NEAR EAST UNIVERSITY

Faculty of Engineering

Department of Computer Engineering

Hotel Management system

Graduation Project
COM-400

Student: Polat Bayur (20001532)

Supervisor: Miss. Besime Erin

Nicosia - 2002

ACKNOWLEDGEMENTS

"First, I would like to thank my supervisor Miss Besime Erin for her valuable advices, encouragement and endless support.

Second, I would like to acknowledge special thank to the Near East Universityfor offering me a suitable environment during my study. And also I will neverforget the teacher's support and help.

Third, I would like to dedicate my research to my parents and the rest of thefamily who are always motivate me with all the love.

I gratefully acknowledge the role of myfriends those who set behind me while I'm preparing thisproject. "

ABSTRACT

The success of a database is completely dependent on the logical database design. Even ifwe buy expensive and fast hardware and software, the quality of the database design will dictate whether a project will succeed. hı a way, it is the Achilles heel of a project.

A good Database Design does the following:

1. Provides minimum search time when locating specific record.
2. Stores data in the most efficient manner possible to keep the database from growing too large.
3. Makes data updates as easy as possible.
4. Is flexible enough to allow inclusion of new functions required of the program.

The process of designing a database begins with an analysis of what information the database must hold and what are the relationships among components of that information. Often, the structure of the database, called the database schema, is specified in one of several languages or notations suitable for expressing designs. After due consideration, the design is committed to a form in which it can be input to a Database Management System (DBMS), and the database takes on physical existence.

Databases today are essential to every business. They are used to maintain internal records, to present data to customers and clients on the World-Wide-Web, and to support many other commercial processes. Databases are likewise found at the core of many scientific investigations. They represent the data gathered by astronomers, by investigators of the human genome, and by hie-chemists exploring the medicinal properties of proteins, along with many other scientists.

The power of databases comes from a body of knowledge and technology that has developed over several decades and is embodied in specialized software called a database management system, or DBMS, or more colloquially a database system." A DBMS is a powerful tool for creating and managing large amounts of data efficiently and allowing it to persist over long periods of time-safely. These systems are among the most complex types of software available.

TABLE OF CONTENTS

INTRODUCTION

A database system is a collection of information organized in such a way that a computer program can quickly select desired pieces of data. You can think of a database as an electronic filing system. Traditional databases are organized by fields, records, and files. A field is a single piece of information; a record is one complete set of fields; and a file is a collection of records.

Delphi Programming Language is a rich, efficient and very organized database system design. Borland Delphi is an object-oriented, visual programming environment for rapid development of 32-bit applications for deployment on Windows and Linux. Using Delphi, you can create highly efficient applications with a minimum of manual coding.

The aim of the project is to develop a Hotel Management System using Delphi programming Language. The project contains introduction, five chapters and conclusion.

Chapter One describes the Database Management Systems. The evaluation of the DBMS, the relational database system and the database design.

Chapter Two is voted to the Delphi Programming Language, describing how we can get the most efficiency of this programming language.

Chapter Three describing how is the system works. That means, when new customer comes to get a room on a hotel how the system deals with him/her.

Chapter Four is describing how is the Delphi code written. The procedures and their brief description of how they works are given as well.

Chapter Five describes the main elements and the general statements of the Structured Query Language (SQL).

Conclusion shows the advantages of using Delphi programming Language over other programming languages.

Chapter One
DATABASE MANAGEMENT SYSTEM

Databases today are essential to every business. They are used to maintain internal records, to present data to customers and clients on the World-Wide-Web, and to support many other commercial processes. Databases are likewise found at the core of many scientific investigations. They represent the data gathered by astronomers, by investigators of the human genome, and by bio-chemists exploring the medicinal properties ofproteins, along with many other scientists.

The power of databases comes from a body of knowledge and technology that has developed over several decades and is embodied in specialized software called a database management system, or DBMS, or more colloquially a database system." A DBMS is a powerful tool for creating and managing large amounts of data efficiently and allowing it to persist over long periods oftime-safely. These systems are among the most complex types of software available.

The capabilities that a DBMS provides the user are:

1. Persistent storage. A DBMS supports the storage of very large amounts of data that exists independently of any processes that are using the data. However, the DBMS goes far beyond the system in providing exibility, such as data structures that support efficient access to very large amounts of data.
2. Programming interface. A DBMS allows the user or an application program to access and modify data through a powerful query language.
3. Transaction management. A DBMS supports concurrent access to data, i.e., simultaneous access by many distinct processes. Databases today are essential to every business. They are used to maintain internal records, to present data to customers and clients on the World-Wide Web, and to support many other commercial processes. Databases are likewise found at the core of many scientific investigations. They represent the data gathered by astronomers, by investigators ofthe human genome, and by biochemists exploring the medicinal properties of proteins, along with many other scientists.

To avoid some of the undesirable consequences of simultaneous access, the DBMS supports isolation, the appearance that transactions execute one-at-a-time, and atomicity, the requirement that transactions execute either completely or not at all. A

DBMS also supports durability, the ability to recover from failures or errors of many types.

1.1. The Evolution of Database Systems

What is a database? In essence a database is nothing more than a collection of information that exists over a long period of time, often many years. In common parlance, the term database refers to a collection of data that is managed by a DBMS. The DBMS is expected to:

1. Allow users to create new databases and specify their schema (logical structure of the data), using a specialized language called a data-definition language.
2. Give users the ability to query the data (a query is database lingo for a question about the data) and modify the data, using an appropriate language, often called a query language or data-manipulation language.
3. Support the storage of very large amounts of data | many gigabytes or more | over a long period of time, keeping it secure from accident or unauthorized use and allowing efficient access to the data for queries and database modifications.
4. Control access to data from many users at once, without allowing the actions of one user to affect other users and without allowing simultaneous accesses to corrupt the data accidentally.

1.1.1 Early Database Management Systems

The first commercial database management systems appeared in the late 1960's. These systems evolved from file systems, which provide some of item (3) above; file systems store data over a long period of time, and they allow the storage of large amounts of data. However, file systems do not generally guarantee that data cannot be lost if it is not backed up, and they don't support efficient access to data items whose location in a particular file is not known. Further, file systems do not directly support item (2), a query language for the data in files. Their support for (1) a schema for the data is limited to the creation of directory structures for files. Finally, file systems do not satisfy (4). When they allow concurrent access to files by several users or processes, a file system generally will not prevent situations such as two users modifying the same file at about the same time, so the changes made by one user fail to appear in the file.

The first important applications ofDBMS's were ones where data was composed of many small items, and many queries or modifications were made. Here are some of these applications.

1.1.2 Relational Database Systems

Database systems should present the user with a view of data organized as tables called relations. Behind the scenes, there might be a complex data structure that allowed rapid response to a variety of queries. But, unlike the user of earlier database systems, the user of a relational system would not be concerned with the storage structure. Queries could be expressed in a very high-level language, which greatly increased the efficiency of database programmers.

Example 1.1: Relations are tables. Their columns are headed by attributes, which describe the entries in the column. For instance, a relation named Accounts, recording bank accounts, their balance, and type might look like:

accountNo	**balance**	**type**
12345	1000.00	savings
67890	2846.92	checking

Heading the columns are the three attributes: accountNo, balance, and type. Below the attributes are the rows, or tuples. Here we show two tuples of the relation explicitly, and the dots below them suggest that there would be many more tuples, one for each account at the bank. The first tuple says that account number 12345 has a balance of one thousand dollars, and it is a savings account. The second tuple says that account 67890 is a checking account with $2846.92.

Suppose we wanted to know the balance of account-67890. We could ask this query in SQL as follows:

```
SELECT balance
FROM Accounts
WHERE accountNo = 67890;
```

For another example, we could ask for the savings accounts with negative balances by:

```
SELECT accountNo
FROM Accounts
WHERE type= 'savings' AND balance< 0;
```

We do not expect that these two examples are enough to make the reader an expert SQL programmer, but they should convey the high-level nature of the SQL "select-from-where" statement. In principle, they ask the DBMS to

1. Examine all the tuples of the relation Accounts mentioned in the FROM clause,
2. Pick out those tuples that satisfy some criterion indicated in the WHERE clause, and
3. Produce as an answer certain attributes of those tuples, as indicated in the SELECT clause.

In practice, the system must "optimize" the query and second an efficient way to answer the query, even though the relations involved in the query may be very large.

1.1.3 Smaller and Smaller Systems

Originally, DBMS's were large, expensive software systems running on large computers. The size was necessary, because to store a gigabyte of data required a large computer system. Today, many gigabytes fit on a single disk, and it is quite feasible to run a DBMS on a personal computer. Thus, database systems based on the relational model have become available for even very small machines, and they are beginning to appear as a common tool for computer applications, much as spreadsheets and word processors did before them.

1.1.4 Bigger and Bigger Systems

On the other hand, a gigabyte isn't much data. Corporate databases often occupy hundreds of gigabytes. Further, as storage becomes cheaper people find new reasons to store greater amounts of data. For example, retail chains often store terabytes (a terabyte is 1000 gigabytes, or 1000,000,000,000 bytes) of information recording the history of every sale made over a long period of time.

Further, databases no longer focus on storing simple data items such as integers or short character strings. They can store images, audio, video, and many other kinds of data that take comparatively huge amounts of space. For instance, an hour of video consumes about a gigabyte. Databases storing images from satellites can involve petabytes (1000 terabytes, or 1000,000,000,000,000 bytes) of data.

Handling such large databases required several technological advances. For example, databases of modest size are today stored on arrays of disks, which are called secondary storage devices (compared to main memory, which is "primary" storage). One could even argue that what distinguishes database systems from other software is, more

than anything else, the fact that database systems routinely assume data is too big to _tin main memory and must be located primarily on disk at all times. The following two trends allow database systems to deal with larger amounts of data, faster.

1.2. Outline of Database-System Studies

Ideas related to database systems can be divided into three broad categories:

1. Design of databases. How does one develop a useful database? What kinds of information go into the database? How is the information structured? What assumptions are made about types or values of data items? How do data items connect?
2. Database programming. How does one express queries and other operations on the database? How does one use other capabilities of a DBMS, such as transactions or constraints, in an application? How is database programming combined with conventional programming?
3. Database system implementation. How does one build a DBMS, including such matters as query processing, transaction processing and organizing storage for efficient access?

1.3. Database Design

The success of a database is completely dependent on the logical database design. Even if we buy expensive and fast hardware and software, the quality of the database design will dictate whether a project will succeed. In a way, it is the Achilles heel of a project. A good Database Design does the following:

1. Provides minimum search time when locating specific record.
2. Stores data in the most efficient manner possible to keep the database from growing too large.
3. Makes data updates as easy as possible.
4. Is flexible enough to allow inclusion of new functions required of the program.

The database design process can be divided into six steps:

1. Requirement analysis.
2. Conceptual database design.
3. Logical database design.
4. Schema refinements.
5. Physical database design.

6. Security database design

In this chapter I will describe in brief the Logical Database Design because it is the step directly before the writing a code.

Logical database design uses several rules or concepts which are reasonably well understood and accepted. Disagreement arises in formulating a particular methodology, the place to start and the sequence of steps to follow in applying those rules, After a brief discussion of database design methodologies, this section presents several concepts, principles, or rules which are generally recognized and applied regardless of the particular methodology used.

1.3.1 Database Design Methodologies

A *database design methodology* specifies a sequence of steps to follow in developing a "good" database design-one that meets user needs for information and that satisfies peıformance constraints. Each step consists of the application of *a* set of techniques or rules that may be formalized to varying degrees and embodied in software tools. A methodology should he (I) usable in a wide variety of design situations and (2) reproducible in different designers. The second objective implies that the methodology be teachable, and that those trained in applying the methodology would arrive at the same end result. This is not evident in the present state of the art. Logical database design remains veıy much than art.

A database design methodology consisting of four steps;

1. *User Information Requirements-involving* the users in analyzing organizational needs, setting the scope of interest, investigating what people do (organizational tasks; usage patterns), and determining the data elements needed to peıform those tasks,
2. *Conceptual* Design-developing a high-level diagrammatic representation of a logical data structure; a structure which includes object domains, events, entities, attributes, and relationships: a structure which seeks to model the users' world.
3. *Implementation Design-refining* the conceptual design, checking for satisfaction of user needs and for consistency, and adjusting it to meet processing and peıformance constraints in a particular computer and DBMS environment.
4. *Physical* Design-developing record storage designs, clustering, and establishing access paths.

The techniques and rules in the steps of a methodology are applied iteratively

in the process of unfolding, growing, and refining a database design. For a starting point, some suggest applying the methodology to individual user application areas or local views. Different user views may contain related complementary parts or overlapping pans. Multiple local views are then consolidated into a global logical structure or conceptual schema. The process of consolidation seeks to resolve inconsistencies, and to integrate related pieces. Even within a local view, there may be redundant, overlapping, and inconsistent pads. The rules of a methodology are intended to assist the designer in asking the right questions and representing the data structure in a coherent and consistent way regardless of the scope of the design activity, and regardless of whether it begins with individual local views or a global perspective. The product of the design activity will grow as it unfolds over the area of interest; and it will be refined as the rules are applied to focus attention on particular aspects of an infinitely complex reality and to resolve ambiguities and inconsistencies in the developing database structure.

1.4. Entity Attribute Relation

Perhaps the most significant difference among methodologies or approaches to logical database design is found in the point at which data items are clustered or grouped into records. The top division of the taxonomy of data structures presented in Chapter 4 reflects this division. The number of basic constructs distinguishes the two approaches: Those which presume an early clustering are often called "Entity Attribute-Relation" or "E-A-R" approaches; the alternative is called the "Object Relation" or "0-R" approach.

Historically data processing has always worked with records. Programming languages such as COBOL and FORTRAN cluster data items into records. The formation of records as a contiguous set of data items is necessary for efficient data processing. A record is the unit of access for getting data in and out of programs. Data is moved to and from secondary storage in blocks of records. Earlier data processing systems forced a "unit record" view, that is, all data for an application had to reside in a single sequence of records (this reflected the technology of the day, which used what was called "unit record equipment"). Even today, with DBMSs supporting a multi file data structure, data exists in the form of records in most organizations. Users are very familiar and comfortable with a record-oriented view of their data. Most designers today use an E-A-R approach to logical database design.

The major problem with the E-A-R approach to logical database design is that it allows the relationships among data items within a record to be hidden. It does not force

the designer to explicitly consider and define inter record structures. This accounts for the recent emphasis in the literature on record decomposition and normalization based on an analysis of functional and multi valued dependencies. These techniques are all aimed at uncovering and making explicit the relationships among individual data items within records.

The end result of repeatedly applying record decomposition rules is irreducible varies-at which point there exists at *most one* non-identifier data item within each record. By then the designer will have considered all inter item relationships.

At the implementation or physical level, data items must be clustered into records for efficient data processing. Even at the logical level, it is still relevant and useful to think of attributes which cluster around and describe entities, whether the attribute items are considered as part of entity records or as individual object domains. It is relatively unimportant whether the design activity starts with records which are decomposed to analyze inter item relationships, or starts with object domains which are clustered to form records. In practice a designer will do both. It is important that certain rules and concepts be applied in the design process. Early formation of records is dangerous only if it inhibits the designer from properly analyzing intra record relationships among data items, and from considering alternative groupings ofitems into records -Ideally, the formation ofrecords should be part ofthe implementation phase ofdatabase design since it is done primarily for system convenience and processing efficiency. In fact, it is desirable to have software tools to perform the clustering, leaving the designer to concentrate on defining the individual data objects, relationships, and performance factors and constraints.

In a strict application ofthe object-relation approach to logical design, all object domains are treated equally. In the E-A-R approach, attention is initially focused on entities, then on the attributes of those entities, which may turn out to be other entities. In fact, the distinction between attributes and entities is often confusing and arbitrary Again, regardless of the approach taken, it is important for the designer to focus attention on the more important parts of the users' world being modeled in the data structure. This is automatically done in the E-A-R approach but can also be done in the 0-R approach. The designer needs a high level of abstraction when developing a data structure and may start out by representing the main entities as boxes labeled with a name only.

1.5. Rules To Guide Logical Database Design

Even though there is no widespread acceptance of any particular design methodology, there is general recognition of many underlying rules and concepts used in logical database design. They relate to conceptual design and part of implementation design.

A good database designer will generally know these rules and apply them, often intuitively, wherever they are relevant in the process of developing, checking, and refining a database design.

The following rules are presented here in a reasonably logical order, but there is no implication that they should be applied in any strict sequence. There is also no implication that these rules are sufficient or complete for the database design task. While progress is being made in formalizing the principles and process of database design, it still depends heavily on human intelligence and experience. Even experienced designers can arrive at different database designs, which purport to model the same user environment.

ENTITY: Clearly identify the entities to be represented in the database. An entity is any object (person, place, thing), event, or abstract concept within the scope of interest about which data is collected. An entity is the object of decisions and actions within an organization. Entities are the pivotal elements in a data structure and must be well defined. Staff out by focusing on the main entities, gradually expanding the logical data structure view to include related entities. When looking at an existing database, clearly define the primary entity, which is described in each file (record type).

INCLUSION: Specify the criteria for including (or excluding) entity instances from a defined class of entities. The ENTITY rule names a class of entities and the INCLUSION rule specifies the conditions for membership in that class. For example, does the EMPLOYEE entity class include managers, job applicants, rejected job applicants, those fired or laid off, those who quit, or employees on definite or indefinite leave'? Consideration of these other "EMPLOYEES" may suggest broadening the name of the entity class, or it may give rise to another entity class. Narrowing (sub setting) or broadening the definition of the entity class represents movement along the generalization hierarchy.

ATTRIBUTE: Identify the attributes of each entity. Initially focus on the major attributes of each entity. Some will be clear and obvious, some will seem to be artificial, and some may also relate to other entities. Include all attributes, which assist in understanding the nature of the entity being described. Include at least one attribute from each set of similar attributes.

ATTRIBUTE CHARACTERISTICS: Define the characteristics of each attribute. Clearly define the characteristics of each attribute. Initially focus on name, type, (size), existence, uniqueness, and some indication of the nature of the value set. When describing an existing database, specify any encoding of data item values. Description of other characteristics can be deferred until later in the database design process. Eventually plan to describe one attribute per page in the final database documentation.

DERIVED ATTRIBUTE: Identify and define derived attributes. The values of an attribute may be derived from the values of other attributes in the database. Specify the derivation rule, which may be an expression for a derived item or a statistical calculation across instances of an entity type or a repeating group.

IDENTIFIER: Designate the attribute(s), which uniquely identify entity instances in each entity class. An entity identifier may be a single attribute (EMPNO) or multiple attributes (UNIT and JOECODE for POSITION). There may be multiple identifiers for the same entity (EMPNQ and SOCIAL SECURITY NUMBER). Indicate if the identifier is not guaranteed to be unique. The identifier can be a good clue to understanding the nature of the entity described in an existing file.

RELATIONSHIP: identify the primary relationships between entities.

RELATIONSHIP CHARACTERISTICS: Define the characteristics of interentity relationships, particularly exclusivity and exhaustibility (or dependency). Exclusivity refers to whether instances of one entity type can be related to at most one or more than one instance of another entity type. Since it is defined in both directions there are four possibilities: 1:1. 1:Many, Many: I, Many: Many. Exhaust stability (also called dependency or personality) specifies whether or not an instance of one entity type must be related to an instance of another entity type. Indicate ifthere is some condition on the dependency of a relationship. Also indicate if there is some minimum or maximum

cardinality on the "many" side of a relationship. (See section 6.3.3 for more detail on these characteristics.)

FOREIGN IDENTIFIER: indicate the basis for each relationship by including, as an attribute in one entity type, the identifier from each related entity type.' Every relationship is based upon common domain(s) in the related entity records. At the logical level, it is necessary to include the identifier of a related entity as a foreign identifier. In the storage structure, if the common domain is not explicitly stored in a related record, then some form of physical pointer is necessary to represent the relationship.

DERIVED RELATIONSHIPS: Suppress derived relationships. The logical database design should not include relationships, which can be derived from other relationships. For example, it is reasonable to think of organizational units as possessing a pool of skills. Furthermore, such information can be retrieved from the database. However, such a relationship should not be defined since it is derived from the ORGANIZATION-EMPLOYEE relationship and the EMPLOYEE-SKILL relationship. An organizational unit only possesses skills because it has employees who possess skills.

REPEATING GROUP: isolate any multi-valued data item or repeating group of data items within a record. This rule ensures that a record only contains atomic (single-valued) data items, thus allowing only flat files. This is also called first normal form. The real importance of this rule is to force the designer to explicitly recognize a "something-to-many" relationship and possibly a new entity type. If a repeating group of data items becomes a new entity record type, the identifier of its parent record must propagate down into the new record. If the relationship was'actually many-to-many, the propagated identifier becomes part of the identifier of the new record; if the relationship was one-to-many, the propagated identifier becomes a foreign identifier in the new record (but not part of the identifier). Multi-valued data items or nested repeating groups of data items may be included in the storage structure of a record.

PARTIAL DEPENDENCY: Each attribute must be dependent upon the whole record (entity) identifier. An attribute that is dependent upon only part of the identifier should be removed from the record, and placed in a record where that part of the identifier is the whole identifier. Suppose we had a record with the following data items: EMPNO,

SKILLCODE, SKILL DESCRIPTION, and PROFICIENCY. The identifier would have to be the first two data items jointly since PROFICIENCY relates to both of them together. However, DESCRIPTION relates only to the SKILLCODE and, therefore, should not be in this record. A record with no partial dependencies is said to be in second normal form.

TRANSITIVE DEPENDENCY: Each attribute within a *record* must *be directly* dependent upon the entity identifier. Any attribute, which is not directly dependent upon the record identifier, should be removed from the record, and related directly to the object on which it is functionally dependent. For example, if the EMPLOYEE record contained UNIT and BOSS, and the employee was moved to another organizational unit, it would not be sufficient to update the employee's UNIT the BOSS data item would also have to be changed. The update anomaly results because BOSS is directly dependent upon UNIT and not EMPNO. BOSS does not belong in the EMPLOYEE record *even if processing is faster and easier;* it belongs in the ORGANIZATIONAL UNIT record. A record with no partial or transitive dependencies is said to be in third normal form. Restated: An attribute should be dependent upon the identifier, the whole identifier, and nothing but the identifier.

Application of the previous three rules to arrive at third normal form requires an examination of every attribute in a record. A record not in third normal form produces undesirable update anomalies. To identify these anomalies, the designer can ask: If a given attribute is updated, what other attributes must change, or if another attribute is updated, what effect will it have on the given attribute?

NAMING: Assign names to entities, attributes, and relationships using a consistent, well-defined naming convention. When describing an existing database, watch for naming inconsistencies= different names for the same object, or the same name used to refer to different objects.

STORAGE & ACCESS: Suppress any consideration of physical storage structures and access mechanisms in describing the logical structure of the data. This includes any stored ordering on the records in a file, and whether or not a data item is indexed. Do not be concerned with questions of how to find or access a particular record in a file,

perhaps along a relationship Remember, all relationships are inherently bi-directional.

Chapter 2
DELPHI PROGRAMMING LANGUAGE

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

Delphi provides a comprehensive class library called the *Visual Component Library* (VCL), Borland Component Library for Cross Platform (CLX), and a suite of Rapid Application Development (RAD) design tools, including application and form templates, and programming wizards. Delphi supports truly object-oriented programmıng.

This chapter briefly describes the Delphi development environment and how it fits into the development life cycle.

2.1. Creating projects

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

You can view the contents of a project in a project management tool called the Project Manager. The Project Manager lists, in a hierarchical view, the unit names, the forms contained in the unit (if there is one), and shows the paths to the files in the project.

Although you can edit many of these files directly, it is often easier and more reliable to use the visual tools in Delphi.

At the top of the project hierarchy, is a group file. You can combine multiple projects into a project group. This allows you to open more than one project at a time in the Project Manager.

Project groups let you organize and work on related projects, such as applications that function together or parts of a multi-tiered application. If you are only working on one project, you do not need a project group file to create an application.

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

file is updated. You specify project options using a Project Options dialog which has tabs for various aspects of your project such as forms, application, compiler. These project options are stored in the project file with the project.

Units and forms are the basic building blocks of a Delphi application. A project can share any existing form and unit file including those that reside outside the project directory tree. This includes custom procedures and functions that have been written as standalone routines.

If you add a shared file to a project, realize that the file is not copied into the current project directory; it remains in its current location. Adding the shared file to the current project registers the file name and path in the **uses** clause of the project file. Delphi automatically handles this as you add units to a project.

When you compile a project, it does not matter where the files that make up the project reside.

2.2. Code Editor

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

The contents of the form, all of its properties, its components, and their properties can be viewed and edited as text in the Code editor. You can adjust the generated code in the Code editor and add more components within the editor by typing code.

As you type code into the editor, the compiler is constantly scanning for changed and updating the form with the new layout. You can then go back to the form, view and test the changes you made in the editor and continue adjusting the form from there.

The Delphi code generation and property streaming systems are completely open inspection. The source code for everything that is included in your final executable file-all of the VCL objects, CLX objects, RTL sources, all of the Delphi project files can be viewed and edited in the Code editor.

2.3. Understanding Datasets

The fundamental unit for accessing data is the dataset family of objects. Your application uses datasets for all database access. A dataset object represents a set of records from a database organized into a logical table. These records may be the records from a single database table, or they may represent the results of executing a query or stored procedure.

All dataset objects that you use in your database applications descend from *TDataSet,* and they inherit data fields, properties, events, and methods from this class. This chapter describes the functionality of *TDataSet* that is inherited by the dataset objects you use in your database applications. You need to understand this shared functionality to use any dataset object.

TDataSet is a virtualized dataset, meaning that many of its properties and methods are *virtual* or *abstract.* A *virtual method* is a function or procedure declaration where the implementation of that method can be (and usually is) overridden in descendant objects. An *abstract method* is a function or procedure declaration without an actual implementation. The declaration is a prototype that describes the method (and its parameters and return type, if any) that must be implemented in all descendant dataset objects, but that might be implemented differently by each of them.

Because *TlıauiSet* contains abstract methods, you cannot use it directly in an application without generating a runtime error. Instead, you either create instances of the built-in *TDataSet* descendants and use them in your application, or you derive your own dataset object from *TDataSet* or its descendants and write implementations for all its abstract methods.

TDataSet defines much that is common to all dataset objects. For example, *TdataSet* defines the basic structure of all datasets: an array of *TField* components that correspond to actual columns in one or more database tables, lookup fields provided by your application, or calculated fields provided by your application. For information about *'I'Field*components, see Chapter 19, "Working with field components."

2.3.1 Opening And Closing Datasets

To read or write data in a dataset, an application must first open it. You can open a dataset in two ways,

• Set the *Active* property of the dataset to *True,* either at design time in the Object Inspector, or in code at runtime:

CustTable.Active := True;

• Call the *Open* method for the dataset at runtime,

CustQuery.Open;

When you open the dataset, the dataset first receives a *BeforeOpen* event, then it opens a cursor, populating itself with data, and finally, it receives an *AfterOpen* event.

The newly-opened dataset is in browse mode, which means your application can read the data and navigate through it. You can close a dataset in two ways,

• Set the *Active* property of the dataset to *False,* either at design time in the Object Inspector, or in code at runtime,

CustQuery.Active := False;

• Call the *Close* method for the dataset at runtime,

CustTable.Close;

Just as the dataset receives *BeforeOpen* and *AfterOpen* events when you open it, it receives a *BeforeClose* and *AfterClose* event when you close it. handlers that respond to the *Close* method for a dataset. You can use these events, for example, to prompt the user to post pending changes or cancel them before closing the dataset. The following code illustrates such a handler:

```
procedure TForml.CustTableVerifyBeforeClose(DataSet: TDataSet);
begin
if (CustTable.State in [dsEdit, dslnsertj) then begin
case MessageDlg('Post changes before closing?', mtConfirmation, mbYesNoCancel, 0)
of
mrYes: CustTable.Post; {save the changes}
mrNo: CustTable.Cancel; {abandon the changes}
mrCancel: Abort; {abort closing the dataset}
end;
end;
end;
```

2.4. Developing The Application User Interface

With Delphi, you design a user interface (UI) by selecting components from the component palette and dropping them onto forms, You get the components to do what you want by setting their properties and coding their event handlers.

2.4.1 Controlling application behavior

TApplication, TScreen, and *TForm* are the classes that form the backbone of all Delphi applications by controlling the behavior of your project. The *TApplicaïion* class forms the foundation of an application by providing properties and methods that encapsulate the behavior of a standard program. *TScreen* is used at runtime to keep track offonns and data modules that have been loaded as well as maintaining system-specific infonnation such as screen resolution and available display fonts.

Instances of the *TForm* class are the building blocks of your application's user interface. The windows and dialog boxes in your application are based on *TForm.*

2.4.2 Using The Main Form

TForm is the key class for creating GUI applications. When you open Delphi displaying a default project or when you create a new project, a form is displayed on which you can begin your UI design.

The first form you create and save in a project becomes, by default, the project's main form, which is the first form created at runtime. As you add forms to your projects, you might decide to designate a different form as your application's main form. Also, specifying a form as the main form is an easy way to test it at runtime, because unless you change the form creation order, the main form is the first form displayed in the running application.

To change the project main form,

1. Choose ProjectjOptions and select the Forms page.
2. In the Main Form combo box, select the form you want to use as the project's main form and choose OK.

Now if you run the application, the form you selected as the main form is displayed.

Adding forms

To add a form to your project, select FilejNew Form. You can see all your project's forms and their associated units listed in the Project Manager (ViewjProject Manager) and you can display a list ofthe forms alone by choosing ViewJForms.

Linking forms

Adding a form to a project adds a reference to it in the project file, but not to any other units in the project. Before you can write code that references the new form, you need to add a reference to it in the referencing forms' unit files. This is called *form linking.* A common reason to link forms is to provide access to the components in that form. For example, you'll often use form linking to enable a form that contains data-aware components to connect to the data-access components in a data module. To link a form to another form,

1. Select the form that needs to refer to another.
2. Choose File!Use Unit.
3. Select the name ofthe form unit for the form to be referenced.
4. Choose OK.

Linking a form to another just means that the uses clauses of one form unit contains a reference to the other's form unit, meaning that the linked form and its components are now in scope for the linking form.

Avoiding circular unit references

When two forms must reference each other, it's possible to cause a "Circular reference" error when you compile your program. To avoid such an error, do one ofthe following:

• Place both **uses** clauses, with the unit identifiers, in the **implementation** parts of the respective unit files. (This is what the FilejUse Unit command does.)

• Place one **uses** clause in an **interface** part and the other in an **implementation** part. (You rarely need to place another form's unit identifier in this unit's **interface** part.), Do not place both **uses** clauses in the **interface** parts oftheir respective unit files. This will generate the "Circular reference" error at compile time.

You can prevent the main form from displaying when your application first starts up. To do so, you must use the global *Application* variable (described in the next topic). To hide the main form at startup,

1. Choose ProjectJView Source to display the main project file.

2. Add the following lines after the call to Application.CreateForm and before the call to Application.Run.

Chapter Three
HOW IS THE SYSTEM WORKS

In this chapter I will describe how the system will deal with the customers who come to the hotel to have a room or to visit their relatives or friends. The system will also describe the different processes toward those customers from the reservation till the payment of their bills.

3.1. MainMenu

The main menu of the 'Near East Hotel' system software is consisting of six submenus beside the 'Close' button as follow:

1. Reception.
2. Reservation.
3. Registration.
4. Rooms.
5. Employees.
6. Accounts.

The main menu and its main submenus are shown in figure 3.1. At the bottom of the main menu there is an animated text showing some information about the program writer.

Figure 3.1. The main menu.

3 .2. Reception Menu

When new customer or visitor comes to the hotel, he/she will meet firstly the receptors who will help him/her and find all the solutions to their queries. The receptors will do that using the data available in the reception menu. Reception menu consists of four submenus and the 'Close' button that will return you to the main menu (see figure 3.2). the submenus of the reception menu are as follow:

1. Phone List.
2. Search.
3. Confirmation.
4. Lists.

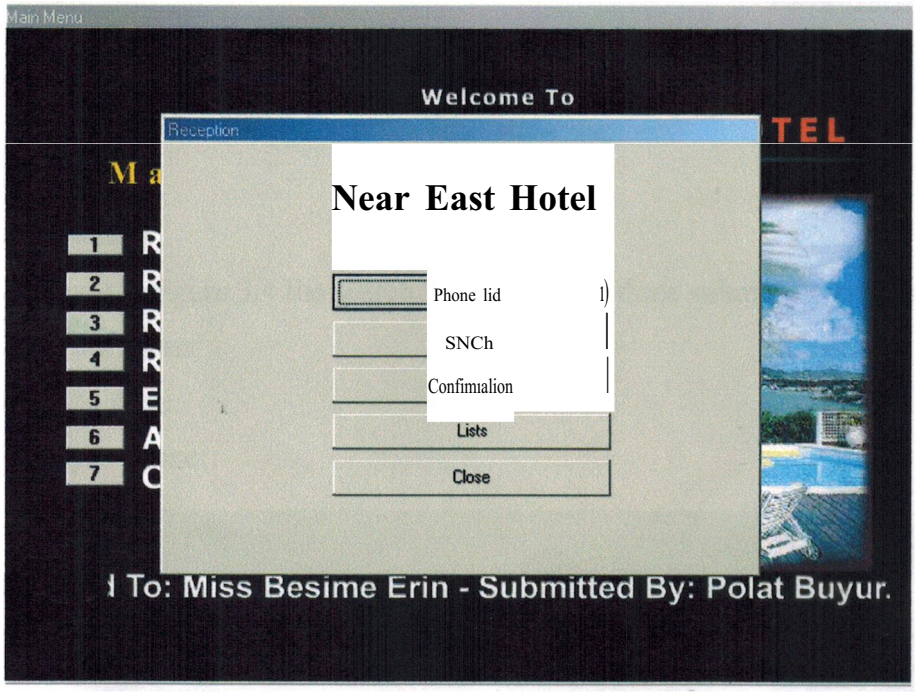

Figure 3.2. The reception menu. ,

3.2.1 Phone List

The receptor will use this window, which is contains the telephone numbers of all the staffs working on the hotel and all the different departments telephone numbers are here as well. As you can see in figure 3.3, the receptor cannot add more numbers but he can change numbers as needed.

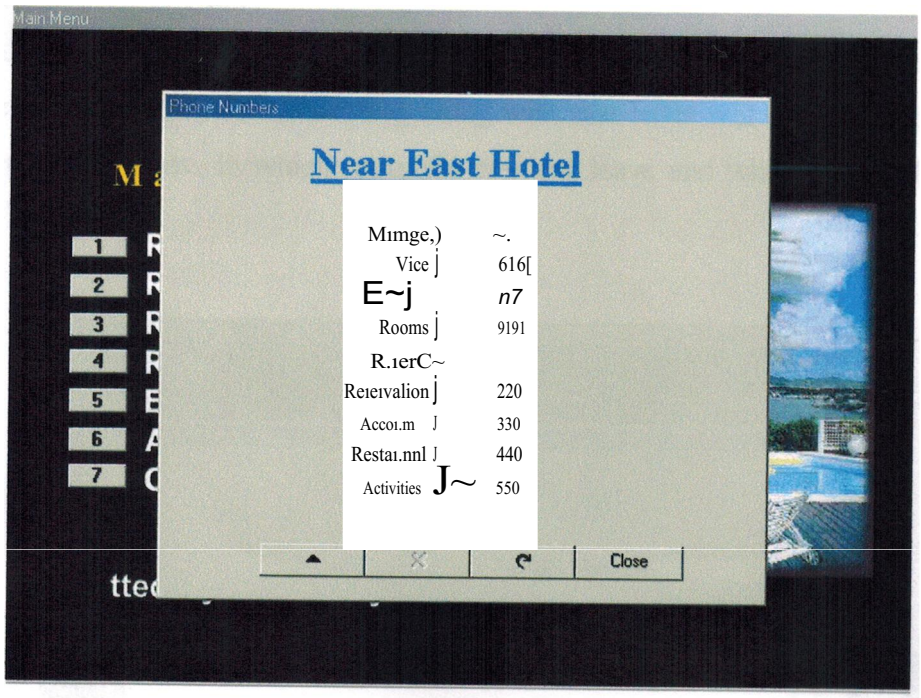

Figure 3.3. The hotel phone numbers.

3.2.2 Search Menu

As shown in figure 3.4 the search menu contains three submenus as follow:

1. Customer Name.
2. Room Number.
3. Employee Name.

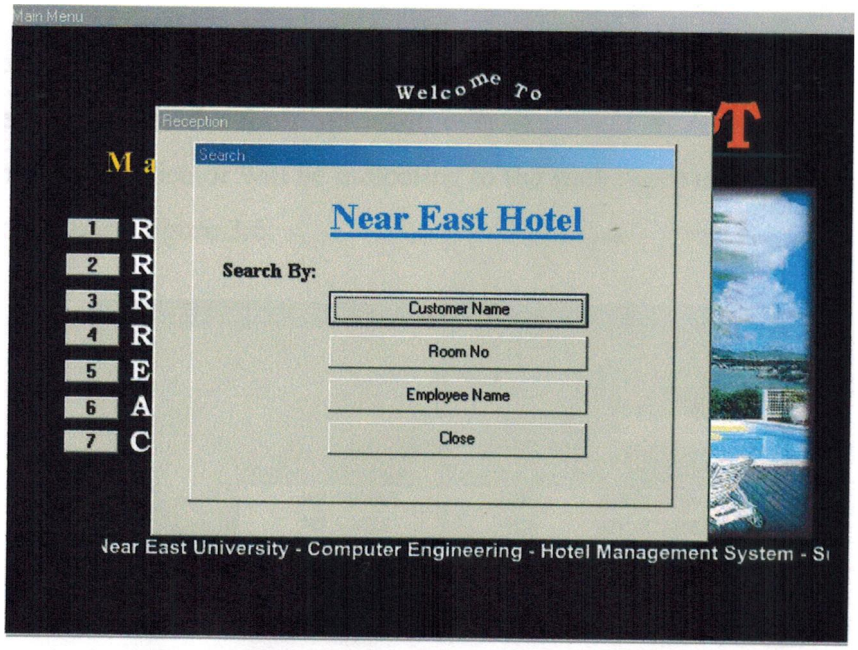

Figure 3.4. The search menu.

Search By Customer Name

As shown in figure 3.5 the search engine here is very sensitive, as well as you write the first letter only the names beginning with that letter will be shown in the list. The receptor will know in which room the resident leave and when he will leave the hotel.

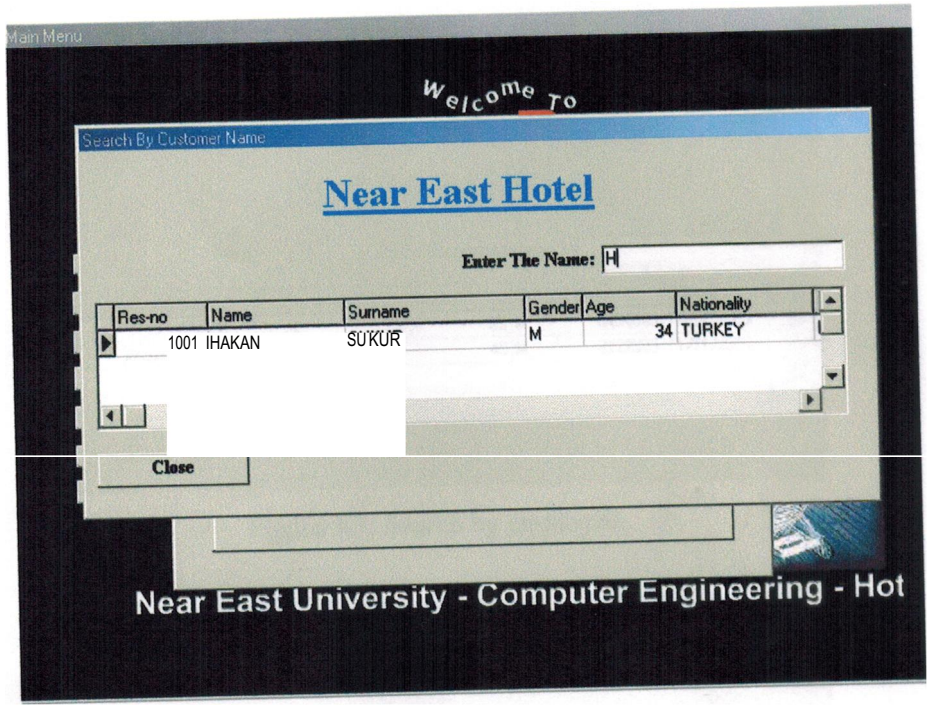

Figure 3.5. Search by customer name.

Search By Room No

It is useful to have the room no when ever you want it. As soon as you write down the room no the indicator will be indicating to the such room no. The search by the room no is shown in the figure 3.6.

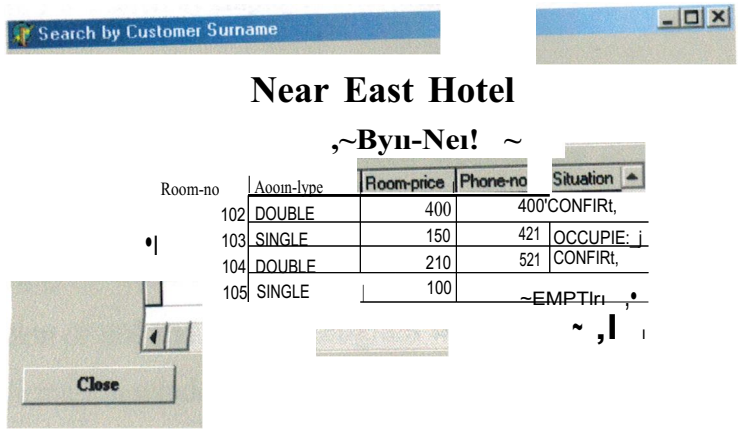

Figure 3.6. Search by room no.

Search By Employee Name

In the case any of the employees has a visitor or if we need some information about one of our employees we will use this search engine to find about him. The employee name will change during writing the letters, so we can find all the employees starting with any letter. The employee search engine is show in figure 3.7. And figure 3.8 showing the same thing after writing one letter more.

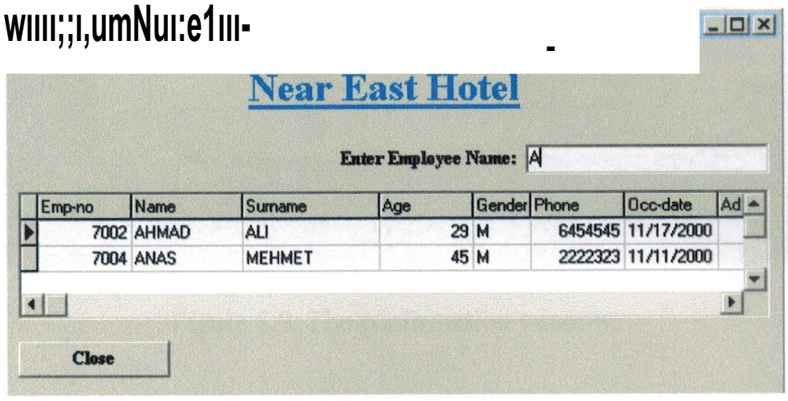

Figure 3.7. Search by employee.

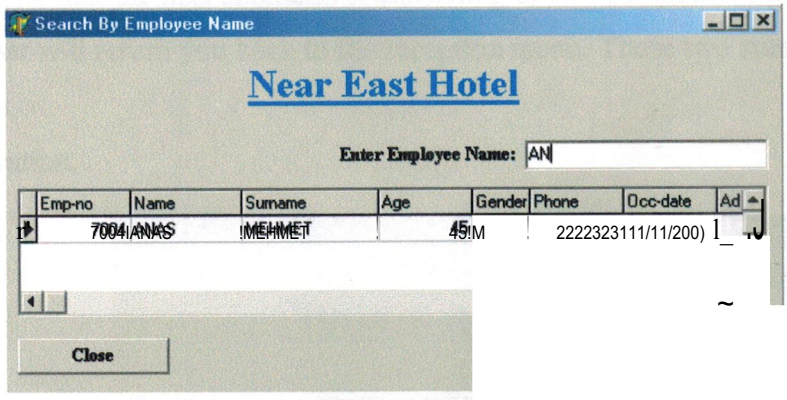

Figure 3.8. Search by employee after writing one more letter.

3.2.3 The Confirmation Submenu

In this window the receptor can dedicated weather the coming person has a confirmation or not. Unconfirmed customers are not allowed to have a room in the hotel. The receptor will simply write down the reservation no then the program will tell if this person has confirmation or not and also the program will show the name and surname of that person. The confirmation window is shown in the figure 3.9.

Figure 3.9. The confirmation window.

3.2.4 The Lists Submenu

The lists submenu is containing some queries that may come into consideration. Figure 3.10 show that the lists submenu is consisting of two submenu beside the "close" push button that will return you back to the reception menu. These two submenus is:

1. Rooms.
2. Reservation.

Figure 3.10. The lists submenu.

Rooms Lists

Inside this submenu the receptor will find some very useful queries about the rooms that help customers choose the suitable room and also help controlling the rooms. The rooms submenu contains the following lists:

1. All Rooms List: when this button pressed all the rooms in the hotel will be displayed (see figure 3.11).

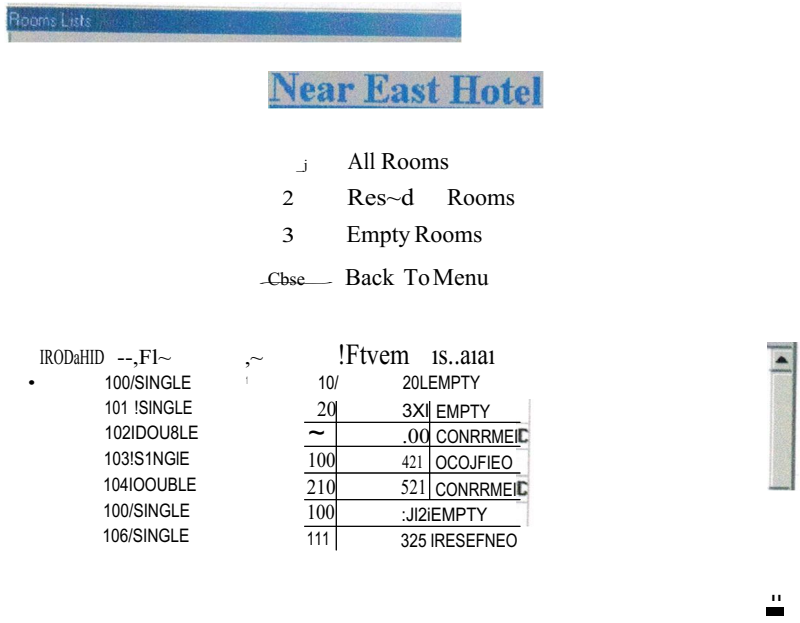

Figure 3.11. All the room list.

2. Reserved Rooms List: when this button pressed the reserved rooms only will appear at the list. This will help the hotel manager to manage his rooms (see figure 3. 12).

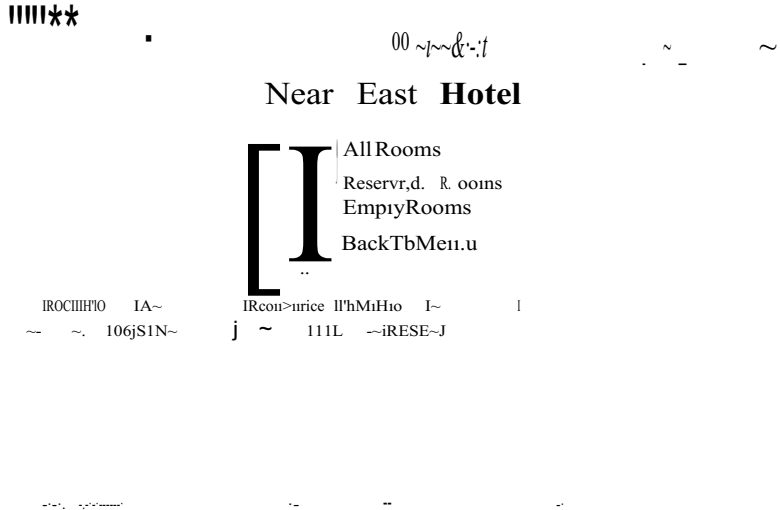

Figure 3.12. The reserved rooms list.

3. Empty Rooms List: when this button pressed only empty rooms will be listed, this will help new customers to know about the empty rooms (see figure 3.13).

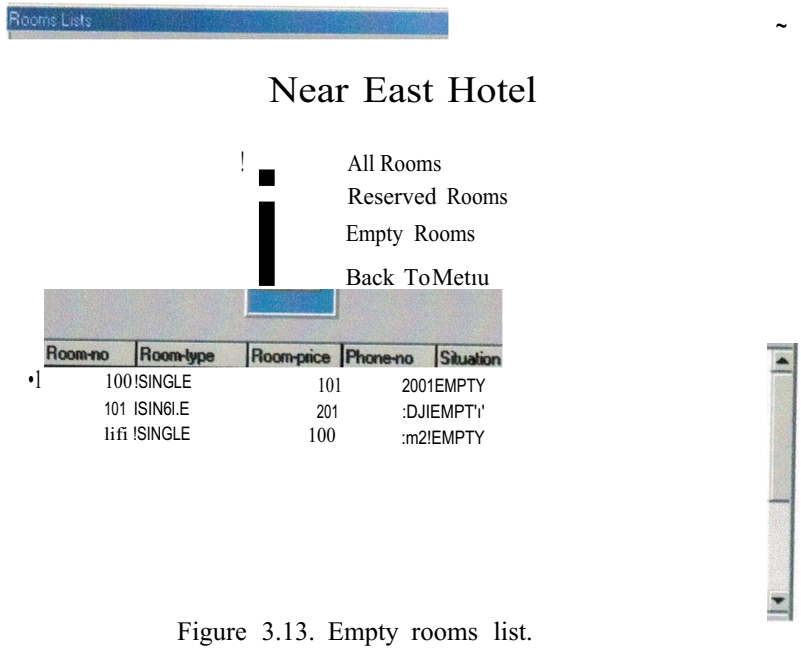

Figure 3.13. Empty rooms list.

Reservation Lists

Inside this submenu the receptor will find some very useful queries about the reserved customers and their rooms that they reserved. The reservation submenu contains the following lists:

I. Reserved Customers: when pressing this button the list of all reserved customers will displayed (see figure 3.14).

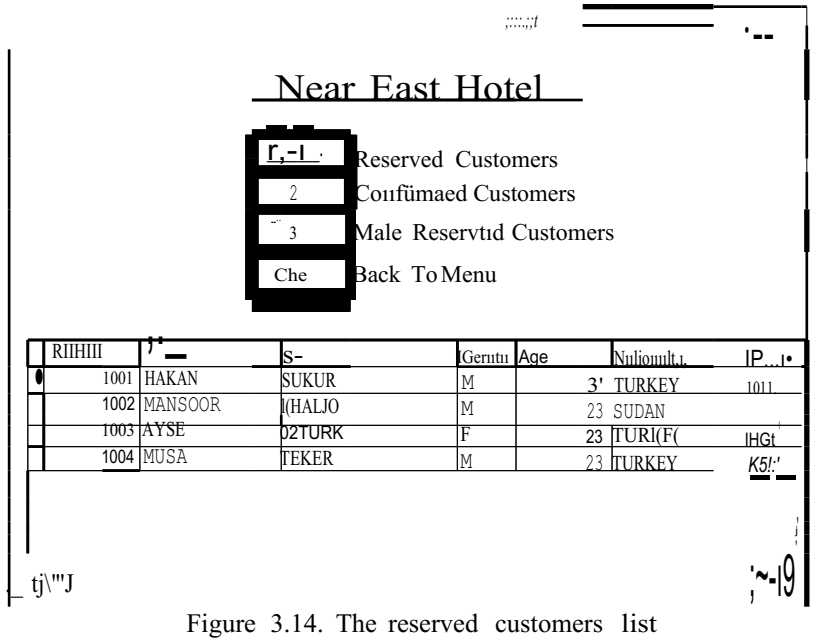

Figure 3.14. The reserved customers list

2. Confirmed Customers: when this button pressed only the customers who confirmed their reservation will appear (see figure 3.15).

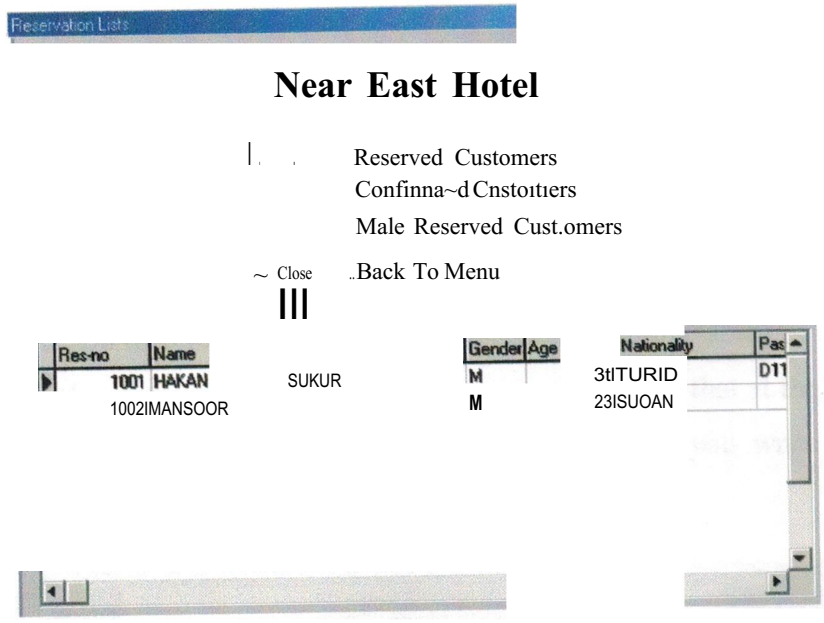

Figure 3.15. Confirmed customers list.

3. Male Reserved Customers: When this buttons pressed only male customers who reserved in the hotel will appear (see figure 3.16).

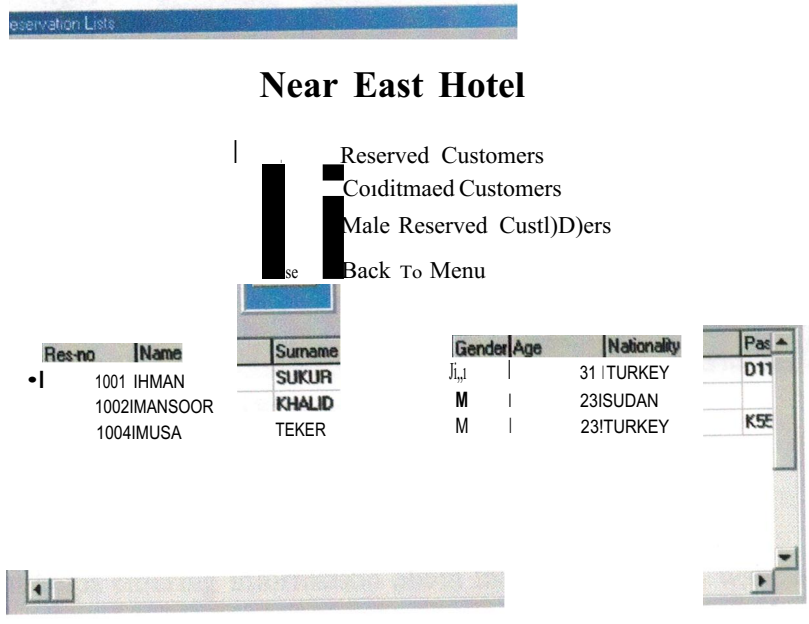

Figure 3.16. Male reserved list.

3.3. Reservation Form

When new customer wants to get a room at "Near East Hotel" he/she first has to

reserved and even though it is not enough for him/her to get a room, it is necessary also to have a confirmation as well.

The customer will get a reservation id and all his information will be recorded to the reservation table. This information will help the employees to deal with this customer, for example the confirmation window takes its information from this table, and also the reservation different lists takes its information from this table.

The reservation entrance form is shown *in* figure 3. 17. Note that at the bottom of the form the room information comes automatically when ever you write the room number.

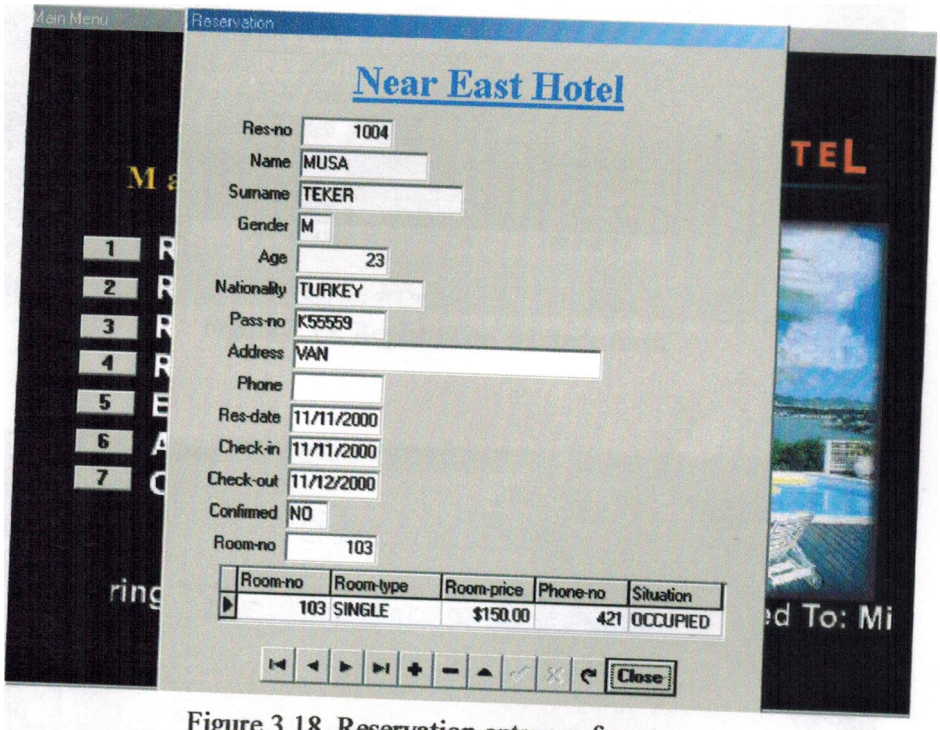

Figure 3.18. Reservation entrance form.

3.4. Registration Form

This form helps entering information about the customer who want to get *in* to the hotel. After ensuring this customer has a confirmed reservation then the customer will get a registration id. After entering his registration id and reservation id the rest of his details will come from the reservation table the only new entrance will be the today date and the date of leaving and some times the room no.

Note that when the room number is entered the room details will appear. The registration form shown figure 3.19. Note that there is a confirmation button that let us ensure if the customer has confirmation or no (See figure 3.20).

It is also important to say that all the information entered through this form will be stored in a separate table called "register.db".

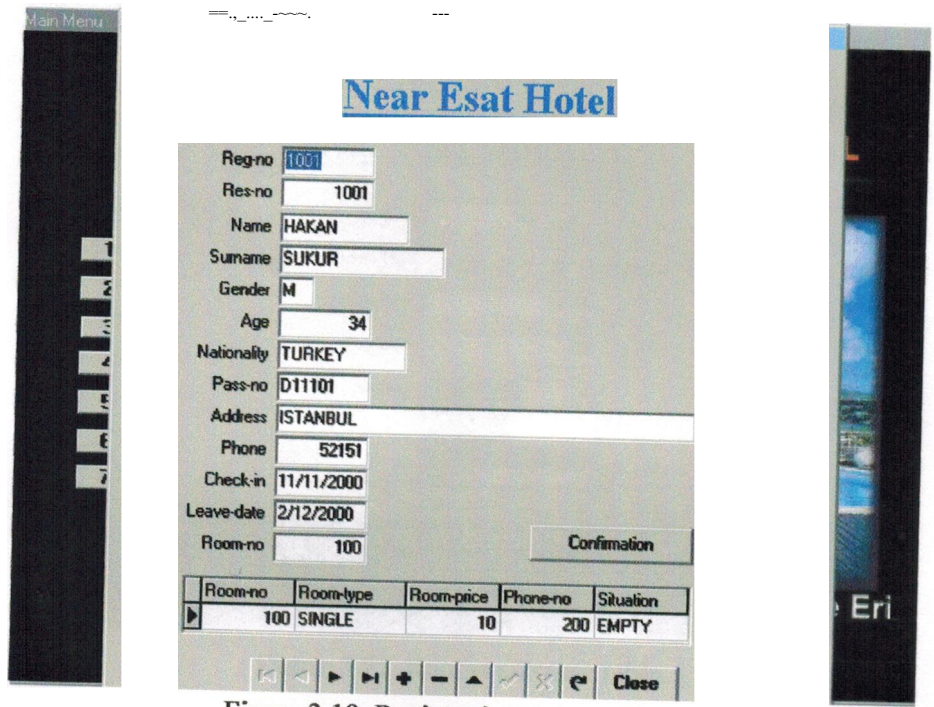

Figure 3.19. Registration entrance form.

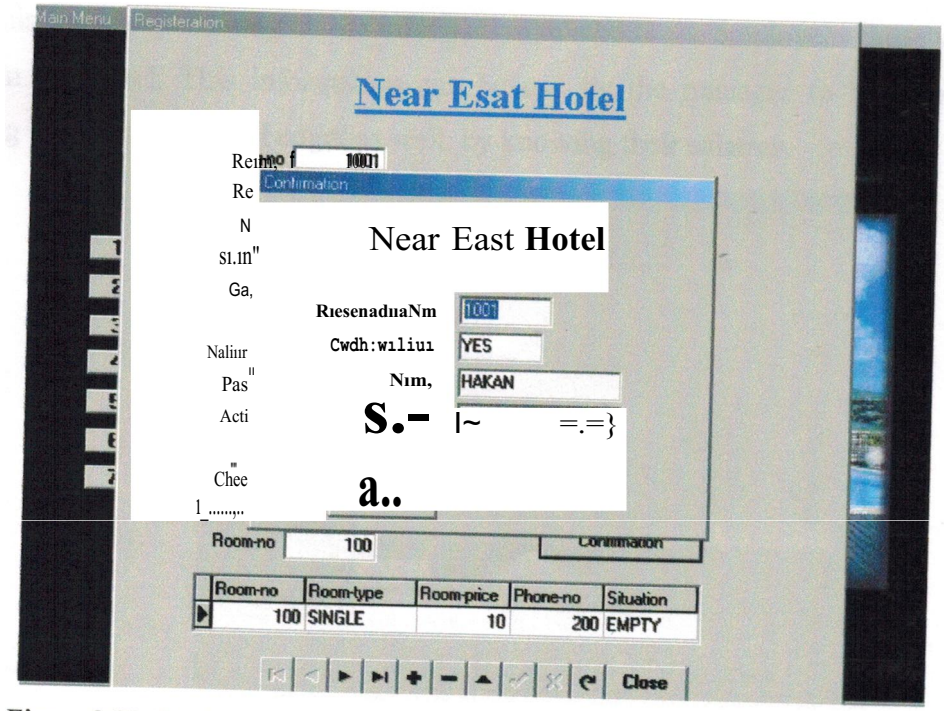

Figure 3.20. Registration entrance form after pressing the confirmation button.

3.5. The Rooms Form

All the information related to any of the different rooms in the hotel is entered through this form that shown in figure 3.21. The information will stored in a separate table called "rooms".

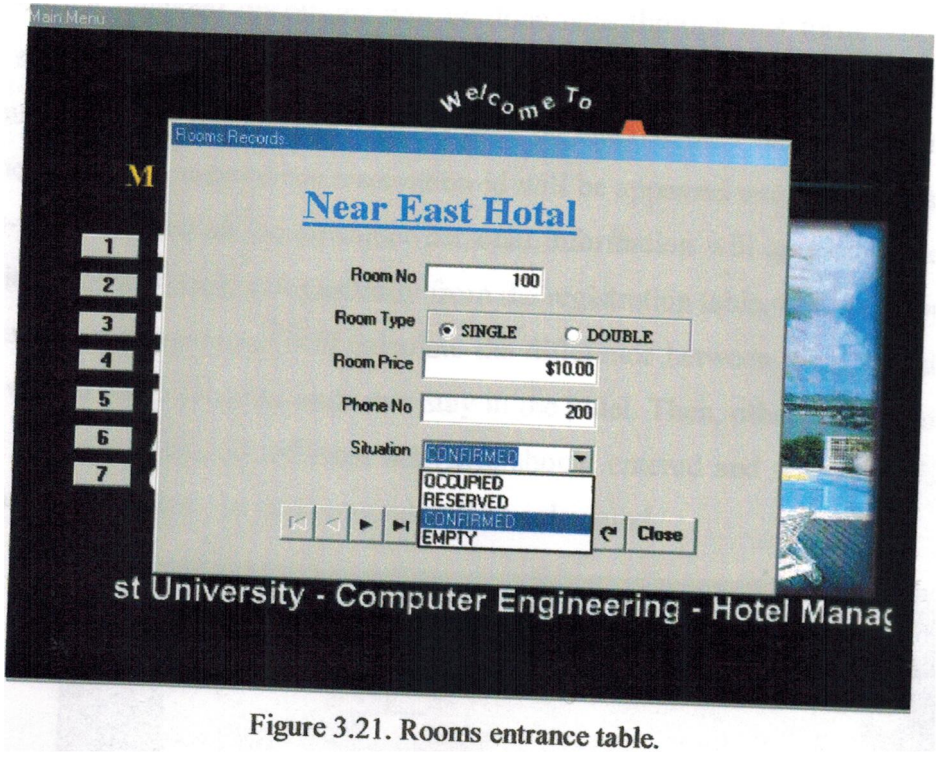

Figure 3.21. Rooms entrance table.

3.6. The Employees form

This form help entering all the information related to the employees those who work in the hotel. This information can be use by the manager to best control his working processes and his budget as well, by knowing their salaries.

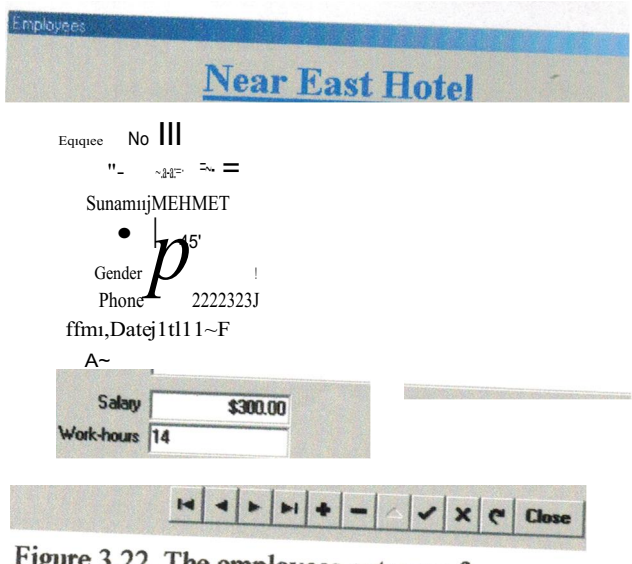

Figure 3.22. The employees entrance form.

All this information will stored in a separate table to facilitate get any data easily. The employees entrance form sown in figure 3.22.

3.7. The Accounts Form

All the financial operations will be done here through this form. The accounts form is show in figure 3.23. Any customers who get a room at a hotel should get an a count number. First the account number will be entered then his/her registration id, when the "reg-id" is entered the reservation id will be appeared automatically and when the "res-id" is entered all the customer personal information will appear. The customer "check-in" date will come automatically from the registration table, then we should enter his leaving date the program will calculate the difference between these two dates and will show who many days the customer stay in the hotel. Then, other money that he/she paid for the restaurant and different activities should entered and when the "Calculate Total" button is pressed the total amount will be calculated.

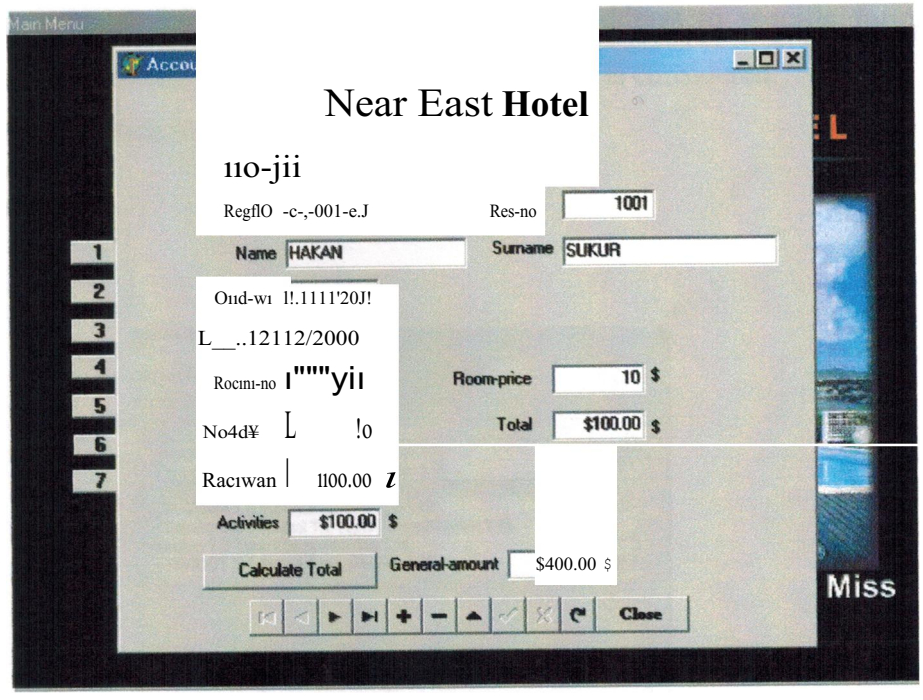

Figure 3.23. The accounts form.

Chapter Four
THE SOFTWARE CODE

The previous chapter has been describing how is the system works, that means how the system deal with customers, employees and even visitors. This chapter will describe in brief how the code written and I will concentrate on the new ideas during writing the software code, the forms generated automatically by the Delphi Compiler will not taken in consideration.

The program has been written using Delphi Programming Language, which is a user friendly computer language and its code is written in Pascal Programming Language. In the next sections I will describe each part separately starting from the main menu.

4.1. Main Menu

The main menu of the 'Near East Hotel' system software is a collection of Seven "Push Buttons", each of these buttons hold an address to specific form. For example to run a registration form the following procedure is considered:

```
procedure TMain.Button3Click(Sender: TObject);
begin
regist.show
end;
```

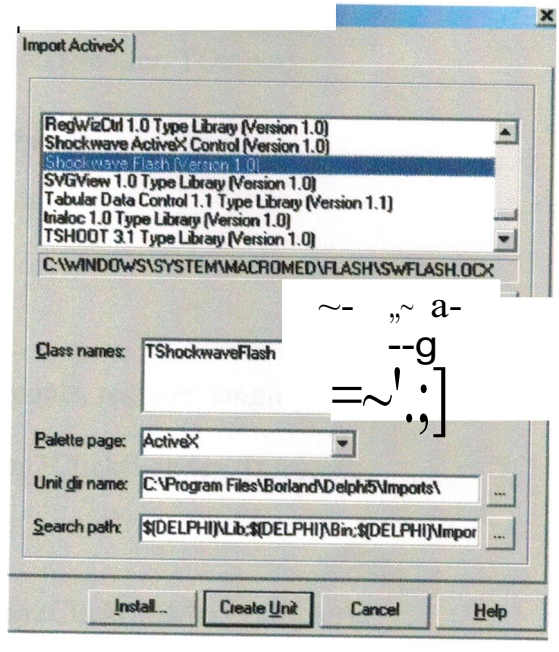

Figure 4.1. How to add an Shockwave component.

The nice thing that comes when you run the main menu is the animation that gives a beautiful look to the main menu. This animation have been done in Flash Application, and to use in Delphi you have first to add its component (see figure 4.1).

In Delphi "tool bar" there a "Component" dropdown menu choose "Import Activex" then choose "Sockwave Flash (vertion I)" then choose "Add" and "Ok".

The full code of the main menu is written bellow:

```
unit main_ menu;
interface
uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
  OleCtrls, ShockwaveFlashObjects _TLB, StdCtrls;
type
  TMain = class(TForrn)
    ShockwaveFlash I: TShockwaveFlash;
    Button 1: TButton;
    Button2: TButton;
    Button3: TButton;
    Button4: TButton;
    Buttons: TButton;
    Button6: TButton;
    Button7: TButton;
    procedure Button7Click(Sender: TObject);
    procedure Button2Click(Sender: TObject);
    procedure Button3Click(Sender: TObject);
    procedure Button4Click(Sender: TObject);
    procedure Button5Click(Sender: TObject);
    procedure Button6Click(Sender: TObj ect);
    procedure Button! Click(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;
  ar
Ylain: TMain;
implementation
uses reserve, room, accounts, register, employee, recept;
{$R *.DFM}
rocedure TMain.Button7Click(Sender: TObject);
egın
close
end;
 rocedure TMain.Button2Click(Sender: TObject);
 egın
res lshow
end;
```

```
procedure TMain.Button3Click(Sender: TObject);
begin
regist.show
end;
procedure TMain.Button4Click(Sender: TObj ect);
begin
rooms.show
end;
procedure TMain.Button5Click(Sender: TObject);
begin
empl.show
end;
procedure TMain.Button6Click(Sender: TObject);
begin
accountl .show
end;
procedure TMain.ButtonlClick(Sender: TObject);
begin
recep.show
end;
end.
```

4.2. Reception Menu

The reception menu is a collection of 4 "Push Buttons", each of these buttons has an address to specific form. The unit describing this menu is shown bellow:

```
unit recept;

interface
uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
  StdCtrls, ComCtrls;
type
  Trecep = class(TForm)
    Buttonl: TButton;
    Button2: TButton;
    Button3: TButton;
    Button4: TButton;
    Label l: TLabel;
    Button5: TButton;
    procedure Button5Click(Sender: TObject);
    procedure ButtonlClick(Sender: TObject);
    procedure Button2Click(Sender: TObject);
    procedure Button3Click(Sender: TObject);
    procedure Button4Click(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;
```

```
var
  recep: Trecep;
implementation
uses sname, ssumame, sroomno, employee, semp, phone, confirm, search,
 report;
 {$R *.DFM}
procedure Trecep.Button5Click(Sender: TObject);
begin
close
end;
procedure Trecep.ButtonlClick(Sender: TObject);
begin
phonel.show;
end;
procedure Trecep.Button2Click(Sender: TObject);
~&£İ8h 1. show;
end;
procedure Trecep.Button3Click(Sender: TObject);
begin
confirm 1. show;
end;
procedure Trecep.Button4Click(Sender: TObject);
begin
rep.show;
end;
end.
```

4.2.1 Search Menu

The search menu contains three submenus linked to the menu by "Push Buttons" just like the reservation menu, these submenus are:

1. Customer Name.
2. Room Number.
3. Employee Name.

Search By Customer Name

The search by customer name is written using SQL commands. In figure 4.2 it is clear seen that we used one SQL component with the command:

select from reservation.DB*

which will select all the fields in the specific table. Then There are two Data Sources when connected to the SQL and the other connected to the Tablel which is the "reservation" table.

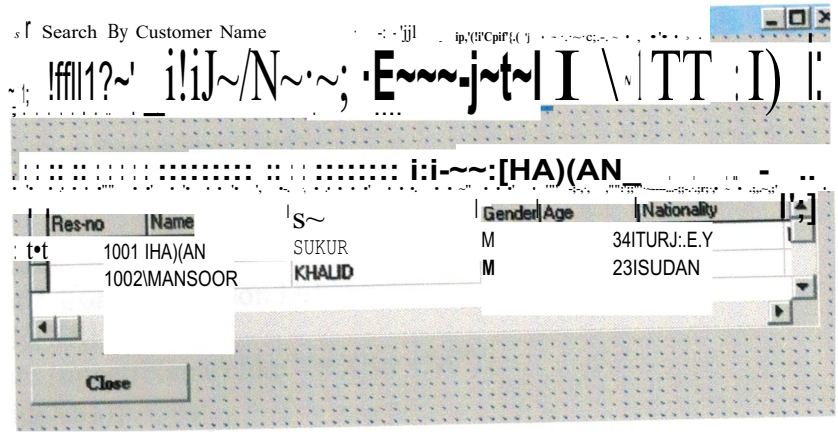

Figure 4.2. Search by name.

In the text box the following code should be written:

```
unit sname;
interface
uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
  StdCtrls, Db, DBTables, Mask, DBCtrls, Grids, DBGrids, ExtCtrls;
type
  Tsname1 = class(TForm)
    DBGri~2: TDBGrid;
    Namebox: TDBEdit;
    cold: TTable;
    Q1: TQuery;
    DSQ1: TDataSource;
    DS1: TDataSource;
    Button1 : TButton;
    Label1: TLabel;
    Label2: TLabel;
    procedure Button1Click(Sender: TObject);
    procedure NameboxChange(Sender: TObject);
    procedure FormCreate(Sender: TObject);
    procedure coldBeforePost(DataSet: TDataSet);
  private
    { Private declarations }
  public
    { Public declarations }
  end;
var
  sname1: Tsname1;
implementation
{$R *.DFM}
procedure Tsname1.Button1Click(Sender: TObject);
begin
close
end;
procedure Tsname1.NameboxChange(Sender: TObject);
```

```
var
  XStr, XSI, XS2 : string[IOO];
begin
  XStr:=UpperCase(NameBox.Text);
  Q1.DisableControls;
  QI.Close;
  Q1.SQL.Clear;
  XS1:='select * from reservation.DB D ';
  QI.SQL.Add(XSI);
  XSI:=' where (upper(Name) like"' +XStr +'%")';
  Ql.SQL.Add(XSI);
  XS1:=' order by Name';
  Ql.SQL.Add(XSI);
  QI.Open;
  Q1.EnableControls;
end;
procedure Tsnamel.FormCreate(Sender: TObject);
begin
cold.Open;
end;
procedure Tsnamel.coldBeforePost(DataSet: TDataSet);
begin
 Abort;
 end;
 end.
```

Search By Room No

In this search engine I have been used another strategy not like the one before, here there is indicator indicates the place of the matched room no. as shown in figure 4.3 there is no SQL used in this form, only Data Source and Table.

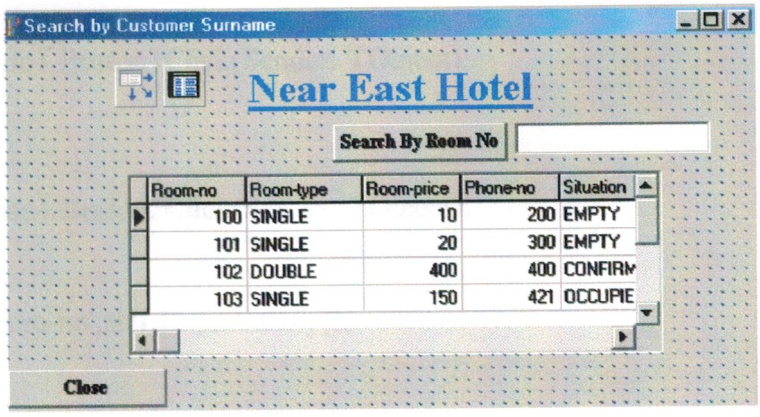

Figure 4.3. Search by room no.

In the text box the following code should be written:

```
unit roomno;
interface
uses
  Windows, Messages, Sysütils, Classes, Graphics, Controls, Forms, Dialogs.
  Db, DBTables, StdCtrls, Mask, DBCtrls, Grids, DBGrids;
type
  Troomnol = class(TForm)
    Label 1: TLabel;
    Buttonl: Tlsutton;
    Tablel: TTable;
    DataSource 1'. TDataSource;
    DBGridl: TDBGrid;
    Button2: TButton;
    edit1; !Edit;
    procedure ButtonlClick(Sender: TObject);
   procedure FormCreate(Sender: TObiect);
    procedure co1dBeforePost(DataS€t: TDataS€t);
    procedure Button2Click(Sender: TObject);
  private
        str 1 : string;
    { Private declarations }
  public
    { Public declarations }
  end;
var
  roomno 1: Troomno 1;
implementation
{$R *.DFM}
procedure Troomnol.FormCreate(Sender: 'I'Object);
begin
and;
procedure Troomno 1. coldBeforePost(DataSet: TDataSet );
begin
 Abort;
 end;
procedure Troomnol,ButtonlClick(Sender: TObject);
begin
close
end;
procedure Troomno 1.Button2Click(Sender: TObject );
begin
  str 1 :=editl .text;
table 1.Locate('Room-no' .str 1,[lopartialkey ]);
€nd;
end.
```

4.2.2 The Confirmation Form

As it is clearly seen in the figure 4.3 this form join two tables one as master and the other as child. But these two tables are both the "reservation.db" table, a question is that why don't we use this table only one time? Because we need after we write the reservation id other information to come automatically.

Consider that only "res-id" text field connected to the master table other information that we want to get automatically must be connected to the other table.

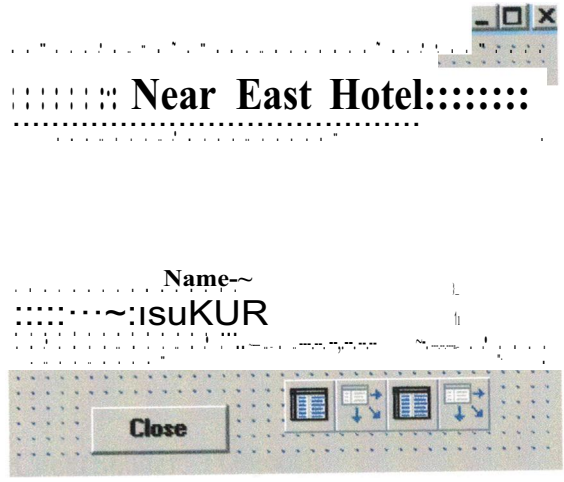

Figure 4.3. Confirmation form.

The confirmation code window is shown bellow:

```
unit confirm;
interface
uses
  Windows, Messages, Classes, SysUtils, Graphics, Controls, StdCtrls, Forms,
  Dialogs, DBCtrls, DB, Mask, DBTables, ExtCtrls;
type
  Tconfuml  = class(TForm)
    Labell : TLabel;
    Label4: TLabel;
    Label2: TLabel;
    Exit: TButton;
    Label3: TLabel;
    Label5: TLabel;
    EditResno: TDBEdit;
    EditConfumed2: TDBEdit;
    EditName2: TDBEdit;
    EditSurname2: TDBEdit;
    DataSource2: TDataSource;
    Table2: TTable;
    Table2Resno: TFloatField;
    Table2Name: TStringField;
    Table2Surname: TStringField;
```

```
  Table2Confirmed: TStringField;
  DataSource 1: TDataSource;
  Tablel: TTable;
  TablelResno: TFloatField;
  TablelName: TStringField;
  Tablel Surname: TStringField;
  Table 1 Confirmed: TStringField;
  procedure FormCreate(Sender: Tübject);
  procedure ExitClick(Sender: TObject);
 private
   { private declarations }
 public
   { public declarations }
 end;
var
 confirm 1: Tconfirm 1;
implementation
{$R *.DFM}
procedure Tconfirml.FormCreate(Sender: TObject);
begin
  Tablel.Open;
  Table2.0pen;
end;
procedure Tconfirml.ExitClick(Sender: Tübject);
begin
close
end;
end.
```

4.2.3 The Lists Submenu

The lists form is a simple menu containing two "Push Buttons" for the rooms lists and reservation lists.

Rooms Lists

This lists consisting of three components as seen in figure 4.4 these three components are:

1. All Rooms List: when this button pressed all the rooms in the hotel will be displayed. Here a simple SQL command written, this command is:

```
procedure TMainR.SQLI BtnClick(Sender: TObject);
 begin
   QI. DisableControls;
   QI.Close;
   QI.SQL.Clear;
   XSQL:='SELECT * FROM Rooms.db';
   QI. SQL.Add(XSQL);
   QI.Open;
```

```
  Ql.EnableControls;
end;
```

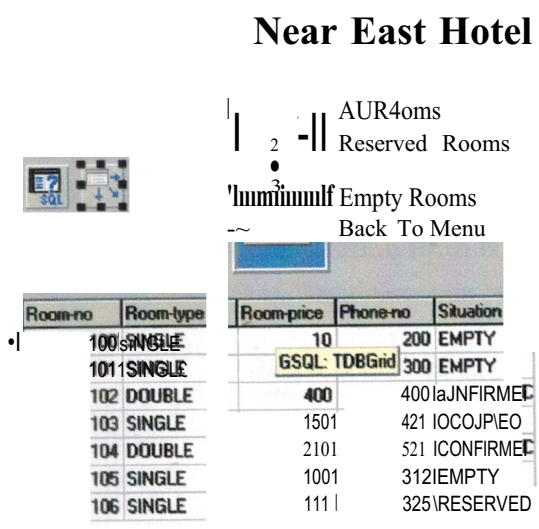

Figure 4.4. Rooms Queries.

2. Reserved Rooms List: when this button pressed the reserved rooms only will appear at the list. The following procedure is wrote to get the reserved rooms only.

```
procedure  TMainR.SQL2BtnClick(Sender: TObject);
 begin
 Q1.DisableControls;
   Ql.Close;
   QJ.SQL.Clear;
  XSJ:='SELECT * FROM Rooms.db';
  XSQL:=XSJ+'  WHERE Situation LIKE  "RESERVED"';     II added 2 strings rather than I
 long string
   QI.SQL.Add(XSQL);
   QI.Open;
   QI.EnableControls;
 end;
```

3. Empty Rooms List: when this button pressed only empty rooms will be listed, To get this list the following procedure should be written down:

```
procedure    TMainR.SQL3BtnClick(Sender:      TObject);
 begin
 Q1. DisableControls;
    Qi.Close;
    QI .SQL. Clear;
   XS1:='SELECT    * FROM  Rooms.db';
```

```
  XSQL: =XSJ +' WHERE Situation LIKE "EMPTY" '; II added 2 strings rather than
I long string
  QI.SQL.Add(XSQL);
  QI.Open;
  QI.EnableControls;
end;
```

4.3. Registration Form

The registration form is a written using information from three different tables as shown in figure 4.5, the code of the registration form is shown bellow:

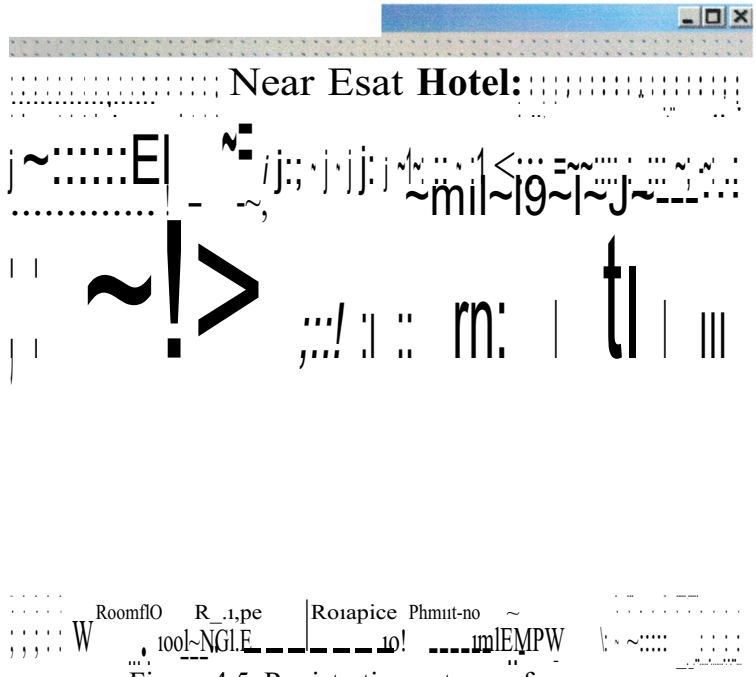

Figure 4.5. Registration entrance form.

```
unit register;
interface
uses
  Windows, Messages, Classes, SysUtils, Graphics, Controls, StdCtrls, Forms,
  Dialogs, DBCtrls, DB, DBTables, Mask, ExtCtrls, Grids, DBGrids;
type
  Tregist = class(TForm)
    Table2Resno: TFloatField;
    Table2Name: TStringField;
    Table2Surname: TStringField;
    Table2Gender: TStringField;
    Table2Age: TFloatField;
    Table2Nationality: TStringField;
    Table2Passno: TStringField;
    Table2Address: TStringField;
    Table2Phone: TFloatField;
    Table2Resdate: TDateField;
```

```
Table2Checkin:  TDateField;
Table2Checkout:  TDateField;
Table2Confirmed:  TStringField;
Table2Roomno:  TFloatField;
TablelRegno:  TFloatField;
TablelResno:  TFloatField;
TablelName:  TStringField;
Table l Surname:  TStringField;
Table 1 Gender:  TStringField;
TablelAge:  TFloatField;
TablelNationality:  TStringField;
Table lPassno:  TStringField;
TablelAddress:  TStringField;
Table 1 Checkin:  TDateField;
Table 1 Checkout:  TDateField;
TablelLeavedate:  TDateField;
TablelRoomno:  TFloatField;
DataSource 1:  TDataSource;
Tablel:  TTable;
Table2:  TTable;
DataSource2:  TDataSource;
Table3:  TTable;
DataSource3:  TDataSource;
EditRegno:  TDBEdit;
Label 1:  TLabel;
Label2:  TLabel;
EditResno:  TDBEdit;
Labell 5:  TLabel;
EditName2:  TDBEdit;
EditSumame2:  TDBEdit;
Labell6:  TLabel;
Label 17:  TLabel;
EditGender2:  TDBEdit;
EditAge2:  TDBEdit;
Label 18:  TLabel;
Labell 9:  TLabel;
EditNationality2:  TDBEdit;
EditPassno2:  TDBEdit;
Label20:  TLabel;
Label2 1:  Tl.abel;
EditAddress2:  TDBEdit;
EditPhone:  TDBEdit;
Label22:  TLabel;
EditCheckin2:  TDBEdit;
Label24:  TLabel;
Labell2:  TLabel;
EditLeavedate:  TDBEdit;
EditRoomno:  TDBEdit;
Labell 3:  TLabel;
DBGrid 1:  TDBGrid;
```

```
    DBNavigatorl:    TDBNavigator;
    Label3:  TLabel;
    Buttonl:    TButton;
    Button2:   TButton;
    procedure   FormCreate(Sender:    TObject);
    procedure   Button 1 Click(Sender:   TObject );
    procedure   Button2Click(Sender:    TObject);
  private
    { private declarations  }
  public
    { public declarations  }
  end;
var
  regist:  Tregist;
implementation
uses  confirm;
 {$R  *.DFM}
procedure   Tregist.F ormCreate(Sender:    TObject );
begin
  Table 1.Open;
  Table2.0pen;
end;
procedure Tregist.ButtonlClick(Sender:  TObject);
begin
close
end;
procedure Tregist.Button2Click(Sender: TObject);
begin
confirm1. show
end;
end.
```

4.4. The Accounts Form

All the financial operations will be done here through this form. The most interesting thing in this form that this form takes its information from four different tables, only the employee writes the accounts id and the registration id then all the personal and financial information come automatically. The accounts form show in figure 4.6.

When the "Calculate Total" button pressed the total that the customer should pay will comes out. The calculation button has the following summation procedure:

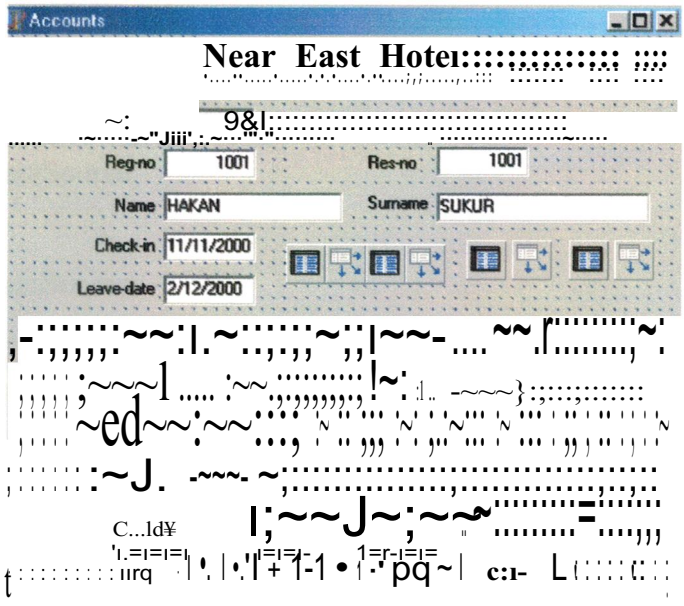

Figure 4.6. The accounts form.

```
procedure  Taccountl.Button2Click(Sender:      TObject);
begin
Tab/el.Edit;
Table] [general-
amount']:  =table] ['total']  +tablel  ['restaurant']+  table 1 [tactivities'],
Tab/el.Post;
end;
```

The accounts form has the following code:

```
unit accounts;
interface
uses
  Windows, Messages, Classes, SysUtils, Graphics, Controls, StdCtrls, Forms,
  Dialogs, DBCtrls, DB, DBTables, Mask, ExtCtrls;
type
  Taccountl  =  class(TForm)
    Table2Regno: TFloatField;
    Table2Resno: TFloatField;
    Table2Checkin: TDateField;
    Table2Leavedate: TDateField;
    Table2Roomno: TFloatField;
    TablelAccno:  TFloatField;
    TablelRegno:  TFloatField;
    TablelName:  TStringField;
    Table1 Surname: TStringField;
    TablelRoomno:  TFloatField;
    TablelRoomprice:  TCurrencyField;
```

```
Table1Noofdays:  TFloatField;
Table! Total:  TCurrencyField;
TablelRestaurant:  TCurrencyField;
TablelActivities:  TCurrencyField;
Table 1 Generalamount:  TCurrencyField;
DataSource 1:  TDataSource;
Table 1:  TTable;
Table2:  TTable;
DataSource2:  TDataSource;
Label 1 :  Tl.abel;
EditAccno:  TDBEdit;
EditResno:  TDBEdit;
Label 13:  TLabel;
Label3:  TLabel;
Label4:  TLabel;
Label6:  TLabel;
Label?:  TLabel;
Label8:  TLabel;
Label9:  TLabel;
Labell 0:  TLabel;
Labell 1:  TLabel;
EditGeneralamount:  TDBEdit;
EditActivities:  TDBEdit;
EditRestaurant:  TDBEdit;
EditTotal:  TDBEdit;
EditRoomprice:  TDBEdit;
EditSumame:  TDBEdit;
EditName:  TDBEdit;
EditRegno:  TDBEdit;
Label2:  TLabel;
Labell 4:  TLabel;
EditCheckin:  TDBEdit;
Label 15:  TLabel;
EditLeavedate:  TDBEdit;
EditRoomno2:  TDBEdit;
Labell 6:  TLabel;
EditNoofdays:  TDBEdit;
DBNavigatorl:  TDBNavigator;
Label5:  TLabel;
Buttonl:  TButton;
Table3:  TTable;
DataSource3:  TDataSource;
Table4:  TTable;
DataSource4:  TDataSource;
Button2:  TButton;
Label12:  TLabel;
Labell 7:  TLabel;
Label 18:  TLabel;
Labell 9:  TLabel;
Label20:  TLabel;
```

```
    procedure  FormCreate(Sender:    TObject);
    procedure  Button 1Click(Sender:  TObject );
    procedure  EditNoofdaysChange(Sender:     TObject);
    procedure  Button2Click(Sender:    TObject);
  private
    { private declarations  }
  public
    { public declarations  }
  end;
var
  accountl:   Taccount 1;
implementation
{$R  *.DFM}
procedure  Taccountl.FormCreate(Sentler:      TObject);
begin
  Table 1. Open;
  Table2.0pen;
end;

procedure  Taccountl.ButtonlClick(Sender:       TObject);
begin
close
end;
procedure  Taccount 1.EditNoofdaysChange(Sender:     TObject );
begin            .
//EditNoofdaysChange=EditLeavedate-EditCheckin;
end;
procedure  Taccountl.Button2Click(Sender:      TObject);
begin
Tablel .Edit;
//Table4 .Edit;
Table 1 ['general-amount']:=table  1 ['total']+table 1 ['restaurant']+table  1 [activities'];
Table I .Post;
//Table4 .Post;
end;
end.
```

4.5. Constructing a Package

The last step here is to construct the Package that help this program to work in different computers the steps of constructing a package is as follow:

- Choose File
- Choose New
- Choose Package
- Choose Add
- Select your *.pas files

- Choose compile

The following code will wrote down after the operation finishing from packaging.

The code shows all units, forms and applications that uses in the "Near East Hotel":

```
program hotel;
uses
  Forms,
  phone in 'phone. pas' {phone 1},
  confirm in 'confirm. pas' {confirm 1},
  employee in 'employee.pas' {Empl},
  register in 'register. pas' {regist},
  reserve in 'reserve. pas' {res l},
  room in 'room. pas' {Rooms},
  sname in 'sname.pas' {snamel},
  ssurname in 'ssurname.pas' {ssurnam},
  semp in 'semp.pas' {sempl},
  accounts in 'accounts. pas' {account 1},
  mairırnenu in 'mainrnemı.pas' {Main},
  search in 'search. pas' {search 1},
  report in 'report. pas' {Rep},
  recept in 'recept. pas' {recep},
  roomreport in 'roomreport. pas' {MainR},
  resreport in 'resreport. pas' {MainF};

{$R *.RES}

begin
  Application.Initialize;
  Application. CreateF orm(Tssurnam, ssurnam);
  Application.CreateForm(TMain, Main);
  Application. CreateForm(Tconfirml, confirm I);
  Application. CreateF orm(Taccount 1, account l);
  Application. CreateF orm(Tsemp 1, semp l);
  Application. CreateF orm(TRooms, Rooms);
  Application. CreateF orm(TMainF, MainF);
  Application. CreateF orm(Tres 1, res l);
  Application. CreateF orm(TRep, Rep);
  Application. CreateF orm(TMainR, MainR);
  Application. CreateForm(Tphone 1, phone l);
  Application. CreateF orm(Tregist, regist );
  Application. CreateF orm(TEmp 1, Emp l);
  Application. CreateF orm(Tsname 1, sname l);
  Application. CreateForm(Tsearch 1, search l);
  Application. CreateForm(Trecep, recep);
  Application.Run;
end.
```

Chapter Five
STRUCTURED QUERY LANGUAGE (SQL)

The importance of the SQL is due to its popularity and the simplest implementation and its powerful effecting on the database.

m \n\ı ~Y\.'a.1)\et,'ıne rnaın eıements, main functions and general statements of the SQL will be described.

I have wrote a separate chapter to the SQL because in the "Near East Hotel" software application I have used SQL statements so many times to derive different queries.

Each topic contains a pseudo-code prototype of an SQL statement or part of a statement that demonstrates the language element discussed. These prototype examples appear at the beginning of topics. Actual SQL statements appear within topics that demonstrate actual use of the language element discussed. Both the prototype and example statements appear in Courier New font.

While the local SQL language itself is case-insensitive (language elements and metadata object names), examples in this help file use the following convention to differentiate between language and metadata objects. All language elements appear in uppercase. Metadata names appear in lowercase. Correlation names appear in mixed case.

5.1. Optional Elements

Language elements that are available, but that do not have to be used with an SQL statement for the statement to be valid appear in prototype syntax examples in brakets ([and]). For example, in the line below, the DISTINCT keyword is optional.

SELECT [DISTINCT} *

5.1.1 Syntax choices

When there is a choice between one of a number of possible syntax elements, such choices will be listed in prototype syntax examples separated by the vertical bar character(!). Unless also enclosed in brackets to make the group of choices optional, one of the group of choices must be used in the statement. In the prototype syntax example belowthe SQL statement may include either the ASC or the DESC keyword, but not

both. Because the list of choices (ASC and DESC) is enclosed in brackets, use of either keyword is optional.

ORDER BY column_reference [ASC | DESC]

5.1.2 Table Names

ANSI-standard SQL confines each table name to a single word comprised of alphanumeric characters and the underscore symbol, "_" Local SQL, however, is enhanced to support multi-word table names.

Local SQL supports full :file and path specifications in table references. Table references with path or filename extensions must be enclosed in single or double quotation marks. For example:

```
SF,J,F,CT*
FROM 'parts.dbf
SELECT*
FROM "c: \sample ıparts.dbf"
```

Local SQL also supports BDE aliases in table references. For example:

```
SELECT*
FROM "tpdoxttable l"
```

If you omit the file extension for a local table name, the table is assumed to be the table type specified in the BDE configuration. The default table type is specified either in the default driver setting or in the default driver type for the standard alias associated with the query.

Finally, local SQL permits table names to duplicate SQL keywords as long as those table names are enclosed in single or double quotation marks. For example:

```
SELECT passid
FROM "password"
```

5.1.3 Column Names

ANSI-standard SQL confines each column name to a single word comprised of alphanumeric characters and the underscore symbol, "_". Local SQL, however, is enhanced to support multi-word column names. Local SQL supports Paradox multi-word column names and column names that duplicate SQL keywords as long as those column names are Enclosed in single or double quotation marks prefaced with an SQL

table name or table correlation name. For example, the following column name consists of two words:

SELECT E. "EmpId"

FROM employee E

In the next example, the column name is the same as the SQL keyword DATE:

SELECT datelog. "date"

FROM datelog

5.1.4 Date Formats

Local SQL expects date literals to be in a U.S. date format, MMIDD/YY or MMIDD/YYYY. International date formats are not supported. To prevent date literals from being mistaken by the SQL parser for arithmetic calculations, enclose them in quotation marks. This keeps 1/23/1998 from being mistaken for 1 divided by 23 divided by 1998.

*SELECT**

FROM orders

WHERE (saledate <= "1123/1998':)

Leading zeros for the month and day fields are optional. If the century is not specified for the year, the BDE setting FOURDIGITYEAR controls the century. If FOURDIGITYEAR is set to FALSE and the year is specified with only two digits, years 49 and less will be prefiex with 20 and years 50 and higher with 19. IfFor example, with FOURDIGITYEAR set to FALSE, the SQL statement below returns rows where the SaleDate column contains dates of “5/5/1980” or “5/5/2030&#8221 ;.

*SELECT**

FROJ1 orders

WHERE (saledate = "5/5/30'~)OR (saledate = "5/5/80'~)

To query using years outside these bounds, specify the century in the date literal.

*SELECT**

FROM orders

WHERE (saledate = "5/5/1930'_')OR (saledate = "5/5/2080")

5.1.5 Time Formats

Local SQL expects time literals to be in the forınat hh:mm:ss AM/PM; where hh are the hours, mm the minutes, and ss the seconds. When inserting new data with a time value, the AM/PM designator is optional and is case-insensitive ("AM" is the same as "am"). The time literal must be enclosed in quotation marks.

```
INSERT INTO Workôrder
(ID, StartTime)
VALUES ("BOOJ20". "10:30:00 PM':)
```

Indicate which half of the day (morning or after noon) a time literal falls under in one of two ways. If an AM or PM marker is specified, that deterınines the half of the day. If no AM/PM designator is specified, the hour field is compared to 12. If the hour is less than twelve, the time is in the AM; if greater than 12, after noon. The hour field overrides an AM/PM designator. For example, the time literal "15:03:22 AM" is translated as "3 :03 :22 PM".

5.1.6 Boolean Literals

The boolean literal values TRUE and FALSE may be represented with or without
quotation marks.

```
SELECT*
FROM transfers
WHERE (paid= TRUE) AND NOT (incomplete = "FALSE'~)
```

5.-1.7 Table· Correlation Names.

Table correlation names are used to explicitly associate a column with the table from which it comes. This is especially useful when multiple columns of the same name appear in the same query, typically in multi-table queries. A table correlation name is defined by following the table reference in the FROM clause of a SELECT query with a unique identifier. This identifier, or table correlation name, can then be used to prefix a column name.

If the table name is not a quoted string, the table name is the default implicit correlation name. An explicit correlation name the same as the table name need not be specified in the FROM clause and the table name can prefix column names in other parts ofjhe staıemenr.

```
SELECT*
FROM customer
LEFT OUTER JOIN orders
ON (customer.custno = orders.custno)
```

If the table name is a quoted string, you need to do one of the following:

Prefix column names with the exact quoted string used for the table in the FROM clause.

```
SELECT*
FROM "customer.db"
LEFT OUTER JOIN "orders.db"
ON ("customer.db".custno = "orders.db'ıcustno)
```

Use the full table name as a correlation name in the FROM clause (and prefix all column references with the same correlation name).

```
SELECT*
FROM "customer.db" CUSTOMRR
LEFT OUTER JOIN "orders.db" ORDERS
ON (CUSTOMER.custno = ORDERS.custno)
```

Use a distinct token as a correlation name in the FROM clause (and prefix all column references with the same correlation name).

```
SELECT*
FROM "customer.db" C
LEFT OUTER JOIN "orders.db" O
ON (C.custno = 0.custno)
```

5.1.8 Column Correlation Names

Use the AS keyword to assign a correlation name to a column, aggregated value, or literal. Column correlation names cannot be enclosed in quotation marks and so cannot contain embedded spaces.

```
SELECT SUBSTRING(company FROM I FOR I) AS sub, "Text" AS word
FROM customer
```

5.2. DML Statement List

Local SQL supports the following data manipulation language (DML) statements:

SELECT Retrieves existing data from a table.

DELETE Deletes existing data from a table.

INSERT Adds new data to a table.

UPDATE Modifies existing data in a table.

5.2.1 Select Statement

Retrieves data from tables.

*SELECT [DISTINCT] * | column_list*

FROM table_reference

[WHERE predicates]

[ORDER BY order_list]

[GROUP BY group_list}

[HAVING having_condition]

Description:

Use the SELECT statement to Retrieve a single row, or part of a row, from a table, referred to as a singleton select. Retrieve multiple rows, or parts of rows, from a table. Retrieve related rows, or parts of rows, from a join of two or more tables. The SELECT clause defines the list of items returned by the SELECT statement.The SELECT clause uses a comma-separated list composed of: table columns, literal values, and column or literal values modified by functions. Literal values in the columns list may be passed to the SELECT statement via parameters. You cannot use parameters to represent column names. Use an asterisk to retrieve values from all columns.

Columns in the column list for the SELECT clause may come from more than one

table, but can only come from those tables listed in the FROM clause. See Relational Operators for more information on using the SELECT statement to retrieve data from multiple tables.

The FROM clause identifies the table(s) from which data is retrieved. The following statement retrieves data for two columns in all rows of a table.

SELECT custno, company

FROM orders

Use DISTINCT to limit the retrieved data to only distinct rows. The distinctness of rows is based on the combination of the columns in the SELECT clause columns list.

In lieu of a table, a SELECT statement may retrieve rows from a Paradox-style .QBE file. This is an approximation of an SQL view.

*SELECT**

FROM "customers.qbe"

5.2.2 Delete Statement

Deletes one or more rows from a table.

DELETE FROM table_reference

[WHERE predicates J

Description

Use DELETE to delete one or more rows from an existing table.

DELETE FROM "employee.db"

The optional WHERE clause restricts row deletions to a subset of rows in the table. If no WHERE clause is specified, all rows in the table are deleted.

DELETE FROM "employee.db"

WHERE (empno IN (SELECT empno FROM "old_employee.db"))

The table reference cannot be passed to the DELETE statement via a parameter.

5.2.3 Insert Statement

Adds one or more new rows of data in a table

INSERT INTO table_reference

[(columns_lis~J

VALUES (update_atoms)

Description:

Use the INSERT statement to add new rows of data to a table. Use a table reference in the INTO clause to specify the table to receive the incoming data.

The columns list is a comma-separated list, enclosed in parentheses, of columns in the table and is optional. The VALUES clause is a comma-separated list of update atoms, enclosed in parentheses. If no columns list is specified, incoming update values (update atoms) are stored in fields as they are defined sequentially in the table structure. Update atoms are applied to columns in the order the update atoms are listed in the VALUES clause. There must also be as many update atoms as there are columns in the table.

INSERT INTO "holdings.dbf'

VALUES (4094095, "BORL", 5000, 10.500, "11211998")

If an explicit columns list is stated, incoming update atoms (in the order they appear in the VALUES clause) are stored in the listed columns (in the order they appear in the columns list). NULL values are stored in any columns that are not in a columns list.

!NSF.RTTNTO "customer.db"

(custno, company)

VALUES (9842, "Borland International, Inc.")

To add rows to one table from another, omit the VALUES keyword and use a subquery as the source for the new rows.

INSERT INTO "customer.db"

(custno, company)

SELECT custno, company

FROM "oldcustomer.db"

Update atom values may be passed to the INSERT statement via parameters. You cannot use parameters for the table reference and columns list. Note Insertion of one or multiple rows from one table to another through a subquery is not supported.

5.2.4 Update Statement

Modifies one or more existing rows in a table.

UPDATE table reference

SET column_ref= update_atom[, column_ref= update_atom…}

[WHERE predicates J

Description:

Use the UPDATE statement to modify one or more column values in one or more existing rows in a table. Use a table reference in the UPDATE clause to specify the table to receive the data changes. The SET clause is a comma-separated list of update expressions. Each expression is composed of the name of a column, the assignment operator(=), and the update value (update atom) for that column. The update atoms in any one update expression may be literal values, singleton return values from a subquery, or update atoms modified by functions. Subqueries supplying an update atom for an update expression must return a singleton result set (one row) and return only a single column.

UPDATE salesinfo

SET taxrate = 0.0825

WHERE (state = "CA")

Update atom values may be passed to the UPDATE statement via parameters. You cannot use parameters for the table reference and columns list. The optional WHERE clause restricts updates to a subset ofrows in the table. Ifno WHERE clause is specified, all rows in the table are updated using the SET clause update expressions.

5.3. Clause List

Local SQL supports the following SQL statement clauses:

FROM Specifies the tables used for the statement.

WHERE Specifies filter criteria to limit rows retrieved.

ORDER BY Specifies the columns on which to sort the result set.

GROUP BY Specifies the columns used to group rows.

HAVING Specifies filter criteria using aggregated data.

5.3.1 From Statement

Specifies the Tables from which a SELECT statement retrieves data.

FROM table_reference {, table~reference…}

Description:

Use a FROM clause to specify the table or tables from which a SELECT statement retrieves data. The value for a FROM clause is a comma-separated list oftable names. Specified table names must follow local SQL naming conventions for tables. For example, the SELECT statement below retrieves data from a single Paradox table.

*SELECT**

FROM "customer.db"

See the section Relational Operators for more information on retrieving data from multiple tables in a single SELECT query. The table reference cannot be passed to a FROM clause via a parameter.

5.3.2 Where Statement

Specifies filtering conditions for a SELECT or UPDATE statement.

WHEREpredicates

Description:

Use a WHERE clause to limit the effect of a SELECT or UPDATE statement to a subset of rows in the table. Use of a WHERE clause is optional. The value for a WHERE clause is one or more logical expressions, or predicates, that evaluate to TRUE or FALSE for each row in the table. Only those rows where the predicates evaluate to TRUE are retrieved by a SELECT statement or modified by an UPDATE statement. For example, the SELECT statement below retrieves all rows where the STATE column contains a value of"CA".

```
SELECT company, state
FROM customer
WHERE state = "CA"
```

Multiple predicates must be separated by one of the logical operators OR or AND.

Each predicate can be negated with the NOT operator. Parentheses can be used to isolate logical comparisons and groups of comparisons to produce different row evaluation criteria. For example, the SELECT statement below retrieves all rows where the STATE column contains a value of"CA" and those with a value of"lll".

```
SELECT company, state
FROM customer
WHERE (state = "CA") OR (state = "Hl")
```

The SELECT statement below retrieves all rows where the SHAPE column is "round" or "square". It also returns those rows where the COLOR column is "red", regardless of the value in the SHAPE column.

```
SELECT shape, color, cost
FROM objects
WHERE ((shape = "round") AND (shape = "square")) OR (color = "red")
```

Subqueries are supported in the WHERE clause. A subquery works like a search condition to restrict the number of rows returned by the outer, or "parent" query. Column references cannot be passed to a WHERE clause via parameters. Comparison values may be passed as parameters. For filtering based on aggregated values, use a HAVING clause.

5.3.3 Order By Statement

Sorts the rows retrieved by a SELECT statement.

ORDER BY column_reference [, column_reference ...} [ASC\DESCJ

Description

Use an ORDER BY clause to sort the rows retrieved by a SELECT statement based on the values from one or more columns. The value for the ORDER BY clause is a comma-separated list of column names. The columns in this list must also be in the SELECT clause of the query statement. Columns in the ORDER BY list can be from one or multiple tables. A number representing the relative position of a column in the SELECT clause may be used in place of a column name. Column correlation names can also be used in an ORDER BY clause columns list.

Use ASC (or ASCENDING) to force the sort to be in ascending order (smallest to largest), or DESC (or DESCENDING) for a descending sort order (largest to smallest). When not specified, ASC is the implied default. The statement below sorts the result set ascending by the year extracted from the LASTJNVOICEDATE column, then descending by the STATE column, and then ascending by the uppercase conversion of the COMPANY column.

SELECT EXTRACT(YEAR FROM lastinvoicedate) AS YY, state, UPPER(company)
FROM customer
ORDER BY YY DESC, state ASC, 3

See the section Relational Operators for more information on retrieving data from multiple tables in a single SELECT query. Column references cannot be passed to an ORDER BY clause via parameters.

5.3.4 Group By Statement

Combines rows with column values in common into single rows.

GROUP BY column_reference[, column reference ...]

Description

Use a GROUP BY clause to combine rows with the same column values into a single row. The criteria for combining rows is based on the values in the columns specified in the GROUP BY clause. The purpose for using a GROUP BY clause is to combine one or more column values (aggregate) into a single value and provide one or

more columns to uniquely identify the aggregated values. A GROUP BY clause can only be used when one or more columns have an aggregate function applied to them.

The value for the GROUP BY clause is a comma-separated list of columns. Each column in this list must meet the following criteria:

Be in one of the tables specified in the FROM clause of the query.

Be in the SELECT clause of the query.

Cannot have an aggregate function applied to it.

When a GROUP BY clause is used, all table columns in the SELECT clause of the query must meet at least one of the following criteria, or it cannot be included in the SELECT clause:

Be in the GROUP BY clause of the query.

Be in the subject of an aggregate function.

Literal values in the SELECT clause are not subject to the preceding criteria.

The distinctness of rows is based on the columns in the column list specified. All rows with the same values in these columns are combined into a single row (or logical _group). Columns that are the subject of an aggregate function have their values across all rows in the group combined. All columns not the subject of an aggregate function retain their value and serve to distinctly identify the group. For example, in the SELECT statement below, the values in the SALES column are aggregated (totaled) into groups based on distinct values in the COMP ANY column. This produces total sales for each company.

```
SELECT company, SUM(sales) AS TOTALSALES
FROMsales1998
GROUP BY company
ORDER BY company
```

A column may be referenced in a GROUP BY clause by a column correlation name, instead of actual column names. The statement below forms groups using the first column, COMPANY, represented by the column correlation name Co.

```
SELECT company AS Co, SUM(sales) AS TOTALSALES
FROM sales 1998
GROUPBYCo
ORDER BY I
```

5.3.5 Having Statement

Specifies filtering conditions for a SELECT statement.

HAVING predicates

Description:

Use a HAVING clause to limit the rows retrieved by a SELECT statement to a subset of rows where aggregated column values meet the specified criteria. A HAVING clause can only be used in a SELECT statement when:

The statement also has a GROUP BY clause.

One or more columns are the subjects of ags;regate functions.

The value for a HAVING clause is one or more logical expressıons, or predicates, that evaluate to true or false for each aggregate row retrieved from the table. Only those rows where the predicates evaluate to true are retrieved by a SELECT statement. For example, the SELECT statement below retrieves all rows where the total sales for individual total sales exceed $1,000.

```
SELECT company, SUM(sales) AS TOTALSALES
FROM salesl998
GROUP BY company
HAVING (SUM(sales) >= 1000)
ORDER BY company
```

Multiple predicates must be separated by one of the logical operators OR or AND.

Each predicate can be negated with the NOT operator. Parentheses can be used to isolate logical comparisons and groups of comparisons to produce different row evaluation criteria.

A SELECT statement can include both a WHERE clause and a HA VTNG clause. The WHERE clause filters the data to be aggregated, using columns not the subject of aggregate functions. The HAVING clause then further filters the data after the aggregation, using columns that are the subject of aggregate functions. The SELECT query below performs the same operation as that above, but data limited to those rows where the STATE column is "CA".

```
SELECT company, SUM(sales) AS TOTALSALES
FROM salesl998
 WHERE (state = "CA")
 GROUP BY company
```

HAVING (SUM(sales) >= 1000)

ORDER BY company

Sub-queries are supported in the HAVING clause. A sub-query works like a search condition to restrict the number of rows returned by the outer, or "parent" query.

CONCLUSION

The power of databases comes from a body of knowledge and technology that has developed over several decades and is embodied in specialized software called a database management system, or DBMS, or more colloquially a database system." A DBMS is a powerful tool for creating and managing large amounts of data efficiently and allowing it to persist over long periods of time-safely. These systems are among the most complex types of software available.

The aim of the project is to develop a hotel management system using Delphi Programming Language. It is clearly seen that using Delphi, you can create highly efficient applications with a minimum of manual coding.

In the Project I have described the Database Management System and its attributes and methods. the importance of the database design also have been described showing how a good design affecting the programming language.

The Delphi Programming Language is described in a separate chapter showing its efficiency and its simplest designing, coding and testing.

Chapter Four and Five about the Hotel Management System software that I have wrote using Delphi. How is the system works and how is the code wrote is fully described.

The last chapter is describing the Structured Query Language (SQL).

REFERENCES

1. http://www.ira.uka.de/bibliography/Database.

2. Abiteboul, S., R. Hull, and V. Vianu, Foundations of Databases, Addison-Wesley, Reading, MA, 1995.

3. M. M. Astrahan et al., "System R: a relational approach to database management," ACM Trans. on Database Systems 1:2 (1976), pp. 97{137.4. P.A. Bernstein et al., "The Asilomar report on database research,"

4. http://s2k-ftp.cs.berkeley.edu:8000/postgres/papers/AsilomarFinal.htm.

5. http://www.informatik.uni-trier.de/-ley/db/index.html . A mirror site is found at http://www.acm.org/sigmod/dblp/db/index.html.

6. Stonebraker, M. and J.M. Hellerstein (eds.), Readings in Database Systems, Morgan-Kaufmann, San Francisco, 1998.

7. M. Stonebraker, E. Wong, P. Kreps, and G. Held, "The design and implementation of INGRES," ACM Trans. on Database Systems 1:3 (1976), pp. 189/222.

8. Ullman, J. D., Principles of Database and Knowledge-Base Systems, Volume I, Computer Science Press, New York, 1988.

9. Ullman, J. D., Principles of Database and Knowledge-Base Systems, Volume II, Computer Science Press, New York, 1989.

10. Marvel Wadbana, Programming With Delphi, Computer Science Press, New York, 1999.