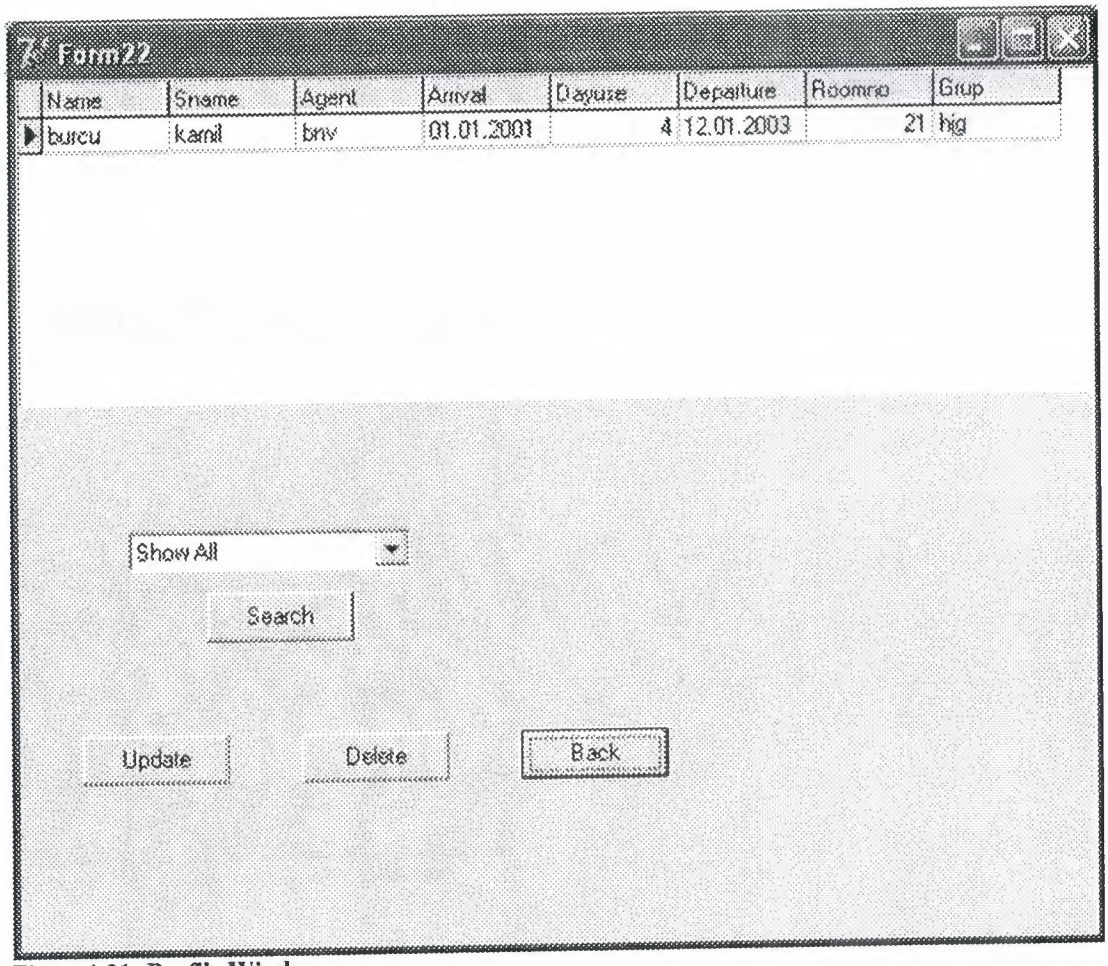


Figure1.21: Profile Window

In Front Desk menu options , there is another important part named as Accounts(Figure1.22). Sometimes there can be some customers comes to otel from outside to use the otel's activities , not to stay in otel. These customers can have some expences in the otel. Accounts of these customers are saved in this option.

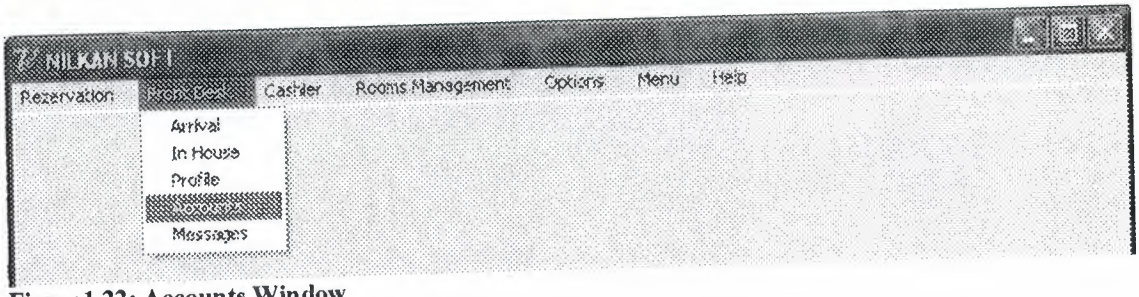


Figure1.22: Accounts Window

When the user make an account record , to see , add or delete an account then use this option. In this alternative the user will see a window with some choice buttons(Figure1.23). There are three choices button on the window. "Search" , "New" , "Back" buttons.

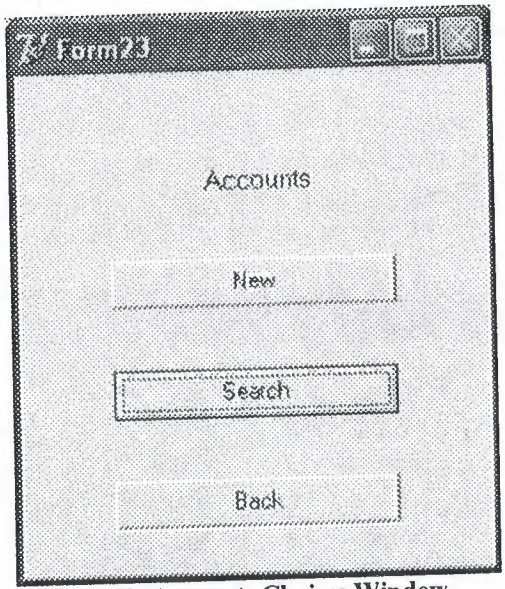


Figure1.23: Accounts Choices Window

When the user choose Search selection then the program will open a new window to show the accounts(Figure1.24). In this new window there is an table to show the accounts and four buttons. These buttons are "Add Account" , "Delete" , "Update" , "Back" . Add Account button provides to user to add new account to the selected customer on the table. When the user choose Add Account the the program open a new window to enter the amount of new excess. And this new account is added to the Account table on the previous table. Update button provides to user to make some changes on the selected record. And the Delete button make delete operation of the selected item on the table. When the user wants to go back to the main menu then will use the Back button.

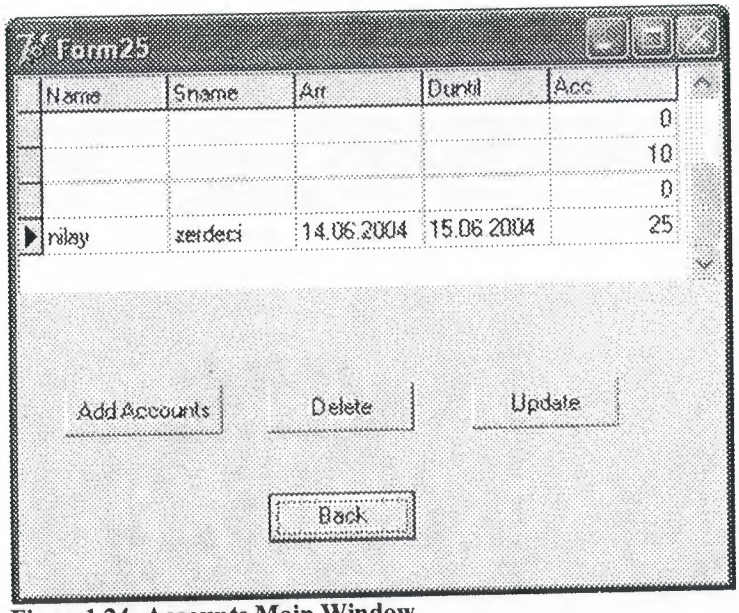


Figure1.24: Accounts Main Window

When the user choose New option the program will open a new window to take the customer informations like Name , Surname , Date , Account Until(Figure1.25). After the user enter these datas then the program will save this new account and provides to enter new acoounts to the this customer.

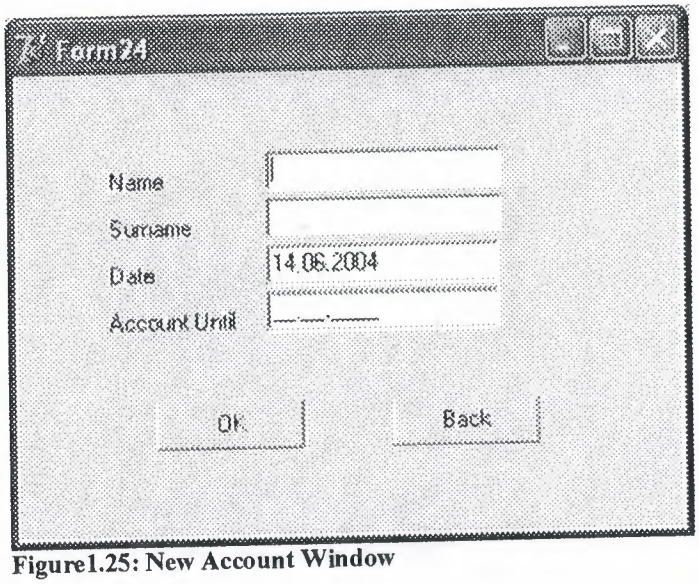


Figure1.25: New Account Window

If the user click the Back button then the program will go back to the main menu window.

Another Front Desk option is Message option(Figure1.26). In an otel there can be some messages to the customers , maybe from a person or with a call. These messages are saved in the program and the user can see and control these messages from the program.

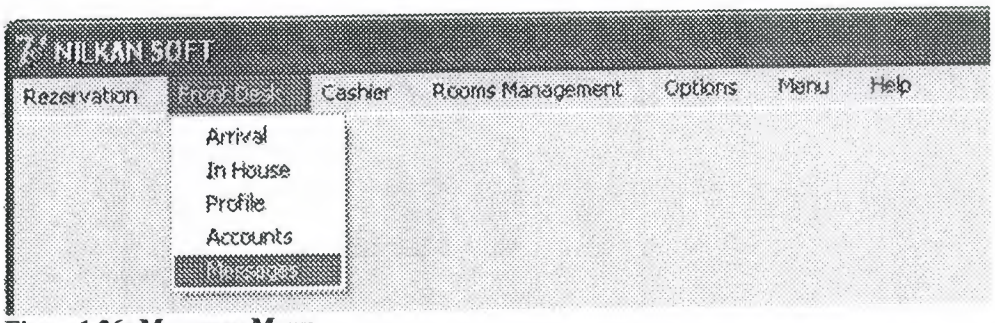


Figure1.26: Messages Menu

When the user choose Messages option then the program will open a new window (Figure1.27). In this window there are some choices about operations on the messages. There are three buttons. "New" , "Search" and "Back" button.

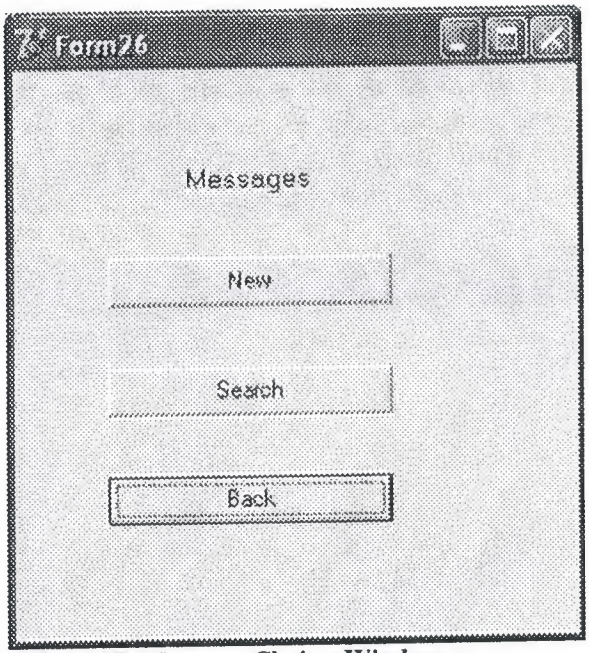


Figure1.27: Messages Choices Window

When the user wants to write a new message to a customer then will choose the New option. After the user choose the New alternative then the program will open a new window to enter the message information to the system(Figure1.28). In this window the program will take information as message to , message from , message date , message time and message contents. After the user enter these information then save by clicking to the OK button. And also can go back to the main menu by using the Back button.

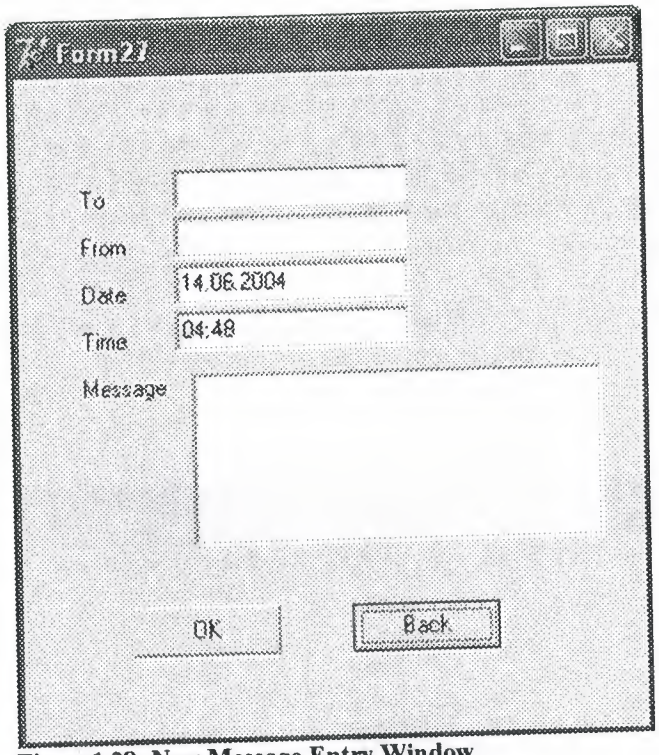


Figure1.28: New Message Entry Window

When the user chooce Search alternative then the program will open a new window and in this window there is a table to show the messages and "Update" , "Delete" , "Back" options(Figure1.29).With Update and Delete option the user can make changes or delete to the selected message and with Back option the user can go back to the main menu.

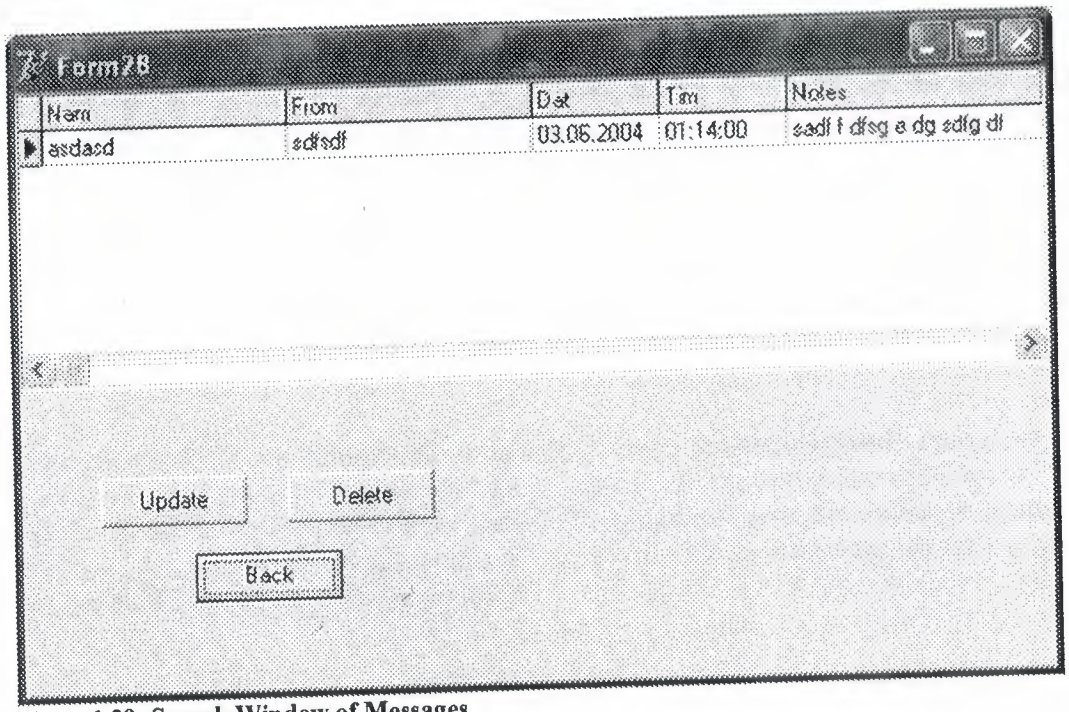


Figure1.29: Search Window of Messages

In an otel the most important part is cashier part. Because all the otel's accounts and reports is saved by the cashiers of otel. Generally in otels , there are cashier numbers or usernames and cashiers makes operations with this information. In otels some important operations can make only with cashiers that is another personnel cannot make those operations. Therefore , the cashier operations are the most important operations in an otel.

In this program's main window , there is a cashier menu which is made carefully(Figure1.30). Cashier inputs and outputs are the most important parts of this menu.

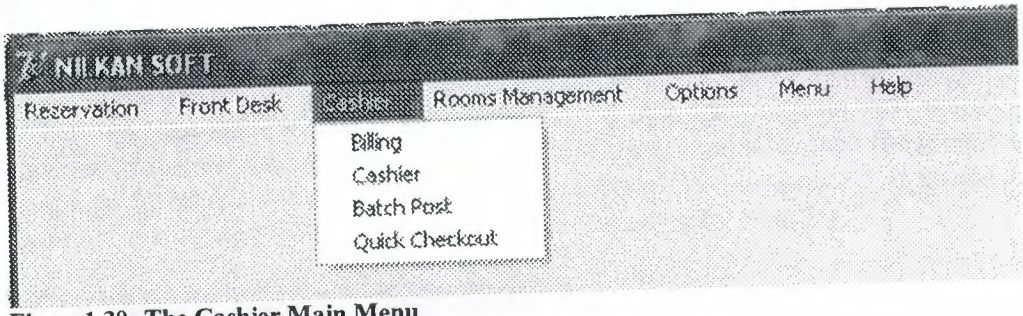


Figure1.30: The Cashier Main Menu

In the program , in Cashier option , there are four options. "Billing" , "Cashier" , "Batch Post" , "Quick Checkout" .

Billing operations in Nilkan Soft 1.0 is prepared carefully. When the user make new rezervation , the program will open new billing account Database tables for each customers seperately. And this Database table is balanced and deleted when the customer is checked-out. All the customers' accounts are saved in these tables. When a billing operation make , first at all the user will find the customer and then will enter the billing information. During this operation the program will open the customer's special billing account and will write to there. In Nilkan Soft 1.0 , to make this operations , there must be a cashier. The program always wants to a cashier name and password before making these operations.

When the user choose Billing option from the cashier submenus then the program will open a new window to allow to user to enter a cashier name and password(Figure1.31). This cashier name is not cashier's name really. The program wants the cashier's username from the user. Also these information are defined during the cashier entry to the system which will be explained in the Cashier part.

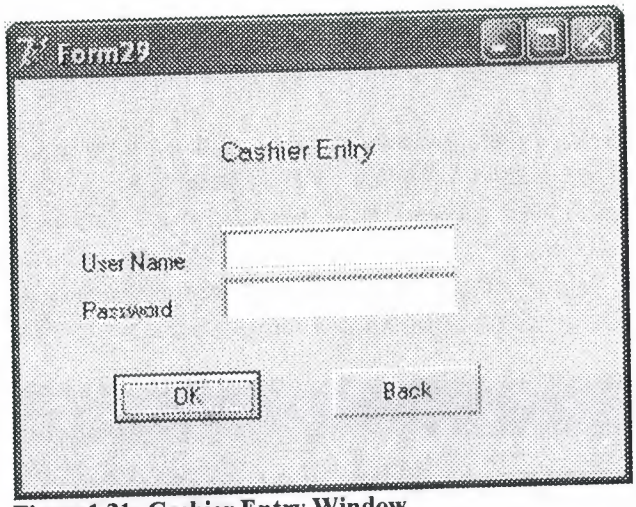


Figure1.31: Cashier Entry Window

When the user cashier username and password then the program will search whether the cashier's information are true or not. If the information is false then the program will give a message "The Username or Password is Wrong!!!Try Again..." (Figure1.32)and after giving this message then will go back to the cashier entry window.

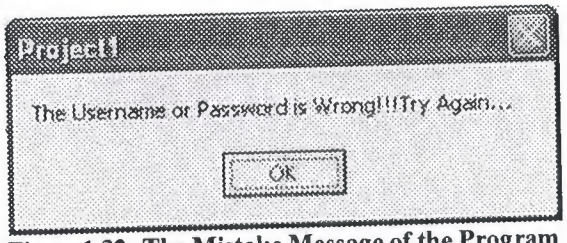


Figure1.32: The Mistake Message of the Program

If the program will accept the cashier's information then the program will open a new window also that is the cashier's username and password is true(Figure1.33). In this new window there are some choices on the window as "Member Number" , "Room Number" , "Name" . Because of these options is to make search to find the customer. As we mentioned before , before the billing operation the user has to find the customer. Therefore the user will choose an search criteria.

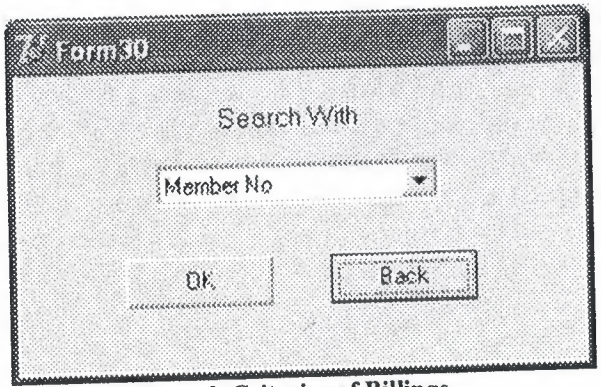


Figure1.33: Search Criterias of Billings

After the user choose an alternetive then according to selected alternative the program will open a new window(Figure1.34). In this window there is a table to show the wanted customer , a text reader with heading according to selected alternative from the previous window and OK and Back buttons. When the user enter the information about the customer then the program will search for the customer. If the program will find the customer then will show the customer details on the table which is on the window.

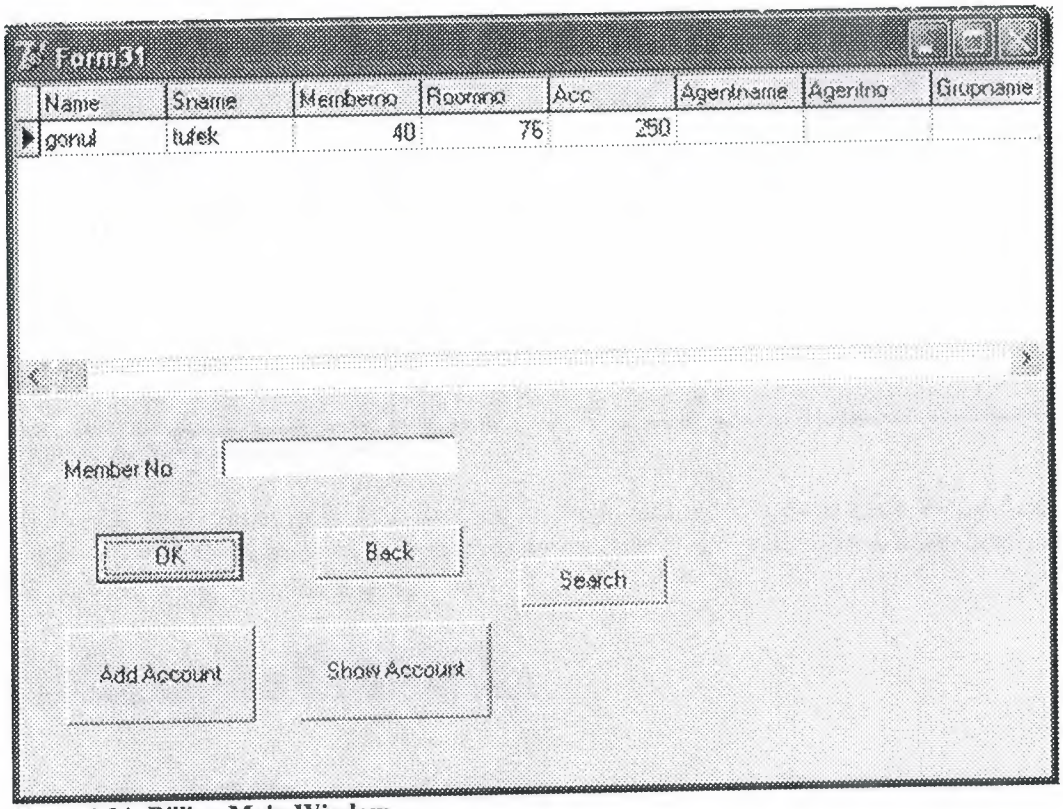


Figure1.34: Billing Main Window

After these operations the program will add three buttons on the window which are named as "Search" , "Add Account" , "Show Account" . When the user click search button then the program will open the previous search window(Figure1.35). If the user wants to see the customer's account details then will click to the Show Account button.After the user will click to the Show Account then the program will show the customer's accounts details on another table but on the same window. In this table there are the customer's account details and also cashier name part. This cashier name part shows which cashier make the billing.

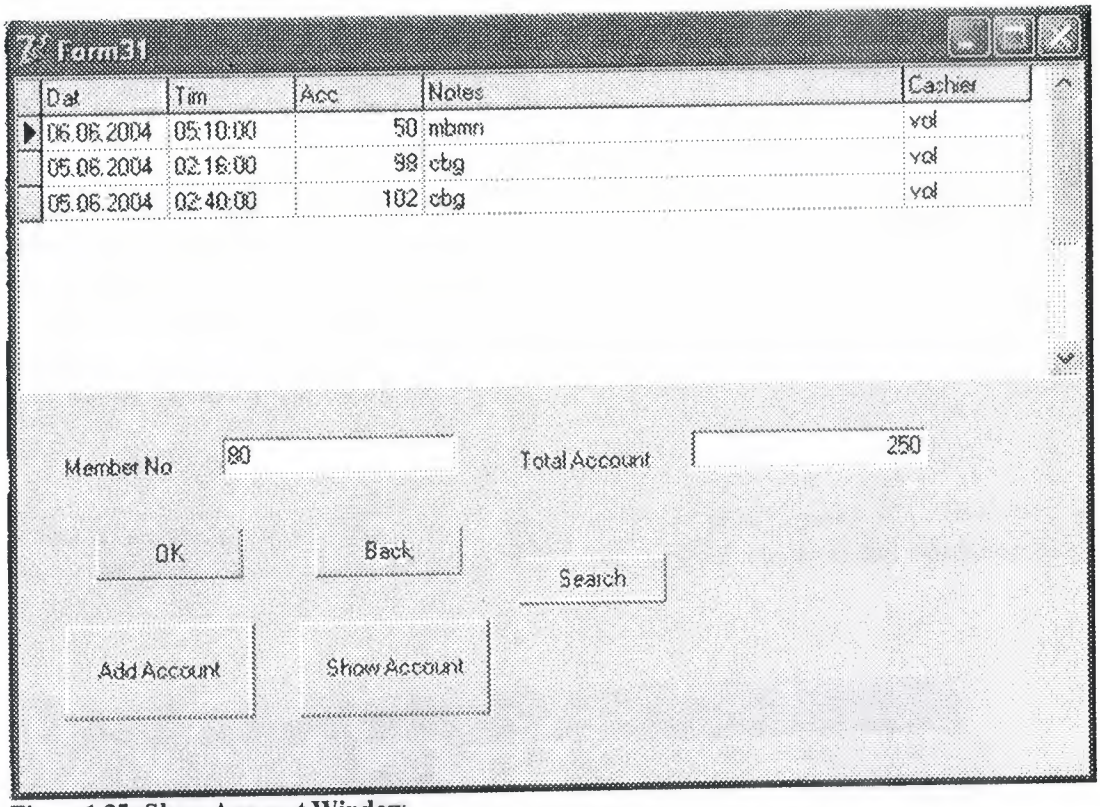


Figure1.35: Show Account Window

When the user wants to add an account to the customer then will click to the Add Account button. After user choose add account then the program will open a new window to enter new account information(Figure1.36).

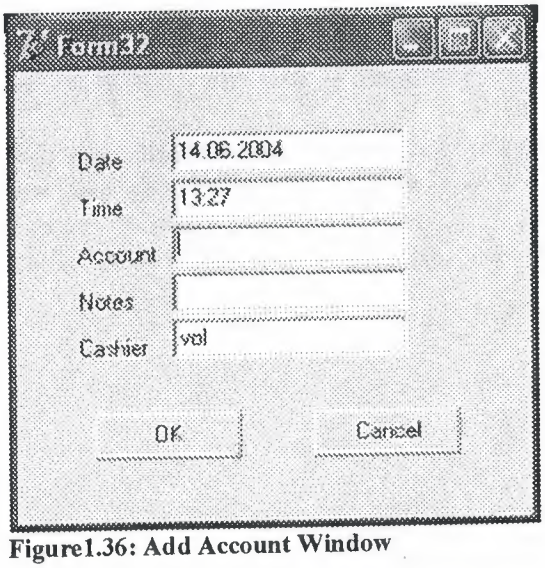


Figure1.36: Add Account Window

In this window date , time and account parts are put by the program. The user also can change these information but cannot change cashier part. Which cashier enter billing part of the program , he/she has to make billing operations. Therefore the cashier part of new account entry cannot be changed by the user. After the user enter information then the program will add account to the customer and after the user click Show Account part again then can see added account on the table.

Another submenu of cashier main menu is Cashier part(Figure1.37). This part is very important part of the otel and program. All the cashier information and operations are made in this part.

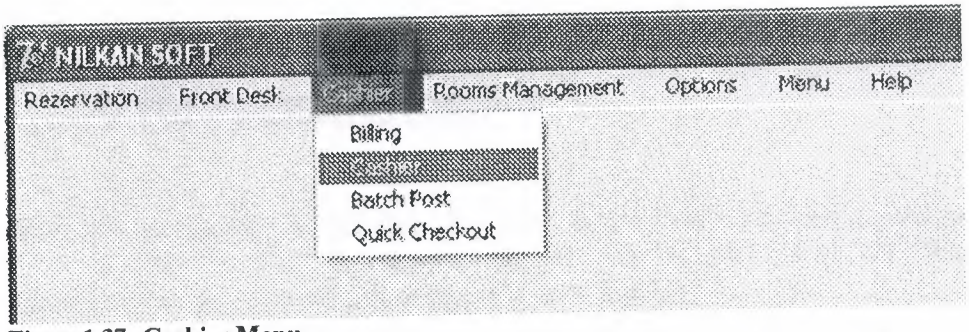


Figure1.37: Cashier Menu

When the user choose cashier then the program will open a new window(Figure1.38).There are five choices on this new window. "Cashier Reports" , "Change Password" , "New Cashier Report" , "Delete Cashier" , "Back" . In Cashier Reports , the program shows the cashier's operations. The user can see all the cashier account from here.

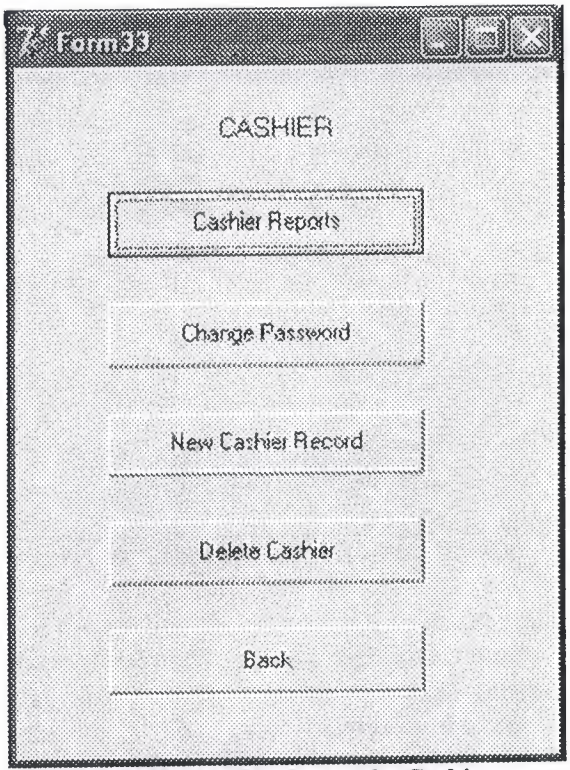


Figure1.38: Choices Window of the Cashier

After the user enter Cashier Reports then the program will open a new window and will show on this new window , general cashier information(Figure1.39). There are also two buttons named as "Show Cashier Report" and "Back" .

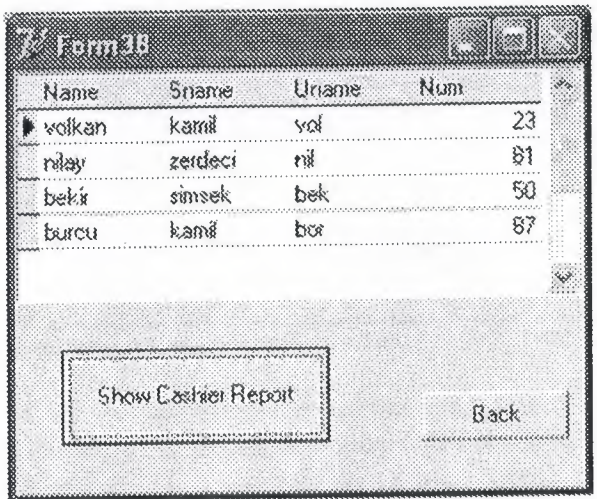


Figure1.39: Cashier Information Table of Cashiers

The user can see the cashiers' name , surname , username and number from this window. When the user click Show Cashier Report then the program will show the selected cashier's report(Figure1.40).

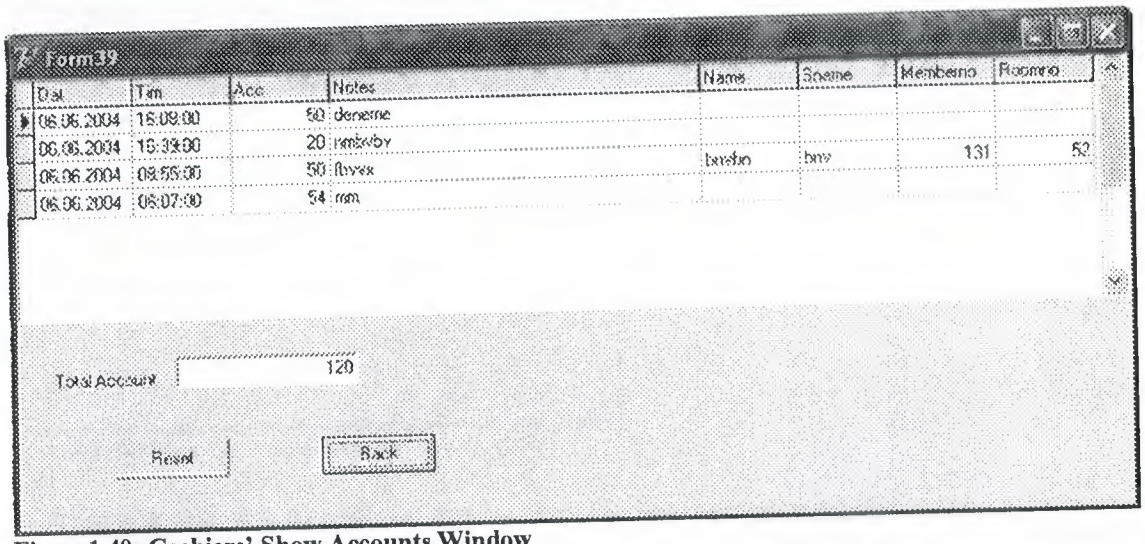


Figure1.40: Cashiers' Show Accounts Window

On the cashier report part the user can see cashier's operations with date , time , amount of excess(account) , notes , customer's name , surname , member number , room number. And there is text which shows the total account of the cashier. The reset button resets all the account of the cashier. Generally this operation is made at the end of the day. The user can go back to the previous window by using "Back" button.

The Change Password part of the cashier part is used to change the password of cashiers. But only the cashier or authority personnels can make this operation. When the user choose this part then the program will open a new window to take the cashier's username(Figure1.41).

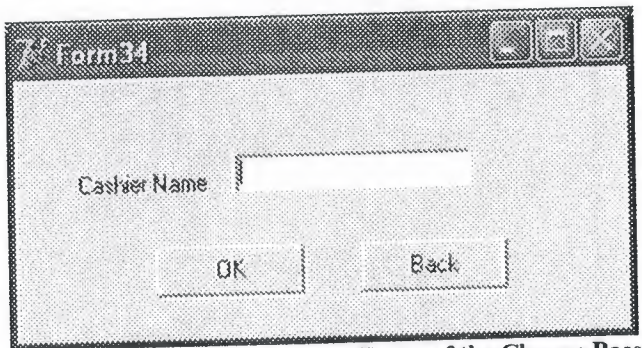


Figure1.41: Cashier Username Entry of the Change Password

After the user enter the cashier username then the program will search the entered cashier username , if it will find then will open a new window to enter new password information (Figure1.42). If there is no cashier named as entered text , then the program will give a message and then wants to enter cashier's username again.

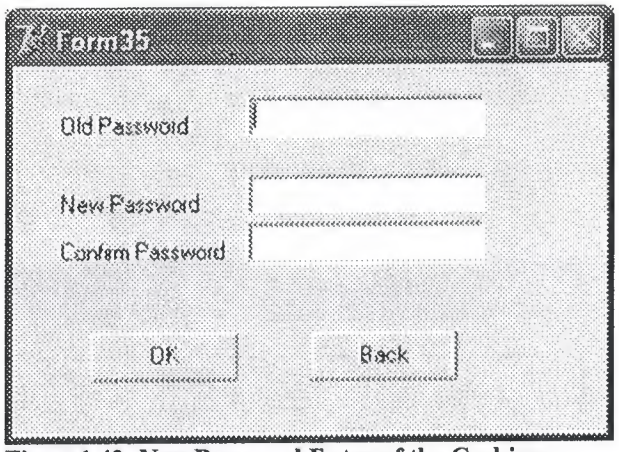


Figure1.42: New Password Entry of the Cashiers

If there is a cashier that is the entered text is true then the program will open a new window and in this window there are three text readers.First one is to enter the Old Password , second one for New Password and last one is to confirm the new password. Then the program will search for the old cashier password.If the old password is true then the program will check new password and confirm password parts on the window is same with eachother or not. If the old password is wrong or entered new passwords are not same with eachother then the program will give a message. Then the program wants to user to enter information again.

In New Cashier Record , the program saves new cashiers. When the user choose this part then the program will open a new window to enter the new cashier's information(Figure1.43).

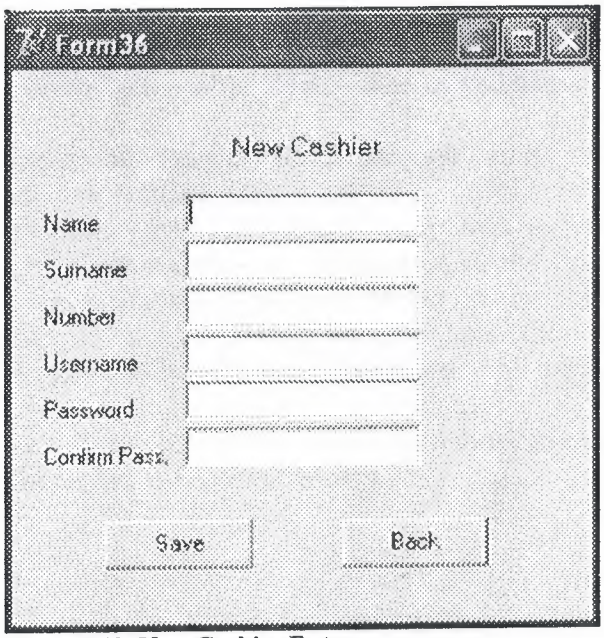


Figure1.43: New Cashier Entry

In this window , the user enter new cashier's name , surname , number of the new cashier , username , password and confirm password part. After the user enter these information then the program will check the entered datas and will save the new cashier's record.

Another cashier operation part is Delete Cashier. This operation is used to delete one of the old cashiers.

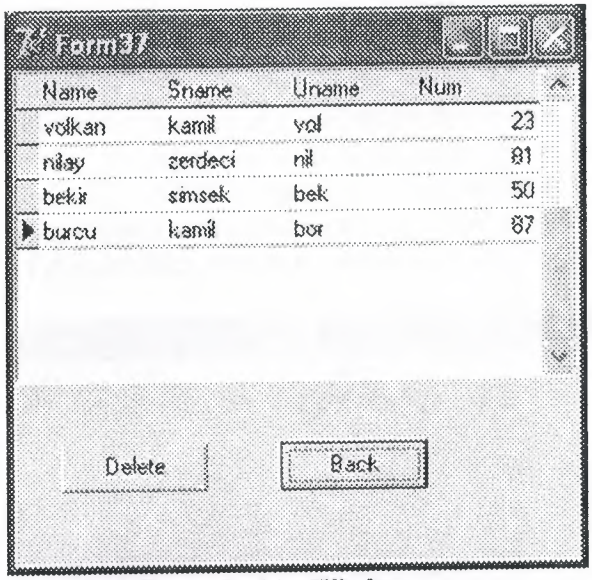


Figure1.44: Delete Cashier Window

When the user choose this menu then the program will open cashier information with two choices "Delete" and "Back" (Figure1.44). On this window , there is a table and information can seen from this table. The user can use Delete button to delete the selected cashier on the table. Back button as you know , will go back to the previous window.

Sometimes there are some customer came to the otel with a group or agent. Generally , the same operations are applied to all customers of a group or agent. For example , a billing operation may be wanted to applied to the group that is all the customers of the group.Instead of enter information to the customers seperately , Nilkan Soft provides to enter to the group or agent. When the user make an operation to a group or an agent and then will enter necessary group or agent information and will make this operation. And the program will apply these operations to all the customers of the group or agent.

In Nilkan Soft , the user can make these operation by using the Batch Post submenu under the cashier menu(Figure1.45).

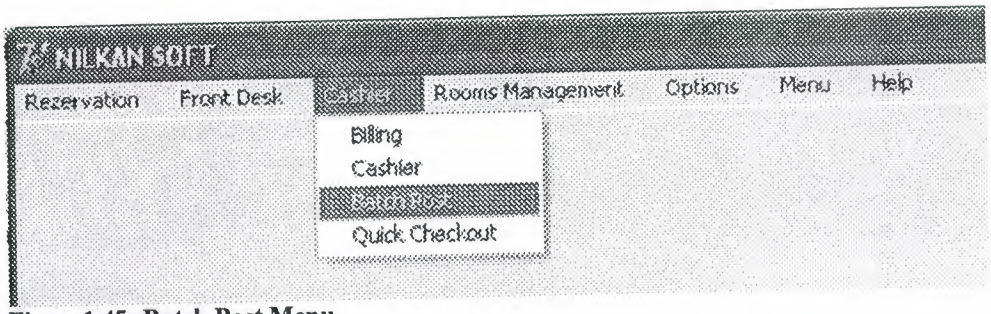


Figure1.45: Batch Post Menu

When the user choose this menu the program of course wants to cashier entry to make this operation. After the user enter cashier's information correctly then the program will ask expenses with an agent or a group. After the user choose one of the these alternatives then the program will open a new window and want Agent or Group Name according to choice of the previous window to make search(Figure1.46).

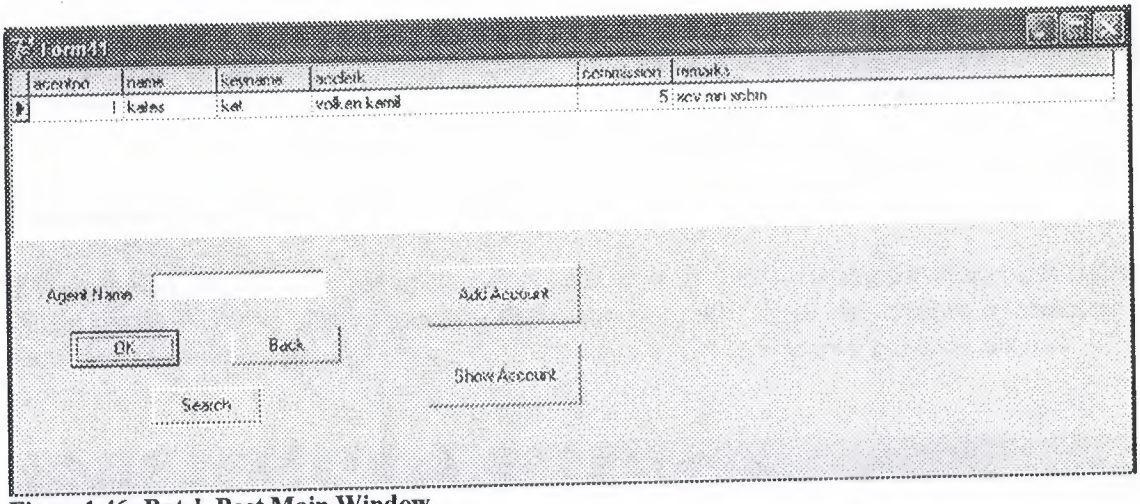


Figure1.46: Batch Post Main Window

After the user enter the name of the group or agent then the program will show the group's or agent's information on the table of this window . there are five buttons on the window named as "OK" , "Back" , "Search" , "Add Account" and "Show Account" buttons.after the user enter the name information then by clicking OK Button provides to make search by the program . The user can add new account by using add account button .When the user click add account button then the program will open a new window to take the new account information(Figure1.47).

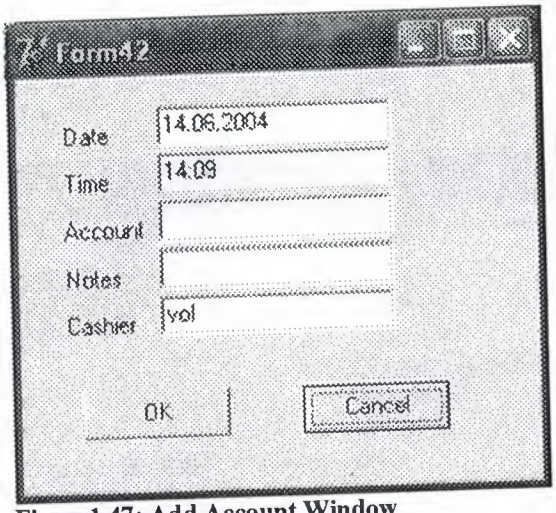


Figure1.47: Add Account Window

On this window the date , time and cashier information are put by the program . The user also can change date and time information but cannot change the cashier information . the program put cashier information according to the first cashier entery . Because only cashiers can make this are operation and who enter to the this part , he/she can make this operation . The user completes the name account entry by entering account and notes information on the window .

The user also can see selected agent or group's old account by using show account button . When the user choose show account then the program will show old account information of the selected group or agent(Figure1.48) . There is a text that shows the total account of the group or agent .

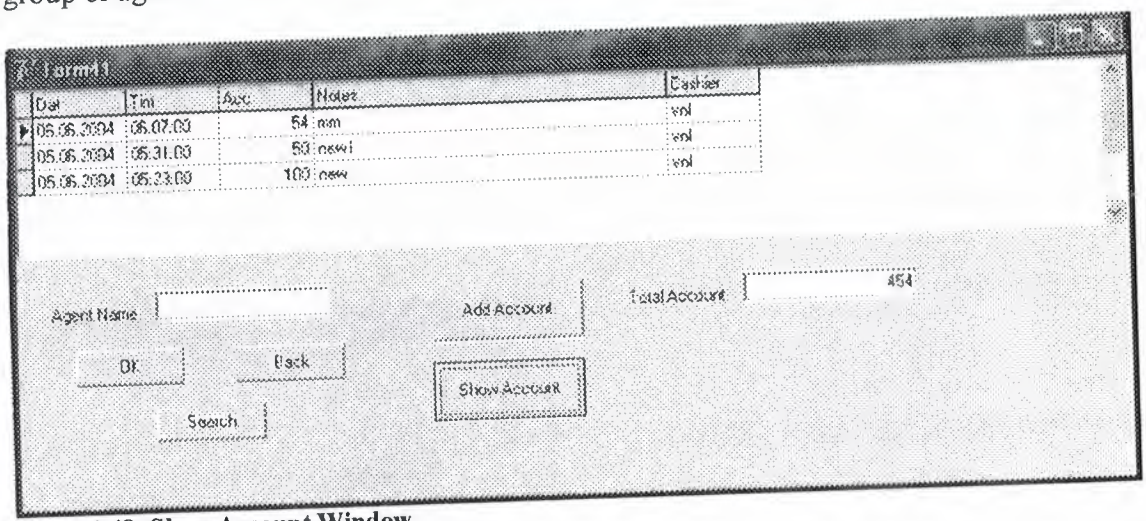


Figure1.48: Show Account Window

The user also can go back to the search window by using search button .

Also end of all these operations the program will saved this operations to the cashier's account .

Another sub menu of cashier menü is quick check-out(Figure1.49) . A quick check-out menu is used to make check-out operation of the customers whose account is balanced .

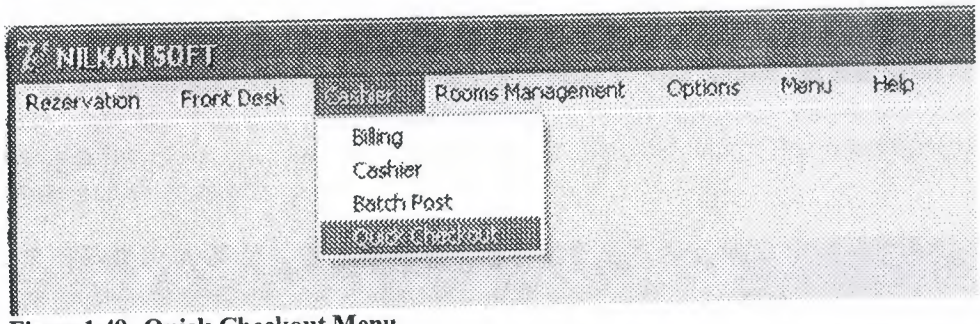


Figure1.49: Quick Checkout Menu

When the user choose quick check-out then the program will open a new window and will show two choices(Figure1.50) . This choices are "show all" , "only balance=0" .

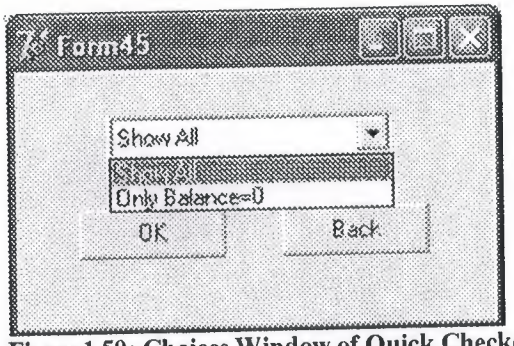


Figure1.50: Choices Window of Quick Checkout

After the user select one of them then the program will open a new window to show search criterias(Figure1.51) . These criterias can be "name" , "member number" and "room number" according to selected choices , the program will open a window to show the customers who are same with criterias (Figure1.52).

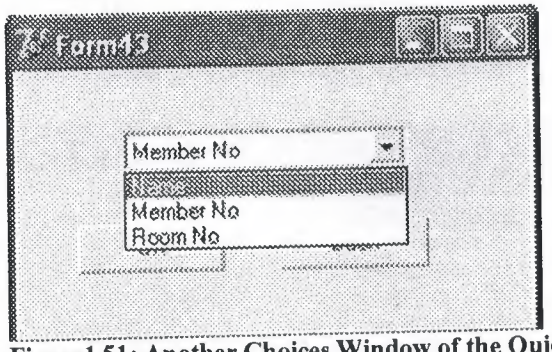


Figure1.51: Another Choices Window of the Quick Checkout

On this window there is a table to show the customer's information and some buttons to make operations . After this window will be opened then there will be a text reader with heading according to the previous choice . After the user enter this information into the this text reader then will click OK button . The program will search the information and if it find the customer then will show the customer's information on the table on this window . The user can make the check-out operation by using quick check-out button . Then the program will give a message to the user . İf the user accept this message then quick check-out operation will be made .

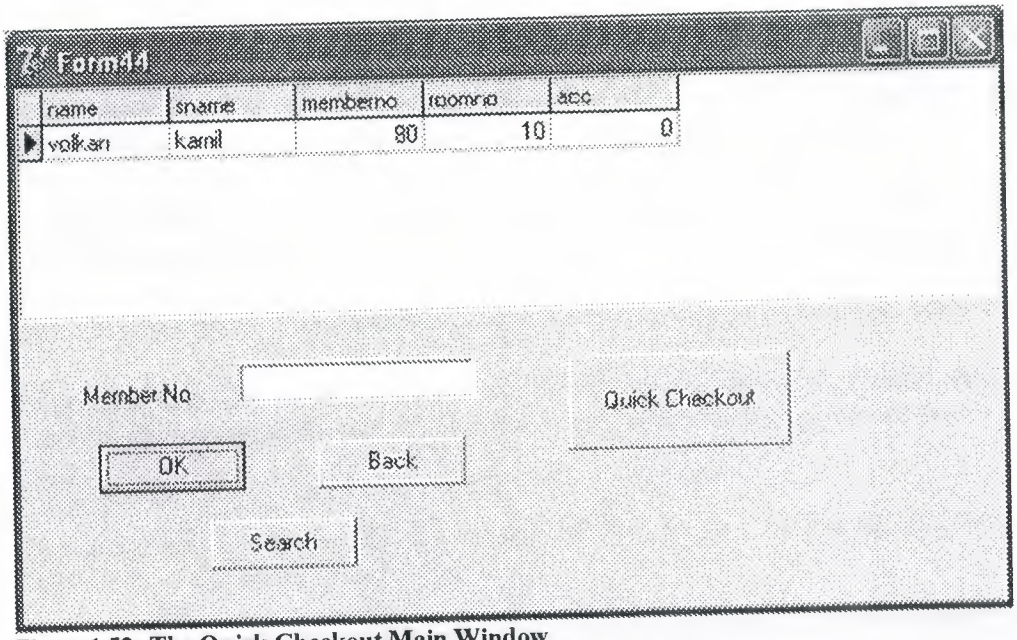


Figure1.52: The Quick Checkout Main Window

In an otel , one of the most important part is rooms management part . In this part the rooms of the otel can be controlled and managed . Almost all operations are made with room status . This status can be room number , number of available rooms , out of service rooms etc. Therefore , the rooms management is very important part of an otel .

In Nilkan Soft 1.0 , there is a main menu to control rooms named as rooms management(Figure1.53) . In this main menu the user can use house status , house keeping , out of order , overbooking , room history , graphicks , maintenance options .

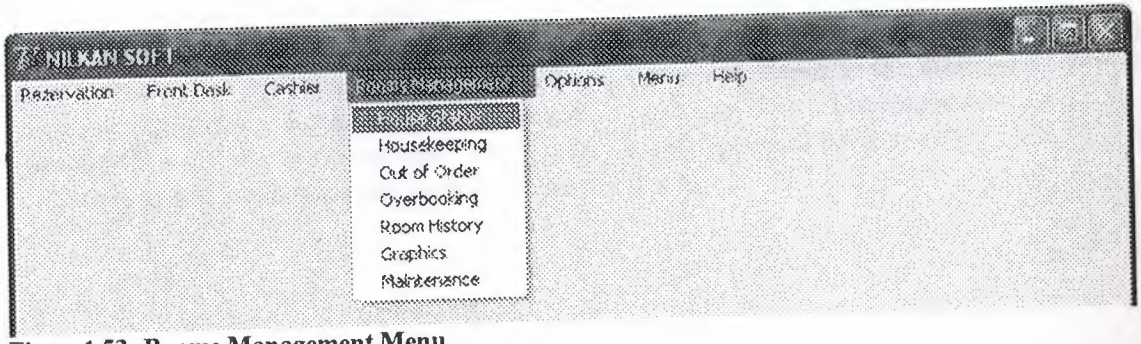


Figure1.53: Rooms Management Menu

In house status menu shows all the rooms status of the otel . When the user select this operations can see all information of the rooms like total rooms , available rooms out of order or out of service rooms , check-out rooms , checked in rooms , dirty and clean rooms etc .By using this window the user only can see the information that is take information about the rooms of the otel .

To control the rooms of the otel the Housekeeping part of the Rooms Management is used. In this part clean and dirty rooms can be pointed , out of order and out of service rooms can be controlled. The changes of the rooms' status on this window is affected to almost all parts of the program.Therefore the changes can be made very carefully.

Nilkan Soft allow to user to take the rooms Out of Order(OO) or Out of Service(OS) status. At all of these two option rooms are not seen at the available rooms .

In this program , the user can see the rooms status at an interval of dates . the user can use graphics option to use this operation after the user choice this part then the program wants to user to enter two dates . First date is starting date , second date is finishing dates . After the user enter these dates then the program will search rooms between these two dates .

In an otel , the detects of the rooms can be controlled carefully . Nilkan Soft 1.0 reserve a menu to control these detects . The maintenance menu under the rooms status main menu is used .

The last menu choice of the program is MENU menu . In this option there are general operations of the otel(Figure1.54) .

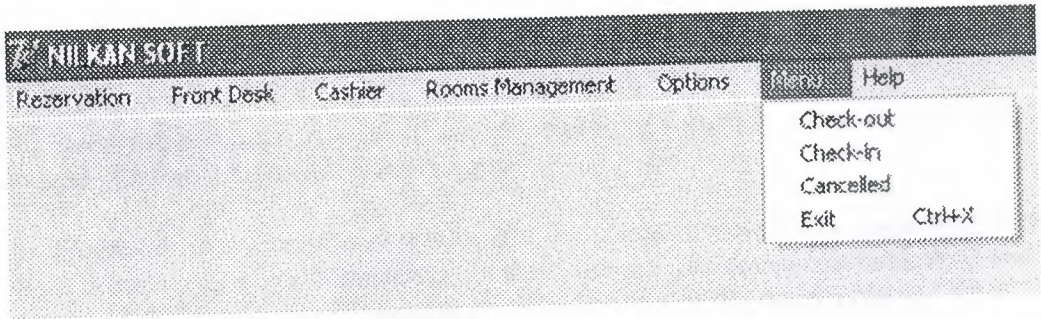


Figure1.54: Menu Options

In this menu there are four options . These options are check-in , check-out , cancelled , exit . The check-in operation shows to the user the reservation which are not checked in but has to checked in until that date . The check-out operation shows to the user the reservation which are not checked out but has to checked in until that date . The cancelled option show the user the cancelled reservations . And finally , Exit option is used to exit all the program . There is a shortcut key of this option which is ALT+x .

4.2 Nilkan Soft 1.0 Structure

The Nilkan Soft 1.0 is an Hotel Rezervation Program and made in Delphi 7.0 programming language.The Database tables are prepared in Paradox 7.0 . The tables are prepared very carefully.

In the first menu and most in use table is Main table. In the main table , the rezervations are saved. In this table , there are Memberno , Name , Sname , Roomno , Company , Source , VIP , Country , City , Arrival , Nights , Departure , Adult , Child , Roomtype , Restype , Time , Breakfast , Lunch , Dinner , Email , Url , Cotime , Corem , Notes and Checkin columns names.

Other rezervation table which is used for agent records , is Agent table. In the agent table , the agent information and records are saved. In this table , there are Agentno , Keyname , Name , Taxoff , Address , City , Postalcode , Telephone , Telefax , Email , Url , Acclerk , Commission , Remarks and Persons columns names.

Other rezervation table which is used for group records , is Grup table. In the Grup table , the group information and records are saved. In this table , there are Groupno , Gname , Company , Agent , Source , Arrival , Nights , Departure , Restype , Time , Breakfast , Lunch , Dinner , Notes , Cotime , Corem , Persons and Rooms columns names.

In Rezervation menu , the room assignments are made with roomass table. In this table , there are Roomno , Memberno , Name , Sname , Arrival , Departure , Stay , Agentno , Groupno and Rtype columns names.

In Rezervation menu , the events are made with events table. In this table , there are Date , Time and Events columns names.

In Rezervation menu , the waitlists are made with waitlist table. In this table , there are Name , Sname , Arrival columns names.

In Frontdesk menu , in the Arrival menu the check-in operations made with checkin table. In this table , there are Memberno , Name , Sname , Roomno columns. And end of the this operation the program will save the checkin column of the main table as "YES".

In In-House menu under the Frontdesk menu , the checked-in customers are shown using main table's checkin column name. In this column , if the text is "YES" the program will show that customer.

In Frontdesk menu , in the profile menu , the old customers are shown using history table. In this table , there are Name , Sname , Agent , Grup , Arrival , Dayuse , Departure , Roomno , Memberno.In this table the checked-out customers are saved.

In Accounts menu which is under the Frontdesk menu , the accounts operations are made by using acoount table. In this table , Name , Sname , Arr , Duntil and Account columns are used.

In Messages menu , the messages are saved to the Mesaj table. In this table , there are Nam , From , Dat , Tim , Notes columns names.

In Cashier menu , for the billing operations there are many tables are used. First at all , at the rezervation part of the program , the program creates a new table for each entry(customer) sepeartely. These tables' names starting with Billing and added customer's member number. For example ; for a customer with member number 45 , the billing table of this customer will be Billing45. In the Billing table , Dat , Tim , Acc , Notes and Cashier columns are used. For groups or agent there are also billing tables , for each group or agent seperately. Same with customers' billing tables , the groups' and agents' tables' names with starting Bilagent or Bilgroup and will continue group or agent number. In these tables the same columns are used.

In Cashier operations there are also many tables are used. For cashier information cashiers table is used. In this table the cashiers' details like as Cashier's name , sname , number , username , password are are saved. In this table , Pass , Uname , Name , Sname , Num , Acc and Coun columns are used. Another table used for Cashiers is used to save Cashier operations for Cashier Reports. This tables' names starting with Cash and continue with cashier username. In this table , Dat , Tim , Acc , Notes , Name , Sname , Memberno , Roomno , Agentname , Agentno , Grupname , Grupno columns are used.

Another important tables are for groups' or agents' rooms tables. These tables are named as for agents , starting with roomsagent and continue with the agent number. For groups starting with roomsgrup and continue with the group name.

CONCLUSION

In an otel , the Front Desk part is very important to control all the otel's functions. All the otel records are saved in the Front Desk department of an otel.Especially , in luxury and developed otels , there must be many functions to control the otel status. Like Rooms Status , Rezervations , Check-in , Check-out , Events in Otel , Reports and so on. To manage , record and control all these functions is very difficult in big otels. Therefore , the hotel rezervation programs are used.

An Otel Rezervation Program makes some general operations of an otel. And an otel rezervation program is the most important and most necessary part of an otel. This program provides to reach immediately to customer detailed information in otel , rezervations to customers and groups and agent who in otel and will come to otel , Rooms Status like full or empty rooms , clean or dirty rooms , The expences of the customers in otel and outside customers , cashier details like cashier reports , events in otel , services in otel and so on.

Nilkan Soft 1.0 is an otel rezervation program and written on Delphi Programming Language. The records are saved in tables which prepared in Paradox 7.0. Paradox 7.0 is a useful database manage program of Delphi . All the components of interfaces are Delphi's own components.