
Business Logic Layer and Data Logic Layer

Maria Bika

Athens 2003
TABLE OF CONTENTS

CHAPTER 0: General......................................................................................1
   0.1 General.................................................................................................1

CHAPTER 1: Introduction..............................................................................3
   1.1 General.................................................................................................3
   1.2 System Usage Analysis..........................................................................3
   1.3 System Aim............................................................................................4
   1.4 3-Tier Application..................................................................................5
   1.5 System Description................................................................................6
   1.6 Tools Used..............................................................................................7
       1.6.1 Creating a Program in Visual Studio .NET 2003..........................7
   1.7 Basic Definitions....................................................................................8

CHAPTER 2: Logical Design of the Relational Database...............................10
   2.1 Entity Analysis......................................................................................10
   2.2 Relationship Analysis and Normalization..........................................15

CHAPTER 3: Implementing the Database in Microsoft SQL Server 2000......21
   3.1 Creating the Database in Microsoft SQL Server 2000......................21
       3.1.1 Creating the Database in the Enterprise Manager....................21
       3.1.2 Creating the Tables of the Database........................................22
       3.1.3 Data Types................................................................................22
       3.1.4 Defining the Primary Key..........................................................23
   3.2 Creating the Database Diagram in Microsoft SQL Server 2000........25
       3.2.1 Creating the Database Diagram in the Enterprise Manager.....25
       3.2.2 Creating the Table Relationships in the Enterprise Manager....25
       3.2.3 Creating the HOSPITAL Database Diagram............................26
The dissertation project titled: “Business Logic Layer and Data Logic Layer of the Medi Gen HOSPITAL application” is the one of the two parts of a bigger project called “Mobile & Wireless Multimedia for Hospital Information Systems: A 3-tier solution based on MS Pocket PC, .NET Framework, and SQL Server.”

The initial idea for designing and developing an innovative Hospital Information System was conceived considering the fact that the use of wireless technologies has been widely spread. An increasing number of professionals and individuals are using mobile devices to accomplish their tasks on daily basis. Such a device is the Personal Digital Assistant mobile device (PDA).

The name Personal Digital Assistant was first coined by the Apple CEO John Sculley in a speech at the Winter Consumer Show in January 1992. A PDA was described as a small handheld computer used to write notes, track appointments and otherwise, keep your life in order. PDAs provide all the functionality of a cheap pad of paper at hundreds of times the cost and with far less capacity. For early PDAs, that was probably a fair description. This innovative device used pen-input and handwriting recognition, Personal Information Management (PIM) features and communications capability.

Today, PDAs are considered to be very useful tools with powerful features. They consist of a capable CPU, a RAM for program and data storage and a persistent storage, such as a disk drive or RAM disk. PDAs belong to the known family of smart devices. As smart devices, they are capable of running programs and connecting to a computer network. They, also, offer the ability to access mobile Web content through a Wireless LAN. Compared to mobile phones, there are a lot of similarities, but PDAs are thought to be more capable. They have greater storage capacity and a larger screen for providing a richer user interface.
In order to evaluate the capabilities of this promising mobile device, the idea of the MEDI GEN hospital application came up. The cases that had to be covered were:

- The capability of a PDA to communicate through a wireless LAN with other application, to send and receive data and process the results.
- The potentials of a PDA to provide a satisfying Graphical User Interface, and
- In what extent a PDA can correspond to the high demands of a professional application environment.

Hence, the concept of developing a Hospital Information System for PDA users proved to be an ideal one. The prospect of doctors and nurses to use PDAs in order to access the medical data is an idea seems to be reasonable and attainable in the near future. Medical information, such as the patient's file, the prescriptions and other documents, had not been kept in an electronic form until today. Usually, the larger part of medical data is kept as manuscripts. In this application, an effort to completely automate all tasks was made.
CHAPTER 1: Introduction

1.1 General

“Medi Gen HOSPITAL” application was designed and developed for the dissertation project with title: “Business Logic Layer and Data Logic Layer of the Medi Gen HOSPITAL application”. It is an application developed with Microsoft Visual Studio .NET 2003 and Microsoft SQL Server 2000, in order to store all the medical data of the patients of the “Medi Gen” hospital and provide access to them.

1.2 System Usage Analysis

The hospital “Medi Gen” is a general surgery hospital that treats patients with heart diseases, pneumonic problems, vassal surgery or thoracic surgery needs and other relevant problems.

When a patient enters the hospital, the nurse on shift completes the “Patient’s Admission Form”, and his medical file is created. This form records patient’s personal information, such as the surname, name and the father’s name and other, as the number of the room which he occupies, and the name of the attendant doctor.

Then, the nurse asks the patient in order to obtain information about his personal and family medical history. After completing the appropriate forms, the medical file of the patient is registered.

In the meanwhile, the attendant doctor carries out the medical examination of the patient and indicates to the nurse the reason of admission to the hospital, in order to be written in the patient’s file.

When the medical examination is completed, the doctor writes some instructions in the patient’s file. These instructions relate to the medication of the patient and the courses of the medicines taken, the treatment to be followed by the patient, and his nutrition schedule. This information can be modified again by the doctor, according to the progress of the patient’s condition and his needs.

Then, the doctor decides and orders the medical tests that have to be done for the patient, by completing the form “Reference for Medical Test”. In this form, the
following are written: the name of the doctor who ordered the medical test, the medical test that has to be done, the patient’s personal data, and, also some instructions concerning the test, addressed to the doctor who will carry out the test.

After the reference is completed, the nurse undertakes the responsibility to make an appointment for the medical test to be done with the appropriate doctor and transfer the patient there.

After all the necessary medical tests are carried out, the patient is transferred back to his room. There he is monitored and examined on a daily basis by his attendant doctor.

The results of the medical tests completed are then written in the patient’s file. The attendant doctor checks over each result, decides the diagnosis for it, and writes it to the patient’s file. If the doctor decides that is necessary more medical tests to be done, a new reference is written.

Nurses are responsible to strictly follow the instructions and accomplish the prescriptions of the doctors. In addition, it is their duty to visit and monitor the patients, for which they are responsible, in order to check their temperature, pulses and blood pressure, every three hours. The results of each check up are written in the form “Patient’s Daily Progress”, in the patient’s file.

Finally, it has to be noticed that a nurse is not allowed to order a reference for a medical test to be done, or write or change the diagnosis for a test. Evenly, a doctor is not allowed to change the contents of a “Patient’s Daily Progress” form.

After the treatment of the patient is completed and he is released from the hospital, his medical file is stored, as it may be needed for future use.

1.3 System Aim

The aim of the system developed is the analysis of the electronic data processing demands of the “Medi Gen” hospital, in order to implement a complete information system for maintaining, storing and managing the medical file of the patient, which will be used from the doctors and nurses of the hospital.

It is assumed that a wireless LAN is established in the hospital. Doctors and nurses use PDAs for accessing the information system and using the services provided by the network. All information is stored in a database.

The reason for developing the “Medi Gen Hospital” application was not to construct an information system that is real, meaning that it can deal with the demands of a real hospital, but to construct a Hospital Information System that supports the basic operations that have to be carried out. That was accomplished, serving the endmost aim of developing the application, which is to explore and test the capabilities and the potentials of a new technology, a Pocket-PC based PDA, to perform certain tasks in cooperation with a database management system.
1.4 3-Tier Application

The hospital application that is described in this report is a 3-tier application. A 3-tier application is an application program that is organized into three major parts, each of which is distributed to a different place or places in a network. The three parts are:

- The workstation or presentation interface.
- The business logic.
- The database and programming related to managing it.

In a typical 3-tier application, the application user’s workstation contains the programming that provides the graphical user interface (GUI) and application-specific entry forms or interactive windows. The presentation layer of the “Medi Gen Hospital” application consists of windows forms especially designed for users with different tasks, demands and access rights.

Business logic is located on a local area network (LAN) server or other shared computer. The business logic acts as the server for client requests from workstations. In turn, it determines what data is needed (and where it is located) and acts as a client in relation to a third tier of programming that might be located on a mainframe computer. The business logic layer of the application is a server, which accepts the client requests and queries the hospital database, in order to satisfy them.

The third tier includes the database and a program to manage read and write access to it. The data logic layer of the application is a database used to store all the necessary medical information of the hospital.

While the organization of an application can be more complicated than this, the 3-tier view is a convenient way to think about the parts in a large-scale program.

The “Medi Gen HOSPITAL” application uses the client/server computing model. With three tiers or parts, two different programmers have developed this application; the one programmer developed the first tier (GUI and client program) and the other the second and third tier (server program and database). Because the programming for a tier can be changed or relocated without affecting the other tiers, the 3-tier model makes it easier for someone to continually evolve an application as new needs and opportunities arise.

In this report, the second and the third tier of the “Media Gen HOSPITAL” application are analytically described and it is explained how they have been developed. For more information on the implementation of the Presentation layer of the “Medi Gen Hospital” application please refer to “Presentation layer of the Medi Gen HOSPITAL application”, Maria Bika, Dissertation Project, Mobile Multimedia Laboratory, Athens University of Economics and Business, 2003.
1.5 System Description

As shown in the above figure, in “Medi Gen HOSPITAL” application, the database and the server are located on the same computer, and clients are all the PDA users that communicate with the server through the wireless LAN. The wireless communication between the client and the server is based on IEEE 802.11.
1.6 Tools Used

For the implementation of the second and the third tier of the application “Medi Gen HOSPITAL”, a database had to be designed and developed, in order to store all the necessary information and a server program to provide access to this information. To satisfy these demands, the following tools were used:

Visual Studio .NET 2003 was used to implement this application. Visual Studio provides various tools like editors, emulators, debuggers, build tools, and other useful tools, that are all available from one single place. This is the primary difference with other platforms used to develop wireless applications.

One the other hand, Visual C# was found to be an easy-to-learn programming language and the programming environment provided by Visual Studio made the development of the application easier.

Microsoft SQL Server 2000 is a powerful database management system, and it was used for implementing the database. It is a set of components that work together to meet the data storage and analysis needs of the largest Web sites and enterprise data processing systems. It is easy to install and use, and provides scalability, availability, and powerful security features. The use of the SQL Server, in order to fulfill the application’s storage needs will be shown analytically, step by step, during this report.

1.6.1 Creating a Program in Visual Studio .NET 2003

In Visual C# .NET, projects are generally grouped in solutions and projects, which are subdirectories of the solution directory.

The solution for the hospital application was created by selecting: File > New > Blank Solution and then the disc location and the name HOSPITAL SERVER for the solution. The project of this solution, named as Project 1, was created by selecting: File > Add Project > New Project and then project type template Empty Project, as there was no need for Visual C# .NET to generate a code.

Within Project 1, by selecting Project > Add New Item, the C# source code files were created. In the Categories list, Local Project Items is chosen, and in the Templates section, the option Code File, as there was no need for Visual C# .NET to generate a code. The source code of the hospital application consists of three source code files: multiserver.cs, worker_thread.cs, db_management.cs.
1.7 Basic Definitions

What is a database?

- A database is an organized collection of information.
- Each record in a database is composed of the important elements of information for a particular item.
- Each record is composed of a set of fields, which contain the individual elements of information.

A database typically has two main parts: first, the files holding the physical database and second, the database management system (DBMS) software that applications use to access data. The DBMS is responsible for enforcing the database structure.

Relational Database

Although there are different ways to organize data in a database, relational databases are one of the most effective. Relational database systems are an application of mathematical set theory to the problem of effectively organizing data. In a relational database, data is collected into tables (called relations in relational theory).

A table represents some class of objects that are important to an organization. Each table is built of columns and rows (called attributes and tuples in relational theory). Each column represents some attribute of the object represented by the table. Each row represents an instance of the object represented by the table.

Relational database theory defines a process called normalization, which ensures that the set of tables defined will organize the data effectively. Normalizing a logical database design involves using formal methods to separate the data into multiple, related tables. Reasonable normalization often improves performance.

Table Relationships and Keys

In a relational database, relationships enable to prevent redundant data.

- **PRIMARY KEY**
  Is a column or set of columns that uniquely identifies all the rows in a table. Primary keys do not allow null values. No two rows can have the same primary key value.

- **FOREIGN KEY**
  Is the column or combination of columns whose values match the primary key or unique key in the same or another table. A foreign key is used to establish and enforce a link between the data in two tables.
The types of table relationships are:

**One-to-Many Relationships:**
In this type of relationship, a row in table A can have many matching rows in table B, but a row in table B can have only one matching row in table A.

**Many-to-Many Relationships:**
In this type of relationship, a row in table A can have many matching rows in table B, and vice versa. Such a relationship can be created by defining a third table, called a junction table, whose primary key consists of the foreign keys from both table A and table B.

**One-to-One Relationships:**
In this type of relationship, a row in table A can have no more than one matching row in table B, and vice versa. A one-to-one relationship is created if both of the related columns are primary keys or have unique constraints.
The logical design of the relational database has to be carried out, before implementing the database. The actions that have to take place are: the entity analysis, the relationship analysis and the normalization of the database.

The first step is to make the analysis of the entities that exist in the environment of the application. The main target is to find the appropriate entities in order to cover all the functionality demands of the hospital. Each entity represents an object of the real world and their attributes describe the characteristics of the entity.

The next step is to define the appropriate attributes of each entity in order to sufficiently describe it. In addition, to achieve the reduction of the redundancy in the attributes chosen, which means the recognition and elimination of the repeated attributes. The most common way to accomplish that, is the disintegration of the entities, meaning the extraction of the repeated attributes and their placement in other entities.

The basic entities are three: **patient**, **doctor**, **nurse**. It can easily become understood that the entity **patient** represents every patient in the hospital, the entity **doctor** every doctor, and the entity **nurse** every nurse, and are used to contain their personal data.

The basic attributes of each entity are shown in the table below:

<table>
<thead>
<tr>
<th>Entity</th>
<th>Attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td>patient</td>
<td>patient_code, surname, name, fathers_name, insertion_date</td>
</tr>
<tr>
<td>doctor</td>
<td>doctor_code, surname, name, specialization</td>
</tr>
<tr>
<td>nurse</td>
<td>nurse_code, surname, name, position</td>
</tr>
</tbody>
</table>

**Table 1: Basic initial entities and basic attributes.**
Of course, there are more entities in the hospital, which are: medical_info, medical_test, and room. The entity medical_info represents the file of a patient, which contains all the necessary medical information. The entity medical_test represents every kind of medical test that can be done in the hospital. The entity room represents every room in the hospital.

The basic attributes of each entity are shown in the table below:

<table>
<thead>
<tr>
<th>Entity</th>
<th>Attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td>medical_info</td>
<td>birthday, blood_type, reason_of_insertion, medical history, daily_progress</td>
</tr>
<tr>
<td>medical_test</td>
<td>test_code, kind, reference, result</td>
</tr>
<tr>
<td>room</td>
<td>room, floor, free</td>
</tr>
</tbody>
</table>

Table 2: Initial entities and basic attributes.

The basic assumptions made, based on the functionality demands of the hospital and its main entities are:

- A doctor is responsible for many patients, but only one doctor who is the patient’s attendant doctor treats one patient.
- Only one patient occupies each room, and the nurses who are responsible for a room, are responsible for the occupant patient.
- A nurse is responsible for many patients, and many nurses, who are responsible for his room, treat one patient.
- The medical information of each patient is related to only to this patient.
- A medical test can be carried out for many patients, and a patient can have many medical tests done.
- A doctor can order many medical tests to be carried out for his patients, and many doctors can order a medical test.
- All medical tests can be ordered by all doctors in the hospital.

So, the relationships between the entities are shaped in the following way:
The following diagram is called entity–relationship diagram (E-R) and shows all the entities of the hospital environment and the relationships between them.

In the E-R the one-to-one relationships are denoted by:

In the E-R the one-to-many relationships are denoted by:

In the E-R the many-to-many relationships are denoted by:

\[ \text{doctor} \rightarrow \text{nurse} \rightarrow \text{patient} \rightarrow \text{medical_info} \]
\[ \text{room} \rightarrow \text{patient} \rightarrow \text{medical_test} \]

**Figure 2.1: Initial E-R diagram.**

In the above E-R diagram two circles are created, so there are alternative paths to access the entities. That means that there are unnecessary (redundant) edges in the diagram. One of the most important problems faced during this process is the identification and reduction of these edges.

In the first case, the edge that can be deleted is either nurse-patient, or patient-room, or room-nurse. Since there is one patient in each room (one-to-one relationship), the edge that should be deleted is the patient-room. Then by adding the attribute room in the entity patient, it is known which is the number of the room the patient is in, and it is not a repeated attribute (one room for each patient), so it will not need further modification.

In the second case, the edge that can be deleted is either doctor-patient, or patient-medical_test, or medical_test-doctor. It is chosen to delete the edge medical_test-doctor. But, then the attribute patient_code must be added in the entity medical_test, so that it is known for which patient a medical test is ordered to
be done. It is a repeated attribute (all patients can do all tests) and it will need further modification. This way the relation between the entities doctor and medical_test is established through a different path, through the entity patient.

The following diagram is the new modified E-R:

![Modified E-R diagram](image)

**Figure 2.2: Modified E-R diagram.**

The next step is to look for sub-entities that may be embedded in the initial entities, and, of course, look for repeated attributes.

In the entity medical_test there are the attributes reference and result. Before a medical test is carried out for a patient, a reference for it to be done is written by his doctor. After the medical test is done there is a result for it. One test can be done for one patient many times. This means that, if the entity medical_test becomes a table of the database, in its current form, where each record describes a medical test, there will be duplicate values.

That is why, two new entities, reference and result, have to be created, where every record is unique. There could have been created only one of them to contain both of them and their attributes, as the relationship between them is one-to-one. The only reason to make this split, is semantic. To separate their meaning and reason of existence, as they represent two different objects in the real environment of the hospital.

The entity reference represents the reference that can be written by a doctor for a medical test to be carried out for a patient. The entity result represents the result of the medical test that was carried out for a patient.
The following assumptions are made:

- The patient’s doctor can order one medical test to be carried out for a patient, by writing a reference, so a doctor can write many references for a patient, and one reference is related only to one patient and one doctor.
- Only the attendant doctor of the patient can write a reference for him.
- The result of a medical test is related to only one reference.

So, the relationships between the entities are shaped in the following way:

<table>
<thead>
<tr>
<th>Entity</th>
<th>Relationship</th>
<th>Entity</th>
</tr>
</thead>
<tbody>
<tr>
<td>patient</td>
<td>one-to-many</td>
<td>doctor</td>
</tr>
<tr>
<td>patient</td>
<td>many-to-many</td>
<td>nurse</td>
</tr>
<tr>
<td>patient</td>
<td>one-to-one</td>
<td>medical_info</td>
</tr>
<tr>
<td>patient</td>
<td>one-to-many</td>
<td>reference</td>
</tr>
<tr>
<td>doctor</td>
<td>one-to-many</td>
<td>reference</td>
</tr>
<tr>
<td>nurse</td>
<td>many-to-many</td>
<td>reference</td>
</tr>
<tr>
<td>medical_test</td>
<td>one-to-many</td>
<td>result</td>
</tr>
<tr>
<td>reference</td>
<td>one-to-one</td>
<td></td>
</tr>
</tbody>
</table>

The following diagram is the new E-R:

![E-R Diagram](image-url)

**Figure 2.3: New E-R diagram.**
In the above E-R diagram one circle is created. The redundant edge has to be deleted from the diagram.

The edge that can be deleted is either doctor-reference, or reference-patient, or patient-doctor. Since there is only one doctor for each patient, who is his attendant doctor, if it is known for which patient the reference is written, it is, also, known which doctor ordered it. So, the edge that should be deleted is the doctor-reference one. Then by adding the attribute patient_code in the entity reference, it is known for which patient the reference is written, and it is not a repeated attribute, so it will not need further modification. This way the relation between the entities doctor and reference is established through a different path, through the entity patient.

In addition, the relation between the entities patient and medical_test is established through the entity reference. That is why, the attribute patient_code does not need to exist in both reference and medical_test entities. It is chosen to be only an attribute for the reference entity, and then, by adding, also, the attribute test_code in the reference entity, it is known for which patient a reference is written and what medical test is ordered to be done for him.

The following diagram is the new modified E-R:

![New modified E-R diagram](image-url)

**Figure 2.4: New modified E-R diagram.**
2.2 Relationship Analysis and Normalization

The next step is the relationship analysis and the normalization.

From the entity analysis resulted all the entities that will be used in the application, and that will become the tables in the database, in order to store the necessary information of the hospital.

Now, the relationships between them have to be established, in a way that reflects the functionality needs and demands of the hospital, and, also, the relationships that derive from the new entities created have to be defined.

The next action is to define the primary attribute for each entity. The primary attribute uniquely identifies each record of an entity. As a result, each record is inserted in the database only once and can be referenced by its primary attribute. When each entity is inserted in the database as a table, its primary attribute will become the primary key, which uniquely identifies the table.

In order to accomplish the definition of primary attributes, each record of an entity is given a code. For example, each patient is uniquely identified by his patient_code. The medical info of a patient is uniquely identified by the patient_code. The room number uniquely identifies a room.

The other attributes are chosen in order to sufficiently describe the object that the entity represents, and, additionally, to satisfy the purposes and demands of the application. Each attribute has a predefined data type and length, which is chosen, according to the object that it represents and the needs of the application.

But, first, there is a task that needs to be done. One of the goals of normalization is to eliminate every many-to-many relationship. As shown in the above E-R, there are two many-to-many relationships:

<table>
<thead>
<tr>
<th>Entity</th>
<th>Relationship</th>
<th>Entity</th>
</tr>
</thead>
<tbody>
<tr>
<td>patient</td>
<td>many-to-many</td>
<td>nurse</td>
</tr>
<tr>
<td>nurse</td>
<td>many-to-many</td>
<td>room</td>
</tr>
</tbody>
</table>

The most practical way of dealing with a many-to-many relationship is the creation of an additional entity, which contains attributes of both entities involved in the relationship. The many-to-many relationship is established through the new entity. So, two new entities are created: patient_nurse in the first case and nurse_room in the second case.

The entity patient_nurse is given as primary key the complex attribute: patient_code, nurse_code. This modification allows inserting unique records in the entity, independently of the involved entities, and implementing this way the many-to-many relationship. In the same way, the entity nurse_room is given as primary key the complex attribute: nurse_code, room.
The new relationships that are created are:

<table>
<thead>
<tr>
<th>Entity</th>
<th>Relationship</th>
<th>Entity</th>
</tr>
</thead>
<tbody>
<tr>
<td>patient</td>
<td>one-to-many</td>
<td>patient_nurse</td>
</tr>
<tr>
<td>nurse</td>
<td>one-to-many</td>
<td>patient_nurse</td>
</tr>
<tr>
<td>nurse</td>
<td>one-to-many</td>
<td>nurse_room</td>
</tr>
<tr>
<td>room</td>
<td>one-to-many</td>
<td>nurse_room</td>
</tr>
</tbody>
</table>

The final form of the E-R diagram is shown below:

![Final E-R diagram](image)

**Figure 2.5: Final E-R diagram.**

The primary keys of the rest of the entities are chosen in the following way:

The attribute doctor_code is added in the entity **patient**, and this way it is shown that a patient has only one attendant doctor, and the one-to-many patient – doctor relationship is established. So, doctor_code is considered to be a foreign key. The attribute room has, also, been added, for the reason described before, so, it is a foreign key in the entity **patient**. Finally, the attribute patient_code is defined as primary key in the entity **patient**.
In the entity **doctor** the attribute doctor_code is defined as primary key, and in the entity **nurse** the attribute nurse_code is defined as primary key. So, each record of each entity is inserted with a code, which cannot have duplicate values, and uniquely identifies it.

In the entity **room** the attribute room is defined as primary key, and represents the room number.

In the entity **medical_info** the attribute patient_code is defined as primary key. This way the one-to-one patient – medical_info relationship is established. Each record of the table **patient**, which is unique and contains the personal data of one specific patient, is related to only one record of the table **medical_info**, which has the same patient_code and contains all the necessary medical information for the patient.

In the entity **medical_test** the attribute test_code is defined as primary key. So, each kind of test that can be done in the hospital is inserted, with a code, only once. In the entity **reference** the attribute reference_code is defined as primary key. The attributes patient_code and test_code have been added, for the reasons described before, so, the one-to-many patient – reference relationship and the one-to-many medical_test – reference relationship are established. So, patient_code and test_code are considered to be foreign keys in the entity **reference**.

Finally, in the entity **result** the attribute result_code is defined as primary key. The relationship between the entities **reference** and **result** is one-to-one. So, each record of the table **reference** is related to only one record of the table **result**. In order to define which this record is, it is assumed in the hospital database, that the reference_code of a record and the result_code of the record related to it have the same value. So, when a reference is written from a doctor for a test to be done, a new record is created and is given a unique reference_code. At the same time, a new record for the result is, also, created and is given a result_code with the same value. When the test is carried out for the patient and the results are ready, the record with the appropriate result_code is updated with the necessary information.

The rest of the attributes are chosen in appropriate way to serve the needs of the hospital application:
(Also the attributes that were chosen during the process are described.)

**patient**

surname, name, fathers_name : patient’s personal data 
doctor_code : the code of the patient’s attendant doctor  
room : the number of the room he occupies 
insertion_date : the date the patient is inserted in the hospital

**doctor**

surname, name : doctor’s personal data  
password : used to authenticate the doctor 
specialization : doctor’s specialization (for example cardiologist)
Business Logic and Data Logic Layer of “Medi Gen HOSPITAL” application

nurse

surname, name: nurse’s personal data
password: used to authenticate the nurse
position_: nurse’s position in the hospital (for example supervisor)

medical_info

birthday : patient’s date of birth
blood_type : blood group
reason_of_insertion : the reason for which the patient was inserted in the hospital
history: medical history (for example diseases he has been through)
daily_progress : the progress of the patient (blood pressure, temperature), which is written by the nurse several times a day after he examines him

room

floor: shows on what floor the room is
free: shows either the room is available or not

medical_test

kind: shows what kind of test it is (for example cardiogram)

reference

patient_code : the patient for which the reference is written
test_code : the medical test that is ordered to be carried out for the patient
reference_date : the date when the reference is written by the doctor
notes: notes of the doctor for the medical test (for example instructions)

result

result_date : the date when the result is ready
diagnosis: the diagnosis that is given by the doctor for a medical test that was carried out, after he has viewed the results.
image_result: the result of a medical test in the form of an image
doc_result: the result of a medical test in the form of a document
kind: is used to define either the result is in image form or a document

The attributes of each entity are shown in the table below:
(# shows the primary attribute)
<table>
<thead>
<tr>
<th>Entity</th>
<th>Attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td>patient</td>
<td>patient_code#, surname, name, fathers_name, doctor_code, room, insertion_date</td>
</tr>
<tr>
<td>doctor</td>
<td>doctor_code#, password, surname, name, specialization</td>
</tr>
<tr>
<td>nurse</td>
<td>nurse_code#, password, surname, name, position_</td>
</tr>
<tr>
<td>patient_nurse</td>
<td>patient_code#, nurse_code#</td>
</tr>
<tr>
<td>medical_info</td>
<td>patient_code#, birthday, blood_type, reason_of_insertion, history, daily_progress</td>
</tr>
<tr>
<td>room</td>
<td>room#, floor, free</td>
</tr>
<tr>
<td>nurse_room</td>
<td>nurse_code#, room#</td>
</tr>
<tr>
<td>medical_test</td>
<td>test_code#, kind, doctor</td>
</tr>
<tr>
<td>reference</td>
<td>reference_code#, patient_code, test_code, reference_date, notes</td>
</tr>
<tr>
<td>result</td>
<td>result_code#, result_date, diagnosis, image_result, doc_result, kind</td>
</tr>
</tbody>
</table>

Table 3: Final entities and attributes.
3.1 Creating the Database in Microsoft SQL Server 2000

3.1.1 Creating the Database in the Enterprise Manager

The logical design of the database has been completed, so the next level is the implementation of the database in the Microsoft SQL Server 2000.

A database in Microsoft SQL Server 2000 consists of a collection of tables that contain data and other objects, such as views, indexes, stored procedures, and triggers, defined to support activities performed with the data.

The first step is to create the database. To create a database the name of the database, its owner (the user who creates the database), its size, and the files and filegroups used to store it, have to be determined.

SQL Server Enterprise Manager is the primary administrative tool for SQL Server and provides a user interface that allows users to create and administer all SQL Server databases, objects, logins, users, and permissions in each registered server.

Using the Enterprise Manager made the creation of the database. The steps followed are:

1. Select the LOCAL server from the SQL server group.
2. Select Databases and create the new Database.
3. Name it HOSPITAL.

The primary database and transaction log files of the HOSPITAL database created have the name HOSPITAL (HOSPITAL_Data.mdf, HOSPITAL_Log.ldf). The defaults initial sizes of the database and transaction log files are chosen. The primary file contains the system tables for the database.
So, this way the database HOSPITAL is created. HOSPITAL can be found in Databases, where all the databases that exist in the SQL Server appear. By selecting HOSPITAL, all the objects of which it consists can be viewed.

### 3.1.2 Creating the Tables of the Database

The next step is to create the tables of the database, as they were defined during the logical design of the database. To create a table using the Enterprise Manager the steps followed are:

1. Select HOSPITAL and in Tables create the new table.
2. Give the name for each column, the data type and the length of the data it will contain, and, also define if null values are allowed.
3. Save it with an appropriate name.

Each table and its columns were described during the logical analysis of the database. Every entity represents a table of the database HOSPITAL, and its attributes represent the columns. Now following the above process creates each one of them.

When selecting Tables all the tables that exist in the database appear. For each table, the existing data can be viewed, changed or deleted, and new data can be inserted. In addition, any table that has already been created can be redesigned, by selecting the table. The existing options for columns can be changed or a new column can be inserted, according to the new needs.

### 3.1.3 Data Types

Data types define the format in which data is stored. The data types used in the database HOSPITAL are:

- **int**: Integer (whole number) data from $-2^{31}$ (-2,147,483,648) through $2^{31} - 1$ (2,147,483,647).
- **varchar**: Variable-length non-Unicode data with a maximum of 8,000 characters.
- **varbinary**: Variable-length binary data with a maximum length of 8,000 bytes.
- **text**: Variable-length non-Unicode data with a maximum length of $2^{31} - 1$ (2,147,483,647) characters.
- **image**: Variable-length binary data with a maximum length of $2^{31} - 1$ (2,147,483,647) bytes.
datetime: Date and time data from January 1, 1753, through December 31, 9999, with an accuracy of three-hundredths of a second, or 3.33 milliseconds.

The appropriate data type for each column was chosen according to what this column represents in the real world.

- **Varchar was chosen for all personal data**, as it represents strings of characters and because it is flexible not having fixed length. The default length was chosen for the data type, which is 50, big enough to store this kind of data.

- For notes in table reference and reason_of_insertion in table medical_info, also **varchar** was selected, with length 500 which is expected to fit data. It was preferred over text data type, as only a few lines are enough for this kind of data.

- All codes, except for doctor_code and nurse_code, are chosen to be **int**, so that the application will automatically increase them every time there is a new insertion in a table.

- **Varchar** was chosen for doctor_code and nurse_code. In the hospital application, to separate doctor from nurse, two characters are used in front of their last name: "d_" for doctor and "n_" for nurse. For example, d_Parker is the username for the doctor Mr. Parker. They, also, represent the username to authenticate a doctor or a nurse.

- **Datetime** was used for every column that represent date.

- The data type used for daily_progress in table medical_info is **varbinary**. This column is used to store an xml file in binary form, which contains the patient's daily progress. This file is not expected to exceed 8KB, so varbinary was used. This type does not have fixed length, so it is flexible to store the file, which increasingly grows.

- The data type used for history in table medical_info, diagnosis in table result, and doc_result in table result is **text**. It is long enough to fit this kind of data. These columns will contain long strings of characters, which are not read-only, but can be viewed, edited and saved, after retrieved from database, so this data type is really flexible and appropriate for this use.

- The data type used for image_result in table result is **image**. The reason why is that this column contains an actual image in .jpg, .bmp or .gif format, which is the result of a test. In database is stored in binary form, as a string of bits, and after retrieved from database can only be viewed from the client application.

- The data type used for the column free in table room is **int**. The value 1 indicates an available room and the value 0 indicates an occupied room.

- The data type used for the column kind in table result is int. The value 1 indicates that the result inserted is an image and the value 2 that it is a document.
At this point some important remarks on image and text data types have to be made. Individual text and image values can be a maximum of 2-GB, which is too long to store in a single data row. So, in Microsoft SQL Server 2000, small text and image values can be stored directly in the row, but values too large to fit in the row are stored in a collection of pages. This can be done when an option called text in row is ON.

In the database HOSPITAL of the application this option is chosen to be OFF, which is the default value. So, for each text or image value all that is stored in the data row is a 16-byte pointer, and all text and image values are stored in a collection of pages, separate from the pages holding the data for the other columns of the row. For each row, this pointer points to the location of the text or image data. A row containing multiple text or image columns has one pointer for each text or image column.

### 3.1.4 Defining the Primary Key

The primary key for each table needs to be defined. This definition is important to enforce uniqueness for values entered in specified columns that do not allow nulls. On the other hand, by defining a primary key for a table in the database, that table can be related to other tables, thus reducing the need for redundant data. A table can have only one primary key.

To define the primary key, while creating or redesigning a table, the following actions take place:

1. Select the column that is chosen to be the primary key. The primary key may be consisted of multiple columns (in that case CTRL key have to be held down while selecting).
2. Choose option Set Primary Key.

So, this way the primary keys selected for each table during the logical design of the database, are defined.

A primary key column is identified by a primary key symbol in its row selector. If a primary key consists of more than one column, duplicate values are allowed in one column, but each combination of values from all the columns in the primary key must be unique.

In case that the primary key needs to be redefined, any relationships to the existing primary key must be deleted before the new primary key can be created. The primary keys can, also, be defined, while designing the database diagram, by following the same process.
3.2 Creating the Database Diagram in Microsoft SQL Server 2000

The next step is to create the relationships between the tables of the database, as they were defined during the logical analysis of the database. The relationships between the tables can be created in a database diagram to show how the columns in one table are linked to columns in another table. In a relational database, relationships enable to prevent redundant data.

3.2.1 Creating the Database Diagram in the Enterprise Manager

First the database diagram has to be created, following the steps below:

1. Select HOSPITAL and in Diagrams create the new diagram.
2. Add all the user tables of the database HOSPITAL in the diagram.
3. Save it with the name HOSPITAL DIAGRAM.

All tables of the database HOSPITAL appear in the diagram. In the database diagram the tables can be redesigned or a new table can be created. For example, a column can be inserted or deleted, the primary key can be set or redefined, the data types can be modified, null values can be allowed or not.

When selecting a table, a number of options appear. By choosing properties full information for a table can be viewed. Also, the view of the table in the diagram can be defined.

3.2.1 Creating the Table Relationships in the Enterprise Manager

The steps to create a relationship between two tables in the database diagram are:

1. Select one of the two tables involved and in Relationships create a new relationship.
2. Select the table that will be on the primary-key side of the relationship and the columns contributing to the table's primary key. Then, for each of these columns, enter the corresponding foreign-key column of the foreign-key table.
3. The name suggested for the relationship can either be retained or changed.
The following rules must strictly be followed:

- You cannot enter a value in the foreign key column of the related table if that value does not exist in the primary key of the related table. However, you can enter a null in the foreign key column.
- You cannot delete a row from a primary key table if rows matching it exist in a related table.
- You cannot change a primary key value in the primary key table if that row has related rows.

Every enforced relationship appears in the database diagram as a solid line.

### 3.2.3 Creating the HOSPITAL Database Diagram

#### One-to-Many Relationships

- **Tables related**: patient, doctor
  - **Relationship Name**: FK_patient_doctor
  - **Primary Key Table**: doctor (table’s primary key doctor_code)
  - **Foreign Key Table**: patient (table’s foreign key doctor_code)

- **Tables related**: patient, reference
  - **Relationship Name**: FK_reference_patient
  - **Primary Key Table**: patient (table’s primary key patient_code)
  - **Foreign Key Table**: reference (table’s foreign key patient_code)

- **Tables related**: patient, patient_nurse
  - **Relationship Name**: FK_nurse_patient_patient
  - **Primary Key Table**: patient (table’s primary key patient_code)
  - **Foreign Key Table**: patient_nurse (table’s foreign key patient_code)

- **Tables related**: nurse, patient_nurse
  - **Relationship Name**: FK_patient_nurse_nurse
  - **Primary Key Table**: nurse (table’s primary key nurse_code)
  - **Foreign Key Table**: patient_nurse (table’s foreign key nurse_code)

- **Tables related**: nurse, nurse_room
  - **Relationship Name**: FK_nurse_room_nurse
  - **Primary Key Table**: nurse (table’s primary key nurse_code)
  - **Foreign Key Table**: nurse_room (table’s foreign key nurse_code)
Business Logic and Data Logic Layer of “Medi Gen HOSPITAL” application

<table>
<thead>
<tr>
<th>Tables related</th>
<th>room, nurse_room</th>
</tr>
</thead>
<tbody>
<tr>
<td>Relationship Name</td>
<td>FK_nurse_room_room</td>
</tr>
<tr>
<td>Primary Key Table</td>
<td>room (table’s primary key room)</td>
</tr>
<tr>
<td>Foreign Key Table</td>
<td>nurse_room (table’s foreign key room)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Tables related</th>
<th>reference, medical_test</th>
</tr>
</thead>
<tbody>
<tr>
<td>Relationship Name</td>
<td>FK_reference_medical_test</td>
</tr>
<tr>
<td>Primary Key Table</td>
<td>medical_test (table’s primary key test_code)</td>
</tr>
<tr>
<td>Foreign Key Table</td>
<td>reference (table’s foreign key test_code)</td>
</tr>
</tbody>
</table>

On the diagram, the primary key side of a one-to-many relationship is denoted by a key symbol. The foreign key side of a relationship is denoted by an infinity symbol.

The tables **patient_nurse** and **nurse_room** used to implement the many-to-many relationships, are called **junction tables**. In a junction table the primary key consists of the foreign keys from both tables involved.

**One-to-One Relationships**

<table>
<thead>
<tr>
<th>Tables related</th>
<th>patient, medical_info</th>
</tr>
</thead>
<tbody>
<tr>
<td>Relationship Name</td>
<td>FK_medical_info_patient</td>
</tr>
<tr>
<td>Primary Key Table</td>
<td>patient (table’s primary key patient_code)</td>
</tr>
<tr>
<td>Foreign Key Table</td>
<td>medical_info (table’s foreign key patient_code)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Tables related</th>
<th>reference, result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Relationship Name</td>
<td>FK_result_reference</td>
</tr>
<tr>
<td>Primary Key Table</td>
<td>reference (table’s primary key reference_code)</td>
</tr>
<tr>
<td>Foreign Key Table</td>
<td>result (table’s foreign key result_code)</td>
</tr>
</tbody>
</table>

In one-to-one relationships, the table that initiated the relationship determines the primary key side.

The primary key side of a one-to-one relationship is denoted by a key symbol. The foreign key side is, also, denoted by a key symbol.
Figure 3.1: View of HOSPITAL DIAGRAM in the Enterprise Manager
CHAPTER 4: Accessing and Changing Relational Data in Microsoft SQL Server 2000

4.1 Introduction

The primary purpose of a database is to store data and then make that data available to authorized applications and users, which can utilize and process this data. They are able to access and retrieve existing data, change and update existing data, add and insert new data and, also, delete existing data.

In Microsoft SQL Server, accessing and changing data is achieved by using an application to send to SQL Server requests for data retrieval and modification. This kind of applications can access SQL Server by using the following two components:

- Transact-SQL statements, which are commands sent to SQL Server. Transact-SQL statements are built using the SQL language defined in the Transact-SQL Reference.
- Database application programming interfaces (APIs), which send commands to SQL Server and retrieve the results of these commands. The APIs can be general-purpose database APIs such as ADO, OLE DB or ODBC. They are used to execute Transact-SQL statements that work with SQL Server data in the form of tabular result sets. They can also be APIs designed specifically to use special features in SQL Server.

4.2 Using Transact-SQL

4.2.1 General

What is Transact-SQL?

SQL stands for Structured Query Language. SQL is used to communicate with a database. According to ANSI (American National Standards Institute), it is the standard language for relational database management systems.

When developing an application with Microsoft SQL Server 2000, the Transact-SQL programming language is the primary programming interface between the application and the SQL Server database.
Transact-SQL is central to the use of Microsoft SQL Server. All applications that communicate with SQL Server achieve that by sending Transact-SQL statements to the SQL server, regardless of the application's user interface. Most of the actions are implemented using the four basic Transact-SQL statements which are: SELECT to retrieve existing data, UPDATE to change existing data, INSERT to add new data rows, and DELETE to remove rows that are no longer needed.

### 4.2.2 Transact-SQL statements

In the hospital application the Transact-SQL statements used are:

(All the code samples, written in italics, are parts of the application’s source code.)

- **SELECT**

  The select statement is used to query the database and retrieve selected data that match the criteria specified.

  Example:

  ```sql
  SELECT doctor_code, surname, name, specialization   -- select all doctors' personal data
  FROM doctor
  ORDER BY surname                                             -- in alphabetic order by surname
  ```

- **INSERT**

  The insert statement is used to insert or add a row of data into the table.

  Example:

  ```sql
  -- insert patient's personal data
  INSERT INTO patient
  (patient_code, surname, name, fathers_name, room, doctor_code, insertion_date)
  VALUES(@pc, @sn, @n, @fn, @r, @dc, @ind)
  ```

- **UPDATE**

  The update statement is used to update or change records that match specified criteria.

  Example:

  ```sql
  UPDATE room -- make room with number r unavailable
  ```
SET free = 0
WHERE room = @r

• **DELETE**

The delete statement is used to delete records or rows from the table.

**Example:**

```
DELETE FROM medical_info   -- delete from the table medical_info the record
WHERE patient_code = @pc   -- of the patient with code pc
```

• **Subqueries**

Subqueries can be nested in UPDATE, DELETE, and INSERT statements, as well as in SELECT statements. An important kind of subquery is the subquery INSERT…SELECT, as it lets more than one row be inserted at one time, as shown in the example below.

**Example:**

```
-- a row for each nurse who is responsible for the patient’s room is inserted
-- in the table patient_nurse
INSERT INTO patient_nurse
    SELECT patient.patient_code, nurse_room.nurse_code
    FROM patient, nurse_room
    WHERE patient.patient_code = @pc  AND nurse_room.room = @r
```

An important remark is that a subquery introduced with an **unmodified comparison operator** (a comparison operator not followed by ANY or ALL) must return a single value or else it fails. In case of more than one values, the subquery must be introduced with IN, which determines if a given value matches any value in a subquery or a list.

**Example:**

```
-- all the results relevant to the patient with code pc will be deleted
DELETE FROM result
WHERE result_code IN
    (SELECT reference_code FROM reference WHERE patient_code = @pc)
```

• **Aggregate Functions**

Aggregate functions are used to compute against a column of numeric data, which is returned from a SELECT statement.

*In the application the functions used are:*

1. **MAX**, which returns the largest value in a given column.
2. **COUNT**, which returns the total number of values in a given column

These statements will return a single result.
Example:

-- if there are records of patients in the table  
IF (SELECT COUNT(patient_code) FROM patient) <> 0  
BEGIN  
    SELECT MAX(patient_code)  
    FROM patient  
END  
ELSE  
BEGIN  
    SELECT COUNT(patient_code) FROM patient  
END

• Table Joins

Table Joins allow linking data from two or more tables together into a single query result, from one single SELECT statement. It is one of the most beneficial features of SQL and relational database systems.

Example:

-- finds all the results that are ready for the medical tests that have been  
-- carried out for the patient with code pc  
SELECT result.result_code, medical_test.test_code, medical_test.kind,  
result.result_date  
FROM medical_test, reference, result  
WHERE reference.patient_code = @pc  
    AND reference.test_code = medical_test.test_code  
    AND reference.reference_code = result.result_code  
    AND result.result_date IS NOT NULL

4.2.3 SQL Stored Procedures

There are two methods provided for storing and executing the Transact-SQL programs:

- Either store the programs locally and then create applications that send the commands to SQL Server and process the results.
- Or, store the programs as stored procedures in SQL Server and create applications that execute the stored procedures and process the results.

Stored Procedures

A stored procedure in SQL Server is a precompiled group of Transact-SQL statements, which is processed as a unit. As all procedures in other programming languages, they accept input/output parameters, return values, contain SQL statements and control-of-flow statements, and can call other procedures.

A SQL Server stored procedure is created with the Transact-SQL CREATE PROCEDURE statement, which contains the procedure’s name and parameters and the body of the procedure.
Example:

```sql
CREATE PROCEDURE login_check
@uid varchar(50) AS
IF (@uid LIKE 'd%') BEGIN
SELECT password
FROM doctor
WHERE doctor_code LIKE @uid
END
ELSE BEGIN
SELECT password
FROM nurse
WHERE nurse_code LIKE @uid
END
```

The data returned from a stored procedure can be in one of the following forms:

- A result set for each SELECT statement contained in the stored procedure.
- Output parameters, which can return either data or a cursor variable.
- Return codes, which are always an integer value.

**Creating a stored procedure in Microsoft SQL Server 2000**

There are two ways to create a stored procedure:

1. Either connect to SQL Server using the SQL Query Analyzer, select the appropriate database, write the code of the stored procedure and execute it.
2. Or, in Enterprise Manage in Stored procedures, create the new stored procedure.

**Stored procedures are stored within the database and can be executed with one call from an application.**

**A user or an application can execute a stored procedure only if he has been given permission to it.** Permissions can be managed by selecting the stored procedure and ticking the box EXEC for the users that are allowed to execute it.

The user only needs to be given permission to the stored procedure; it is not necessary to be given permissions to SELECT, INSERT, UPDATE, or DELETE statements, even if the procedure contains these statements. So, stored procedures can act as a security mechanism; they help giving a user as fewer permissions as possible.
Advantages of using stored procedures

In the hospital application mostly stored procedures were used. They were preferred for many reasons. They can increase the security level for the database, as described above. On the other hand, they are part of the database, so they can easily be modified, without affecting the source code of the application. In addition, the source code is smaller and less complex, as the code of the procedure is not written in it.

A series of SQL statements can be execute in a single stored procedure, and, also, other stored procedures can be referenced from within the stored procedure, which can simplify a series of complex statements.

Finally, the stored procedure is compiled on the SQL server when it is created, so it executes faster than individual SQL statements. The statements and conditional logic are written into a stored procedure, they become part of a single execution plan on the SQL server, so all of the work is done on the server. As a result, stored procedures can improve the performance of the application.

4.2.4 Transactions

A transaction is a sequence of tasks performed as a single logical unit of work. While executing a transaction, if no errors are encountered, all the modifications in the transaction become a permanent part of the database. If errors are encountered, none of the modifications are made to the database. In the hospital application they were used for a sequence of tasks that have to be done together.

Example:

-- this procedure is executed when a patient with code pc is decided to be deleted
-- from the hospital’s database
-- that means that all relevant files have to be deleted and all tables involved have to
-- be informed
-- if one of these tasks fails, all will fail

CREATE PROCEDURE delete_patient
    @pc int
AS
BEGIN TRANSACTION
    -- start point of the transaction
DELETE FROM result
    -- delete the results of medical tests
WHERE result_code IN
    (SELECT reference_code FROM reference WHERE patient_code = @pc)

DELETE FROM reference
    -- delete references for medical tests
WHERE patient_code = @pc

DELETE FROM patient_nurse
    -- his nurses are not responsible for him anymore
WHERE patient_code = @pc

DELETE FROM medical_info
    -- delete his medical information
WHERE patient_code = @pc
**Business Logic and Data Logic Layer of “Medi Gen HOSPITAL” application**

```sql
UPDATE room  -- make his room is available
SET free = 1
WHERE (SELECT room FROM patient WHERE patient_code = @pc) = room

DELETE FROM patient  -- delete his personal data
WHERE patient_code = @pc

COMMIT  -- end point of the transaction
```

### 4.3 Accessing SQL Server Data in C# with ADO.NET

#### 4.3.1 Connect, Command and Execute

In the hospital application, the API used for accessing data in SQL Server is ADO.NET. ADO.NET is a set of classes that expose data access services to the .NET programmer. It consists of a rich set of components for creating distributed, data-sharing applications.

An application that uses ADO.NET to communicate with Microsoft SQL Server 2000 performs the following tasks:

- Connects with a database.
- Sends SQL statements.
- Processes the results.
- Processes errors and messages.
- Terminates the connection.

ADO.NET includes .NET Framework data providers for connecting to a database, executing commands, and retrieving results. In the application, the .NET Framework Data Provider for SQL Server is used to connect to SQL Server.

The SQL data provider classes are defined in the `System.Data.SqlClient` namespace and some general database related classes are defined in the `System.Data` namespace. The following code is written to include the namespaces in the application, in order to use ADO.NET:

```csharp
using System.Data;
using System.Data.SqlClient;
```

The source code of the application was written, as mentioned before, in Microsoft Visual Studio. To include these namespaces in the application, except for writing the above code, the `System.Data.dll` file has to be added to the References of the project.

#### Creating the connection to the database

The first step is to create a Connection object. The Connection object provides connectivity to the database. Using the Execute method of the Connection object is one way to execute an SQL statement against a SQL Server database. All connections require a connection string:

```csharp
private string connectionString = @"Initial Catalog=<database name>;" +
    @"Data Source=<server name>;" +
    @"User ID=<username>;" +
    @"Password=<password>";
```
A connection by the name `connection` is created with the following code:

```csharp
SqlConnection connection = new SqlConnection(connectionString);
```

That means that the database client application, which is the **hospital server**, needs a login account that allows connection with the database **HOSPITAL**, in order to access data. So, at the end of this section a reference is made on authentication modes and login accounts, and it is described how the login account for the user “userid” has been created, in order to be used by the **hospital server** for connecting to the database **HOSPITAL**.

The first action after creating the account is to select permit in database **HOSPITAL**, in user’s properties, so that “userid” is granted access to the database, otherwise he cannot access **HOSPITAL**.

The user “userid” is only member of public fixed database role, at least at first. No other fixed database role or fixed server role is given, if not needed. That is because it is important to give the database client as fewer privileges and permissions as possible, in order to ensure and preserve the security of the database. But, while developing the application, “userid” had to become member of dbcreator fixed server role in order to have permissions for certain tasks that have to be done.

Permissions for a user can be managed by selecting user’s properties, and permission to execute SQL statements (INSERT, UPDATE, SELECT) can be given by ticking the appropriate ones. At first no permissions were given to the user “userid”. The permissions were shaped and chosen along with the implementation of the application, according to the needs and demands for execution. Since, in the **hospital application** only stored procedures were used, “userid” was only given permissions to them.

So, the following code was used in the application to connect to **HOSPITAL**:

```csharp
SqlConnection conn = new SqlConnection("Data Source=.;" + "Initial Catalog=HOSPITAL;" + "User ID=userid;" + "Password=pass;";

conn.Open();    // opens the connection to the database
conn.Close();    // closes the connection
```

The ‘.’ used for data source points at the local server. Otherwise, ‘LOCAL’ could be used.

### Creating a command

Once the connection has opened, commands can be sent to the database to retrieve and manipulate data. The next step is to create a Command object and set up the command. The syntax for setting up a command is the same regardless of what kind of command it is (SELECT, INSERT, UPDATE, DELETE) or if it is a stored procedure.

```csharp
SqlCommand command = new SqlCommand("SELECT * FROM doctor",connection);
```

In case there is a stored procedure on the SQL server by the name “`select_doctor`”:

```csharp
SqlCommand command = new SqlCommand("select_doctor", connection);
```
• **properties**

The connection is included at the end of the strings. This is not necessary as long as the **Connection property** has been defined. Also, there is the **Timeout property** that can be defined in order to set the wait time before terminating the attempt to execute a command and generating an error. The **CommandType** property has to be defined in order to indicate how the **CommandText** property is to be interpreted. The default value is **Text**. When the **CommandType** property is set to **StoredProcedure**, the **CommandText** property should be set to the name of the stored procedure.

• **parameters**

For passing parameters to an SQL statement or a stored procedure, the **Parameters** collection is used. A **Parameter** object can be created using the **Parameter** constructor, or by calling the **Add** method. Also, the property **Value** can be used to set the value of the parameter. The names of the parameters added to the collection **must match** the names of the parameter markers in the SQL statement or the stored procedure, and, of course their data type and size **must match**.

**Example:** (case of a Transact-SQL statement)

```sql
-- insert a new medical test that can be carried out in the hospital
SqlCommand c = new SqlCommand("INSERT INTO medical_test (test_code, kind)
    VALUES (@tc, @k)", cn);
SqlParameter cpr1 = c.Parameters.Add("@tc", SqlDbType.Int);
cpr1.Value = i;      -- pass test code as input parameter
SqlParameter cpr2 = c.Parameters.Add("@k", SqlDbType.VarChar, 50);
cpr2.Value = kind;      -- pass kind of test as input parameter
```

**Executing a command**

The Command object provides several Execute methods that can be used to perform an operation in the database, such as: **ExecuteNonQuery**, which is used to execute commands that do not return rows, **ExecuteScalar**, which is used to return a single value, and **ExecuteReader**, which is used to return results as a stream of data.

• **ExecuteNonQuery**

Usually for INSERT, UPDATE and DELETE commands the **ExecuteNonQuery** method is used. The code is:

```csharp
command.ExecuteNonQuery();
```

This method returns an integer for the number of rows affected.

• **ExecuteScalar**

The **ExecuteScalar** method returns the first column of the first row of the result set, or a null reference if the result set is empty. It is usually used to return the value of an aggregate value, such as COUNT(), MAX(), and other. The code is:

```csharp
command.ExecuteScalar();
```
Example:

```csharp
SqlCommand cmd = new SqlCommand("SELECT COUNT (*) FROM patient", conn);
int count = (int) cmd.ExecuteScalar();
```

It can also be used in order to check if results matching the criteria of a SELECT statement have been found. This way, the use of ExecuteReader, which needs more code, is avoided.

Example:

```csharp
SqlCommand c1 = new SqlCommand("SELECT patient_code FROM patient
WHERE patient_code=@pc", cn);
SqlParameter p1 = c1.Parameters.Add("@pc", SqlDbType.Int);
p1.Value = pc;       // patient's code
if (c1.ExecuteScalar() == null)
    Console.WriteLine("There is no patient in database with
code :{0}", pc);
```

- **ExecuteReader**

  The DataReader is used to retrieve a read-only, forward-only stream of data from the database. The code for creating a DataReader is:

  ```csharp
  SqlDataReader reader  = command.ExecuteReader();
  ```

  The DataReader retrieves rows from the database, which are the results of one or more SELECT queries. The **Read** method is used to obtain a row from the results. Each column of the returned row can be reached by passing the name or ordinal reference of the column to the DataReader. After finishing with the DataReader, it **must always** be closed, by calling the **Close** method, or else the connection will not be able to execute any other commands or create another DataReader.

  Example:

  ```csharp
  SqlDataReader dr;                               // create the data reader
  ArrayList doctors = new ArrayList();    // array list to store the results

  // call the stored procedure that shows the personal data of all doctors in the hospital
  SqlCommand cmd = new SqlCommand("show_docs");
  cmd.CommandTimeout = 60;
  cmd.Connection = conn;
  cmd.CommandType = CommandType.StoredProcedure;
  dr = cmd.ExecuteReader();     // execute the command
  Console.WriteLine("Succesfully Selected 
");
  string s = null;

  if (dr.HasRows)         // if there are rows returned
      {                       
          while(dr.Read())    // read one row at a time
              {
```

Bika Maria

38
s=dr[0].ToString()+"+dr[1].ToString()+"+dr[2].ToString()+"+dr[3].ToString();
doctors.Add(s);  // add the data returned in the array list
}
}
dr.Close();    // close the data reader

The code of the stored procedure “show_docs” is:

CREATE PROCEDURE show_docs    -- shows all doctors in table doctor
AS
-- select all doctors' personal data
SELECT doctor_code, surname, name, specialization
FROM doctor
ORDER BY surname      -- in alphabetic order based on surname

The Read method returns true while there are still rows in the table to read, and must always be used before accessing the columns. There are several methods for accessing the column values in their native data types, such as GetString, GetInt32, GetChar and other. The HasRows property, is used to determine if the DataReader returned any results before reading from it. In case of multiple result sets, meaning when a command contains more than one SELECT statements, for each one of which a result set is returned, the DataReader provides the NextResult method to process these results in order.

The CommandBehavior values, overloaded to the ExecuteReader, provide a description of the results of the query, and can increase performance. SingleResult can be used when a query is expected to return a single result set, and SingleRow can be used when the query is expected to return a single row. A very important CommandBehavior flag is SequentialAccess, which enables the DataReader to load data as a stream. It is used when dealing with rows that have columns that contain large binary values (BLOB fields). When specifying no CommandBehavior flags, the default behavior is that the query returns multiple result sets.

4.3.2 Authentication Modes and Logins

Microsoft SQL Server can operate in one of two security (authentication) modes: the Windows Authentication Mode (Windows Authentication) and Mixed Mode (Windows Authentication and SQL Server Authentication). The authentication mode chosen for the hospital application is Mixed Mode.

Mixed Mode can be set up in the Enterprise Manager, by following the steps below:

1. Select the Properties for the LOCAL server, from the SQL server group.
2. In Security, authentication must be set to SQL Server and Windows.

When a user connects with a specified login name and password from a non-trusted connection, SQL Server performs the authentication itself by checking to see if a SQL Server login account has been set up and if the specified password matches
the one previously recorded. If SQL Server does not have a login account set, authentication fails and the user receives an error message.

A SQL Server login account must be created for the hospital server to use in order to connect to the HOSPITAL database. So, in the Enterprise Manager:

1. For the LOCAL server select Security, where all logins can be managed.
2. Create a new login account by the name “userid”, to be used over SQL Server Authentication, with the password “pass”.

4.4 Managing BLOB Fields

BLOB stands for binary large objects and BLOB fields are columns that contain BLOB values, such as text and image columns. A BLOB is stored in a database either as binary or character data, depending on the data type of the column that contains it. A SQL Server image field stores a BLOB value in binary format and a SQL Server text field stores a BLOB value as text.

NOTE: A column containing varbinary data is handled the same way as an image field.

4.4.1 Retrieving BLOB values from a database

As mentioned before, the CommandBehavior.SequentialAccess value can be passed to the ExecuteReader method to modify the default behavior of the DataReader. This way, rather than loading the entire row, SequentialAccess enables the DataReader to load data as a stream. For accessing the data in the BLOB field the GetBytes or GetChars method can be used. An important remark is that the different fields returned by the DataReader must be accessed in order.

Example:

The code of the stored procedure that retrieves an image value from the database is:

```sql
-- retrieve the image result with code rc
CREATE PROCEDURE retrieve_image
@rc int
AS
SELECT image_result
FROM result
WHERE result_code = @rc AND image_result IS NOT NULL
```

The above example will be completed when implementing the server in order to show the whole operation.
4.4.2 Writing BLOB values in a database

A BLOB can be written in a database by using the appropriate INSERT or UPDATE statement and passing the BLOB value as an input parameter. In case of a text field the BLOB can be passed as string parameter. In case of an image field the BLOB can be passed in a byte array as a binary parameter.

- **image**

Example 1:

The code used in the application for inserting an image result in table `result` is:

```csharp
Console.WriteLine("Please type the path of the image file you want to insert:");
Console.WriteLine("(.bmp, .jpg, .gif files supported...)");
string FilePath = Console.ReadLine();  // read the user’s input

// read the BLOB from the file
FileStream fs = new FileStream(FilePath, FileMode.Open, FileAccess.Read);
// streams the BLOB to the filestream object
BinaryReader br = new BinaryReader(fs);

// buffer to store the image
byte[] image = br.ReadBytes((int)fs.Length);
br.Close();
fs.Close();

Console.WriteLine("Please type the code of the result you want to insert/update:");
string x = Console.ReadLine();  // read the user’s input
rc = int.Parse(x);                  // result’s code

cn.Open();  // open the connection to the database

// the SQL statement to be executed
SqlCommand cmd = new SqlCommand("UPDATE result SET result_date=@date,
kind=@kind, image_result=@image WHERE result_code=@rc", cn);

SqlParameter prm1 = cmd.Parameters.Add(@date, SqlDbType.DateTime);
prm1.Value = DateTime.Now;  // current date and time
SqlParameter prm2 = cmd.Parameters.Add(@kind, SqlDbType.Int);
prm2.Value = 1;  // value 1 is used to show that the data stored is an image
SqlParameter prm3 = new SqlParameter(@image, SqlDbType.Image,
image.Length, ParameterDirection.Input, false, 0, 0, null,
DataRowVersion.Current, image);
prm3 = cmd.Parameters.Add(prm3);  // image result to be updated
SqlParameter prm4 = cmd.Parameters.Add(@rc, SqlDbType.Int);
prm4.Value = code;  // pass the result code as a parameter
```
Considering that a BLOB value may be quite large, if it is sent over a network to the database, like in the hospital application, in order to be inserted or updated, not the entire BLOB is sent, but it is sent in data units, as a series of messages and it must be written to the database in "chunks".

On the other hand, it is better to write BLOB data to a database in "chunks", as a BLOB value, due to its size, may consume extensive system memory when written as a single value, resulting in decreased application performance. So, stored procedures that accomplish the insertion of values in the BLOB fields, of the HOSPITAL database, are used, and, this way, the application's performance improves.

Example 2:

The following stored procedure “insert_history” updates the medical history of a patient in the column history of table medical_info, which is contains text data, of the table medical_info. Medical history is sent over a wireless network in messages, and data contained in each message is received and then appended to the text field.

UPDATETEXT is used, as it changes only a portion of a text or image column in place, and it is appropriate to modify this kind of data. With UPDATETEXT existing data can be replaced or deleted, or new data can be inserted. Its arguments are defined depending on the operation that has to be done.

As mentioned before, for each text column, its data is not stored in the row, but there is a pointer that points the data of the column. SQL Server saves space by not initializing text columns when explicit or implicit null values are placed in text columns with INSERT, and no text pointer can be obtained for such nulls. So, text columns must always be initialized to NULL, by using an UPDATE statement.

On the other hand, when there is a request to update the patient’s medical history, first all existing data have to be deleted. So, the stored procedure “clear_history” is executed first in order to either initialize the textpointer, if there is no data in the column, or to delete all existing data. The code of the stored procedure is:

```
CREATE PROCEDURE clear_history
    @pc int  -- patient's code
AS
EXEC sp_dboption 'HOSPITAL', 'select into/bulkcopy', 'true'
DECLARE @ptrval binary(16)
SELECT @ptrval = TEXTPTR(history)  -- select the textpointer of history column
FROM medical_info
WHERE medical_info.patient_code = @pc  -- for patient with patient code pc
IF @ptrval IS NOT NULL  -- if there is data in history
BEGIN
    UPDATETEXT medical_info.history @ptrval 0 NULL  -- delete all existing data
END
ELSE
    -- else if the textpointer is null
```
BEGIN
UPDATE medical_info
SET history = NULL                      -- initialize the text column to null
WHERE patient_code = @pc
END
EXEC sp_dboption 'HOSPITAL', 'select into/bulkcopy', 'false'

After all existing data is deleted, the new data will be inserted. The new medical history is sent by the client in messages, and each time a message is received the stored procedure “insert_history” is executed to append the received data to text. The code of the stored procedure is:

CREATE PROCEDURE insert_history
@pc int, @data varchar(256)         -- patient’s code, data received
AS
EXEC sp_dboption 'HOSPITAL', 'select into/bulkcopy', 'true'
DECLARE @ptrval binary(16)
SELECT @ptrval = TEXTPTR(history)        -- select the textpointer of history column
FROM medical_info
WHERE medical_info.patient_code = @pc              -- for patient with patient code pc
UPDATETEXT medical_info.history @ptrval NULL 0 @data  --append data to history
EXEC sp_dboption 'HOSPITAL', 'select into/bulkcopy', 'false'

The above example will be completed when implementing the server to show the whole operation.

To use UPDATETEXT the following statement is needed:
EXEC sp_dboption 'HOSPITAL', 'select into/bulkcopy', 'true'

That means that a database option needs to be modified. That is why the user “userid” of the database HOSPITAL has to become a member of dbcreator fixed server role, in order to have permission to execute the above statement. A dbcreator can create, alter, and drop databases.
CHAPTER 5: Implementing the Server

5.1 TCP Communication

The hospital application works by following the next steps:

The client sends its request for an operation to be performed in the database, the server “translates” this request to a SQL query and send it to the database. After the query has been executed and the results are available, the server sends back a response to the client. The server of the hospital application (HOSPITAL SERVER) has been developed according to this model.

The communication between the server and a client is implemented over TCP protocol.

The System.Net.Sockets.TcpListener class is used, as it provides simple methods that listen for and accept incoming connection requests in blocking synchronous mode, from TCP network clients. The following code is written to include the necessary namespaces in the application, in order to use the TcpListener class:

```
using System.Net;
using System.Net.Sockets;
```

The System.Net namespace provides a simple programming interface for many of the protocols used on networks today. The System.Net.Sockets namespace provides a managed implementation of the Windows Sockets interface for developers who need to tightly control access to the network.

To create a TcpListener, a Local IP address and a port number have to be specified. The Start method is used to begin listening for incoming connection requests, and to queue all incoming connections.

In the hospital application the TcpClient class is used, from the side of the client, to connect with the TcpListener. So, from the side of the server, each time
there is a request for creating a connection, a TcpClient has to be created to accept the connection. Then the `AcceptTcpClient` method of the TcpListener is used to pull the connection from the incoming connection request queue.

The `Stop` method can be used to stop listening for connections, but does not close any accepted connections. So, the server is responsible for closing each of these connections separately, by closing each TcpClient with the method `Close`, when it is through exchanging data.

The `System.Net.Sockets.NetworkStream` class is used for exchanging data between the server and the client, as it provides methods for sending and receiving data over Stream sockets in blocking mode. The method `Read` can be used to read data from the NetworkStream, and the method `Write` can be used to write data to the NetworkStream. The method `Close`, closes the NetworkStream, but it is not really needed as, closing the TcpClient, with his method Close, will release the NetworkStream to the garbage collector.

Once the server has accepted a connection request, the `GetStream` method of the TcpClient created is called to establish the connection, by returning the NetworkStream that will be used to send and receive data between the server and the client. So, the operations can begin.

---

**Figure 5.1: Client – Server TCP communication.**

### 5.2 Multithreading

The server implemented in the application, is multithreaded. That means that it can serve a lot of users at the same time. The server listens for connection requests from TCP clients; each time the server accepts a new TCP client, creates a new thread, which establishes the connection with the client and is responsible for managing the client’s requests. Each client is connected once to the server, and keeps the TCP connection open, while exchanging messages with the server, until
he accomplishes the wanted operations. The thread created for the client carries out these operations.

The advantage of this model is that the server can deal with all the clients at the same time. If a single thread was used instead of multiple threads, the server would accept a connection request from a TCP client, and wouldn’t be able to serve any other clients, until he was through communicating with the current client. The other connection requests arrived at the server, would queue waiting to be pulled and served, only one at a time.

On the other hand, using more than one thread is the most powerful technique available to increase responsiveness to the user and process the data necessary to get the job done at almost the same time. On a computer with one processor, multiple threads can create this effect, taking advantage of the small periods of time in between user events to process the data in the background.

The System.Threading namespace has to be included in the application, as it provides classes and interfaces that enable multithreaded programming. The code is:

```csharp
using System.Threading;
```

Creating a new instance of the Thread object creates new managed threads. The constructor for Thread takes, as its only parameter, a ThreadStart delegate that wraps the method that will be invoked by the new thread when the Thread.Start method is called to start the thread.

This method, which is named as ThreadProc in the server program, does all the work that has to be done. So, it is the method that contains all the services provided by the server and manages all the requests of the client.

The ThreadStart delegate has no parameters or return value. This means that a thread cannot be started using a method that takes parameters. That is why, to pass data to a thread, an object has to be created to hold the data and the thread method. Then an instance of this object, with the state information required, is created for each new thread. In the server program, the object ThreadWithState has been created for this purpose.

Example:

The multithreaded TCP server implemented in the application has the following code:

```csharp
using System;
using System.Net;
using System.Net.Sockets;
using System.Threading;

namespace tcp_server
{
    public class tcp_communication
    {

        [STAThread]
    }
}
```csharp
public static void Main()
{
    try
    {
        // Set the TcpListener on port 13000.
        int port = 13000;
        IPAddress localAddr = IPAddress.Parse("127.0.0.1");

        // Create the TcpListener with the specified values.
        TcpListener server = new TcpListener(localAddr, port);

        // Start listening for client requests.
        server.Start();

        int client_num = 0; // number of clients on the server

        // Enter the listening loop.
        while (true)
        {
            Console.Write("Waiting for a connection... ");

            // Perform a blocking call to accept requests.
            TcpClient client = server.AcceptTcpClient();
            Console.WriteLine("........Connected!");
            client_num++; // increase the number of clients
            Console.WriteLine("client number:{0}", client_num);

            // Supply the state information required.
            ThreadWithState tws = new ThreadWithState(client, client_num);

            // Create a thread to serve the client’s requests,
            // and then start the thread.
            Thread t = new Thread(new ThreadStart(tws.ThreadProc));
            t.Start();
        }
    }
    catch (SocketException e)
    {
        Console.WriteLine("SocketException: {0}" , e);
    }

    Console.WriteLine("\nHit enter to continue...");
    Console.Read();
}
```
5.3 Managing BLOB Values

In section 4.4 MANAGING BLOB FIELDS, it was described how BLOB values can be accessed and retrieved from a database. In this section it will be described how the server manages this kind of values.

Example 1:

As, shown in the example used in section 4.4.1 Retrieving BLOB values from a database, the stored procedure “retrieve_image” retrieves an image value from the column image_result of the table result of the database. So, if a client requests this value, the server, first, has to execute this stored procedure and, then, send the data retrieved to the client. As, the data will be sent over a network, and a BLOB value may be quite large, it will be sent in data units, as a series of messages. This is implemented with the following code:
public void retrieve_BLOB (int code, NetworkStream stream)
{
    SqlDataReader dr;
    try
    {
        // call the stored procedure retrieve_image
        SqlCommand cmd = new SqlCommand("retrieve_image");
        cmd.CommandTimeout = 60;
        cmd.Connection = conn;
        cmd.CommandType = CommandType.StoredProcedure;
        SqlParameter myParm = cmd.Parameters.Add(@"rc", SqlDbType.Int);
        myParm.Value = code;  // the result's code passed as input parameter
        // load data from SqlDataReader as a stream...
        dr = cmd.ExecuteReader(CommandBehavior.SequentialAccess);
        // the BLOB value will be read from SqlDataReader in "chunks" of 256 bytes
        // and will be sent to the client over the network in data units
        // as a series of messages

        int n=0;  // start index for reading data
        long k=0;  // number of bytes read from SqlDataReader
        byte[] msg = new byte[256];  //buffer for reading and writing data

        if (dr.HasRows)  // if data has been already inserted
        {
            while (dr.Read())
            {
                // every time read 256 bytes from the SqlDataReader
                k = dr.GetBytes(0,0+n,msg,0,256);
                while (k == 256)
                {
                    // send them to the client
                    stream.Write(msg, 0, msg.Length);
                    // increase the counter
                    n = n + 256;
                    // read the next 256 bytes
                    k = dr.GetBytes(0,n,msg,0,256);
                }
            }
            // when read from SqlDataReader less than 256 bytes
            // it is the last unit of data

            // if the size of the BLOB value is a pure multiplicate of 256 bytes
            // the last time 0 bytes will be read and the client
            // must be informed that is the “end of file”
            if (k == 0)
            {
                // send EOF
                msg = System.Text.Encoding.ASCII.GetBytes("EOF");
                stream.Write(msg, 0, msg.Length);
                Console.WriteLine("Send message: EOF");
            }
            else write the last bytes left of the BLOB value to stream
            if (k != 0)
It is important to describe the communication protocol followed:

The server sends the BLOB value, with a number of data units with size 256 bytes, which are read from the SqlDataReader. The last unit of data read from the SqlDataReader, is less than 256 bytes, so when sent to the client, he knows that the server is through sending the value.

In case that the size of the BLOB value is a pure multiplicate of 256 bytes, the server sends a number of messages with size 256 bytes, but the last unit of data is, also, 256 bytes. So, the client must be informed with a message “EOF” (or else he would be blocked waiting to read from stream).

Example 2:

As shown in the second example used in section 4.4.2 Writing BLOB values in a database, the stored procedure “clear_history” deletes any existing data from the text column history of the table medical_info and the stored procedure “insert_history” inserts new data to this column. So, if a client requests to change the history data (insert or update), the server, first, executes “clear_history”, then,
receives the BLOB value in data units, with a series of messages from the client, over the network, and each time he receives one of the messages, executes "insert_history", in order to insert history data in "chunks". This is implemented with the following code:

```csharp
public void insert_history (int pc, NetworkStream stream)
{
    Int32 i = 0;                                  // number of bytes read from NetworkStream
    Byte[ ] bytes = new Byte[256];   // buffer for reading data
    string data = null;                       // data received from client

    try
    {
        // first call the stored procedure to clear any existing data
        // if no data has yet been inserted the procedure initializes the textpointer
        SqlCommand cmd1 = new SqlCommand("clear_history");
        cmd1.CommandTimeout = 60;
        cmd1.Connection = conn;
        cmd1.CommandType = CommandType.StoredProcedure;
        SqlParameter myParm = cmd1.Parameters.Add("@pc", SqlDbType.Int);
        myParm.Value = pc;         // patient's code passed as input parameter
        cmd1.ExecuteNonQuery();    // execute the stored procedure to clear data

        // then call the stored procedure to insert new data
        SqlCommand cmd = new SqlCommand("insert_history");
        cmd.CommandTimeout = 60;
        cmd.Connection = conn;
        cmd.CommandType = CommandType.StoredProcedure;
        SqlParameter myParm1 = cmd.Parameters.Add("@pc", SqlDbType.Int);
        myParm1.Value = pc;        // patient's code passed as input parameter
        SqlParameter myParm2 = cmd.Parameters.Add("@data", SqlDbType.VarChar, 256);

        // every time read 256 bytes from NetworkStream
        i = stream.Read(bytes, 0, bytes.Length);
        data = System.Text.Encoding.ASCII.GetString(bytes, 0, i);

        // if history is empty the client sends message "EMPTY_STRING"
        if (data == "EMPTY_STRING")
            Console.WriteLine("Received message : EMPTY_STRING");

        // if there is new data in history to insert
        // the client sends them in data units, in a series of messages
        else
        {
            while (i == 256)
            {
                data = System.Text.Encoding.ASCII.GetString(bytes, 0, i);
                // pass the received data as input parameter
                myParm2.Value = data;
                // execute the procedure to insert the data
                // the received data are appended to the text field
            }
        }
    }
}
```
cmd.ExecuteNonQuery();
    // read the next 256 bytes
    i = stream.Read(bytes, 0, bytes.Length);
  }
  // when read from stream less than 256 bytes from stream
  // it is the last unit of data
  if (i != 0)
  {
    data = System.Text.Encoding.ASCII.GetString(bytes, 0, i);
    // if a message "end of file" has been sent from client it means
    // that there are no more bytes to insert in the BLOD field
    // else insert the last unit of data to database
    if (data != "EOF")
    {
      // insert the bytes left
      myParm2.Value = data;
      cmd.ExecuteNonQuery();
    }
    Console.WriteLine("Succesfully inserted \n");
  }
  catch(Exception exp)
  {
    Console.Write(exp.Message);
  }
}

It is important to describe the communication protocol followed:

The client sends the BLOB value, with a number of data units with size 256 bytes. The last unit of data is less than 256 bytes, so, when received, the server knows that client is through sending the value.

In case that the size of the BLOB value is a pure multiplicate of 256 bytes, the client sends a number of messages with size 256 bytes, but the last unit of data is, also, 256 bytes. So, the server must be informed with a message “EOF” (or else he would be blocked waiting to read from stream).
5.4 Communication Protocol and Services

5.4.1 Creating and Establishing the Connection

As described above, a new thread is created for each TCP client. The thread’s constructor initializes the thread with the state information of the client and then calls the ThreadProc method to serve the new client’s requests.

The ThreadProc first, by calling the GetStream method on the client, creates a NetworkStream to exchange data with the client. Then establishes a new connection to the database HOSPITAL, in this way:

An object named as manage_db has been created for the management of the database, in order to hold the connection string information for connecting to the database and all the methods that perform the necessary operations in it. The ThreadProc method creates an instance of this object, and its constructor creates the connection and opens it. The code is:

```csharp
// Get a network stream object for reading and writing.
NetworkStream stream = client.GetStream();
// Connect to the database.
manage_db db = new manage_db();
```

After accomplishing that, the thread is ready to receive the client’s requests and response to them. It is entering a while loop for receiving requests and satisfy them.

Once the thread stops receiving requests from the client, meaning the connection has closed from the client’s side, the connection to the database has to be closed and, also, the TcpClient has to be closed (the connection from the server’s side). The code is:

```csharp
// Shutdown and end connection with the client.
client.Close();
// End connection to the database.
db.end_conn();
```

5.4.2 Client Requests

When the client wants an operation to be performed, first sends a request containing a code, which indicates what type of request it is. So, the server checks this code and knows what operation needs to be done. This request may, also, be containing information concerning the task he wants to be carried out. So, the server checks, the first two bytes of the request, which define its type, and, also stores the other information. Then, the client with a series of messages sends the rest of the information needed for the task to be completed.

Depending on the type of the request, the server calls the appropriate method of the manage_db object, which, then, executes the stored procedure that performs the wanted task in the database.
Example: The server checks the request code using a switch statement.

```csharp
while((i = stream.Read(bytes, 0, bytes.Length))!=0)
{
    // Translate data bytes to an ASCII string.
    data = System.Text.Encoding.ASCII.GetString(bytes, 0, i);
    // Process the data sent by the client.
    string choice = data.Substring(0,2);  // get the request type
    data = data.Remove(0,2);      // store the rest of the data received
    Console.Write("<Request type [{0}] : ", choice);

    switch (choice)
    {
    .
    .
    .
    case "16":
        Console.WriteLine("SHOW THE MEDICAL TESTS THAT HAVE TO BE DONE
FOR THE PATIENT >");
        // data -->the code of the patient the client requests
        pc = int.Parse(data);
        // look into database and find all the references for
        // medical tests to be done
        object[] references = db.show_references(pc);
        // send them to the client
        send_records(references, stream, 2);
        break;

    case "17":
        Console.WriteLine("RETREIVE THE DOCTOR'S DIAGNOSIS FOR A RESULT >");
        // data -->the code of the result the client requests
        rc = int.Parse(data);
        Console.WriteLine("Server: Retrieves the diagnosis for the result of a medical test
and sends it to client.");
        Console.WriteLine("The code of the result is: {0}", rc);
        // retrieve diagnosis of the result from database
        // and send it to the client
        db.retrieve_BLOB(rc, stream, 2);
        break;
    }
}
```

### 5.4.3 Network Streams

A very important remark that has to be made is that, using the network streams created some problems, while developing the hospital application. The method Flush, which flushes data from the stream and is used for file streams, is supported for network streams but has no effect on them, as they are not buffered. So, there is no predefined way to force data to be flushed from stream.
The cases were problems appeared are the following ones:

The server writes data to the network stream for the client to receive, by calling the Write method, as many times as needed. The server makes different Write calls, in order to separate the data that sends, for the client to know what kind of data he is receiving. The client makes a Read call, for each Write call of the server. So, he is expected to read the data separately, but this does not always happen. Often, the client reads the data sent with a number of Write calls, with only one Read call and then, either he is blocked, as, due to the rest of the Read calls he is waiting to receive data, or the data received are not in valid form, mixed with other data, and errors occur.

![Figure 5.3: Repeated Write and Read calls.](image)

The same thing happens when the client repeatedly writes data to stream, by calling different Write methods, but the server receives data all together, even though he uses different Read calls.

To deal with this problem, some precautions were taken, each time the client and the server exchange data. So, when a record has to be sent, instead of sending each one of its fields separately, the whole record is sent with an identifier between the fields, used to split them when received. To separate the fields the characters ‘ ’ and ‘#’ were used.

![Figure 5.4: The fields of the record are separated by the character ‘ ’.](image)

On the other hand, when a series of records has to be sent, to avoid making repeated Write calls, the following model is implemented:

For each record a Write call is made by the sender with the appropriate data and a Read call is made by the receiver to get the record. An acknowledgement is sent from the receiver for confirmation, the sender receives the “ACK” message and, then, he sends the next record, by repeating this procedure for all records that have
Business Logic and Data Logic Layer of “Medi Gen HOSPITAL” application

to be sent. This way the repeated Writes and Reads are avoided and replaced with this first Write – then Read model. It is effective because, the one who has to read from stream, then makes a write operation on it, and as a result forces any existing data to be flushed.

![Diagram](image)

**Figure 5.5: First Write – then Read model.**

5.4.4 Describing the Protocol and the Services Provided

- **Request Code** [01]
- **Request Type** “Login”

This is the first request of the client after he has been connected to the server. It is needed in order to authenticate himself. The server receives the request code and the username of the client. He calls the appropriate method `db.login`, passing the username as an input parameter. This method executes the stored procedure “login_check” to obtain the password of the client, by, first, checking if the client is a doctor or a nurse and, then, by looking for the password in the appropriate table. In case of invalid username the server sends the message “NOTEXIST” to the client and waits for a new request. If the username is valid he sends message “EXIST” and waits to receive the client’s password. When received, he checks if it matches the password retrieved from database, and sends either “EXIST” in case of valid password, or “NOTEXIST”. An important remark that has to be made is that the username of the client is his code (doctor’s code or nurse’s code).

![Diagram](image)

**Figure 5.6: Communication Protocol for a “Login” request of an authorized user.**
This request is made by the client in order to view all the patients, for which he is responsible. The server receives the request code from the client, and calls the method `db.show_patients`, passing as input parameter his username, which was stored when the client was authenticated. This method executes the stored procedure “show_patients”, which retrieves the personal data of the patients from the database. If he is a doctor, these patients are ones that he attends, if he is a nurse, these are the ones that occupy rooms the nurse is responsible for. First, the number of the patients is sent to the client, in order to know how many records he is about to receive, and, then, an acknowledgement for that is received. Then, the records are sent one by one, and each time one is received from client, an acknowledgement is sent to server to send the next record.

The method `db.send_records` of the manage_db object, is used any time a number of records need to be sent to the client, and it works as described above:  
\[
\text{send the number of records} \rightarrow \text{receive ack} \rightarrow \text{(for each record)}
\]
\[
\text{send the record} \rightarrow \text{receive ack}
\]

The form of the record sent varies according to the needs of the client.

Here the form of the record sent: each field (patient’s code, surname, name, father’s name) of each patient separately, as a different record.
Business Logic and Data Logic Layer of “Medi Gen HOSPITAL” application

Figure 5.8: Communication Protocol for a “Show List Of My Patients” request .

- **Request Code** [03]
- **Request Type** Find A Patient

This request is made when the client wants to find the patients matching some criteria chosen. The server receives the request code and the surname of the patient. He may, also, receive the name or the father’s name of the patient, but these fields are not obligatory for the search. The method `db.find_pat` is called, with the data received as input parameters. This method executes the stored procedure “find_patient”, which looks for patients in the database based on the criteria. An important remark is that, the selection is made only for patients that the client attends. The results found are sent to the client, using the method `db.send_records`. **Form of the record sent:** each field (patient’s code, surname, name, father’s name) of each patient separately, as a different record.

Figure 5.9: Communication Protocol for a “Find A Patient” request .

- **Request Code** [04]
- **Request Type** Insert A New Patient

The client makes this request, in order to insert a new patient in the database. First, the server receives the request code and the surname of the patient. Then he receives the rest of the personal data of the patient in the following form: `name “#” father’s name “#” room number`. In case a nurse makes the insertion, the code of the
attendant doctor is, also, sent, as it is needed for the insertion. In case of a doctor, his own code is used for the insertion, which is known to the server (it is his username and it has been stored). After that the server calls the method `db.insert_pat`, with input parameters the data received. This method, first, executes the stored procedure “insert_check”, in order to check if this patient exists in the database. If the patient has already been inserted, the server sends message “ALREADY” to the client and waits for a new request. If not, the “set_code” stored procedure is executed, in order to automatically give the patient a code, by increasing the maximum patient code found in the database. Then follows the “insert_patient”, which makes the insertion with the data given. This procedure is responsible to inform all the tables relevant to the new insertion: the patient’s personal data and medical information are created, the nurses responsible for him are defined, according to the room he occupies and this room becomes unavailable. When completed, the message “OK” is sent to the client.

![Diagram](image1)

**Figure 5.10: Communication Protocol for a “Insert A Patient” request of a doctor.**

- **Request Code**: [05]
- **Request Type**: Show Doctors

The client makes this request in order to view all the doctors working in the hospital. The server receives the request code, and then he calls the method `db.show_docs`. This method executes the stored procedure "show_docs", which retrieves the personal data of all doctors in the hospital. The records found are sent to the client, using the method `db.send_records`. 

**Form of the record sent**: doctor’s code “ ” surname “ ” name “ ” specialization

![Diagram](image2)

**Figure 5.11: Communication Protocol for a “Show Doctors” request.**
• **Request Code** [06]  
  **Request Type** Show Available Rooms

The client makes this request in order to view all the rooms available in the hospital. The server receives the request code, and then he calls the method `db.show_rooms`, passing the username of the client as an input parameter. This method executes the stored procedure "show_rooms", which retrieves the numbers of the rooms that are not occupied. In case of a doctor, all the available rooms are returned, but in case of a nurse only the free rooms for which he is responsible are returned. The records found are sent to the client.  
**Form of the record sent:** room number

![Diagram of communication protocol for Show Available Rooms request]

**Figure 5.12: Communication Protocol for a “Show Available Rooms” request.**

• **Request Code** [07]  
  **Request Type** Show Medical Tests

The client makes this request in order to view all kinds of medical tests that can be carried out in the hospital. The server receives the request code, and then he calls the method `db.show_tests`. This method executes the stored procedure "show_tests", which retrieves all the tests from database.  
**Form of the record sent:** test's code “ ” kind of test

![Diagram of communication protocol for Show Medical Tests request]

**Figure 5.13: Communication Protocol for a “Show Medical Tests” request.**
• **Request Code** [08]
  **Request Type** Write New Medical Test

This request is made by the client, in order to insert in database a new reference for a medical test to be carried out. Only doctors are allowed to make a request of this kind, but there is no need for any check control in the server, as nurses are restrained from the application’s interface to perform it. First the server receives the request code and the code of the patient, for which the reference is written. Then he receives the rest of the data concerning the insertion in the following form: `test code “#” reference date “#” doctor’s notes`. After that the server calls the method `db.insert_reference`, with input parameters the data received. The doctor’s code is, also, needed for the insertion, which is known (the client’s username). This method, first, executes the stored procedure "set_code_ref", and a code is automatically set for the new reference, by increasing the maximum reference code found in the database. The "insert_reference" follows, which makes the insertion with the data given. This procedure is responsible to inform all the tables relevant to the new insertion: so, a row for the new reference is added in the table reference and, also, a row is added in the table result with the same result code as the reference code and null values in the other columns. When completed, the message “OK” is sent to the client.

![Diagram of communication protocol for writing a new medical test](image)

**Figure 5.14: Communication Protocol for a “Write New Medical Test” request.**

• **Request Code** [09]
  **Request Type** View History

The client makes this request, in order to retrieve the medical history of a patient from the database. The server receives the request code and the code of the patient, of whom he wants to view the medical history. Then he calls the method `db.retrieve_BLOB`, passing as input parameters the patient’s code and the kind of BLOB value he needs to obtain. This method executes the stored procedure "retrieve_history", which obtains the history from the database. Then the server reads it in "chunks" and sends it to the client, in data units, with a series of messages, as described in section 5.3 Managing BLOB Values. Also, the message “EOF” has to be sent, if necessary, to inform the client that the server is through sending the BLOB value. In case that, there is no data in history, either because no data has yes been inserted or because it is empty, the server sends message “NODATA”. The method `db.retrieve_BLOB` is used to retrieve all BLOB values from the database, and it works the same for each one of them, as described above.
The communication protocol followed for retrieving values from all BLOB fields in the database is the same.

- **Request Code** [10]
- **Request Type** Edit History

The client makes this request, in order to insert or update the medical history of a patient. The server receives the request code and the code of the patient, for which the medical history will be changed. Then he calls the method `db.insert_history`, passing as input parameter the patient’s code. This method, first, executes the stored procedure "clear_history", which is used to clear any existing data. Then the server starts receiving the BLOB value in data units, as a series of messages and each time a message is received, it executes the stored procedure "insert_history", in order to insert the value in the database in “chunks”, as described in the section Managing BLOB Values. The message “EOF” is, also, sent from client, if necessary, to inform the server that he is through sending the BLOB value. If the first message arrived, when start receiving the BLOB, is the "EMPTY_STRING” message, it means that there is no data to insert and history remains empty, as it has been cleared. The same procedure is followed for inserting all BLOB values in the database.

---

**Figure 5.15: Communication Protocol for a “View History” request.**

**Figure 5.16: Communication Protocol for a “Edit History” request.**

The communication protocol followed for inserting values in all BLOB fields of the database is the same as in Figure 5.12.
Business Logic and Data Logic Layer of “Medi Gen HOSPITAL” application

- **Request Code [11]**
  - **Request Type**: View Medical Info

  This request is made by the client, in order to obtain the medical information of a patient. The server receives the request code and the code of the patient, of whom the data the client demands to view. Then, he calls the method `db.show_medinfo1`, passing as input parameter the patient’s code. This method executes the stored procedure "show_medinfo1", which obtains all the necessary information from the database. The results are sent to the client in the following form: birthday “#” blood type “#” reason of insertion “#” doctor’s surname “ ” name

  ![Diagram](image1)

  **Figure 5.17: Communication Protocol for a “View Medical Info” request.**

- **Request Code [12]**
  - **Request Type**: Edit Medical Info

  The client makes this request, in order to insert or update the medical information of a patient. The server receives the request code and the code of the patient. Then he receives the data to be inserted in the following form: birthday “#” blood type “#” “reason of insertion”. He calls the method `db.edit_medinfo1`, passing as input parameters the patient’s code and the data given. He executes the stored procedure “edit_medinfo1”, which accomplishes the insertion. When completed, the server sends message “OK” to the client.

  ![Diagram](image2)

  **Figure 5.18: Communication Protocol for a “Edit Medical Info” request.**
Business Logic and Data Logic Layer of “Medi Gen HOSPITAL” application

• **Request Code** [13]  
  **Request Type** View Daily Progress

The client makes this request, in order to retrieve the daily progress of a patient from the database. The server receives the request code and the code of the patient, of whom he wants to view the daily history. So, the `db.retrieve_BLOB` method is called, passing as input parameters the patient’s code and the kind of BLOB value he needs to obtain. The stored procedure executed is the “retrieve_progress”, and the communication protocol is followed, as described.

![Diagram](image)

**Figure 5.19: Communication Protocol for a “View Daily Progress” request.**

• **Request Code** [14]  
  **Request Type** View Completed Tests

This request is made by the client, in order to view all the tests that have been carried out for a patient. The server receives the request code and the patient’s code, for which the tests done are requested. When a medical test is carried out and its results are ready, then these results are inserted in the database in the form of an image or a document, and, also, their date is inserted. The server calls the method `db.show_results`, passing the patient’s code as an input parameter. This method executes the stored procedure “show_results”, which looks into the database for results that have the same code with the patient’s references and their result date is not null, meaning that the result has been inserted in the database. The records found are sent to the client.

**Form of the record sent:** result’s code “#” medical test’s code “#” kind of test “#” result’s date

Bika Maria 64
Business Logic and Data Logic Layer of “Medi Gen HOSPITAL” application

Figure 5.20: Communication Protocol for a “View Completed Tests” request.

- **Request Code** [15]
- **Request Type** View results

The client makes this request, in order to retrieve the result of a medical test that has been done for a patient. The server receives the request code and the code of the result, which the client demands to view. Then he calls the method `db.retrieve_BLOB`, passing as input parameters the result’s code and the kind of BLOB value he needs to obtain. This method, first, executes the stored procedure "check_result", which defines, either the result is in the form of an image, or it is in the form of a document, by checking the column `kind` of the table `result`. If it is an image the server sends the message “ISIMAGE” to the client and executes the stored procedure "retrieve_image" to obtain the BLOB value from database. If it is a document he sends the message “ISDOC” and executes the stored procedure "retrieve_doc" to obtain it. When the client gets the message that indicates the type of data he is about to receive, he sends an acknowledgement. Then the server reads the BLOB value in “chunks” and sends it to the client, described in the previous section.

Figure 5.21: Communication Protocol for a “View Results” request.
Business Logic and Data Logic Layer of “Medi Gen HOSPITAL” application

- **Request Code** [16]
  - **Request Type** View Tests in Progress

This request is made by the client, in order to view all the tests that have to be done for a patient. The server receives the request code and the patient’s code, for which the tests in progress are requested. A test that is in progress is a test for which, either, the reference has been written, but the test has not been carried out yet, or its results are not still available. The server calls the method `db.show_references`, passing the patient’s code as an input parameter. This method executes the stored procedure "show_references", which looks into the database for results that have the same code with the patient’s references but their result date is null, meaning that the result has not yet been inserted in the database for one of the above reasons. The records found are sent to the client.

**Form of the record sent:** medical test’s code “#” kind of test “#” reference’s date “#” doctor’s notes

```
16 + patient code
```

```
med_code + "#" + kind + "#" + date + "#" notes
```

```
"ACK"
```

**Figure 5.22:** Communication Protocol for a “View Tests in Progress” request.

- **Request Code** [17]
  - **Request Type** View Diagnosis

The client makes this request, in order to retrieve the diagnosis for the result of a medical test that has been done from the database. The server receives the request code and the code of the result for which the diagnosis is written. Then he calls the method `db.retrieve_BLOB`, passing as input parameters the result’s code and the kind of BLOB value he needs to obtain. Then, the stored procedure "retrieve_diagnosis" is executed.

```
17 + result code
```

```
"NODATA"
```

```
"diagnosis.txt" file
```

```
if necessary
```

```
"EOF"
```

**Figure 5.23:** Communication Protocol for a “View Diagnosis” request.
• **Request Code** [18]
  **Request Type** Edit Diagnosis

The client makes this request, in order to insert or update the diagnosis for a result of a medical test. The server receives the request code and the code of the result for which the diagnosis will be changed. Then he calls the method `db.insert_diagnosis`, passing as input parameter the result’s code. The stored procedures executed are, first, "clear_diagnosis", and then, "insert_diagnosis".

![Diagram of communication protocol for "Edit Diagnosis" request.](image)

**Figure 5.24: Communication Protocol for a “Edit Diagnosis” request.**

• **Request Code** [19]
  **Request Type** Edit Daily Progress

The client makes this request, in order to add new data to the daily progress of a patient. The server receives the request code and the code of the patient to be updated. He calls the method `db.insert_progress`, passing as input parameter the patient’s code. Then the server receives a message that informs him of the size of the BLOB value he is about to receive. He sends an acknowledgement to the client and creates a buffer with the size given to store the BLOB value. Then, the server starts receiving the BLOB in data units, with a series of messages, each one of which is appended to the buffer when received. Message of type “EOF” is not needed, as the server knows the size of the value. Finally, the stored procedure ‘insert_progress’ is executed, which inserts the data received in the database.

![Diagram of communication protocol for "Edit Daily Progress" request.](image)

**Figure 5.25: Communication Protocol for a “Edit Daily Progress” request.**
Business Logic and Data Logic Layer of “Medi Gen HOSPITAL” application

Figure 5.26: Serving requests with codes [14] and [15].

Figure 5.27: Serving requests with codes [11] and [09].
It is assumed that in the hospital there is someone who is responsible for the maintenance of the database, the administrator of the database. His duty is to check the normal conduction of the operations in the database. On the other hand, he is responsible for performing some tasks that have to be done, but cannot be done from the doctors or nurses of the hospital.

The most important of them is to update the database with the results of the medical tests. When the result of a medical test that was carried out for a patient is ready, from the specialized doctor, it is sent to the administrator. Then he inserts this result in the database, so that it can be viewed from the attendant doctor to make his diagnosis. The contents of the result, which is either an image or a document, are inserted, and, in addition, the date of the result and its kind are inserted.

Other tasks that can be performed are, when there is a new doctor or a new nurse working in the hospital, the administrator is responsible for the insertion of their personal data in the database. For the nurse, also, the rooms for which he is responsible are defined. In addition, the insertion of a new medical test that can be carried out in the hospital is made by the administrator and, also, the insertion of the rooms of the hospital (this option is probably used only once, in the beginning).

Finally, only the administrator of the database can delete a patient. Usually, the files of a patient are stored in the database, even after he leaves the hospital, for future use. In case that the deletion of a patient is decided, for some reason (for example, his file is very old), it is performed by the administrator.

So, the administrator program was developed in order to accomplish the above tasks. The source code exists in the solution ADMINISTRATOR, in project Project1, by the name CodeFile1.cs.

The administrator of the HOSPITAL database is a System Administrator, by default member of the sysadmin fixed Server role. That means that his user ID is “sa” (built-in administrator login) and he has the permission to execute all statements and stored procedures in the database HOSPITAL. The password for this login is set to be “mariampika”. This assignment is made in the Enterprise Manager by following the next steps:

1. Select LOCAL from the SQL server group. In Security all logins are managed.
2. In Properties of the login sa, type the new password.

So, the connection string used in the administrator program in order to connect to the database HOSPITAL is:

```csharp
SqlConnection cn = new SqlConnection("Data Source=.;" + "Initial Catalog=HOSPITAL;" + "User ID=sa;" + "Password=mariampika;");
```

The insertion of the results of the tests is made by using this program, which reads the data from a file containing the results (.txt, .jpg), and inserts it in the database.

The insertion of doctors, nurses, tests and rooms can be made with this program, but, also, the administrator directly managing the tables with the Enterprise Manager can make this insertion.

The deletion of a patient is made by using this program, which is responsible to inform all the tables relevant to the patient deleted.

![Figure 6.1: Administrator program running.](image-url)
6.2 Video Data

It must be underlined that the database HOSPITAL and the hospital server are designed to support video data. But, the "Medi Gen Hospital" application is not ready yet to support video data from the client’s side, so that, this kind of data to be viewed from the user.

In the table result, there is the BLOB field video_result, of image data type, which can store video data in binary form. In the server program, code is written for retrieving video data from database and sending it to the client, and in the administrator program there is an option to insert the result of a medical test, in video form (the code of this option is in comments, as it must not be used yet).

This code has been written and maintained for future use.
CHAPTER 7: Conclusions

7.1 Conclusions

"Medi Gen HOSPITAL" is a 3-tier application. Organizing the application in three major parts helped to simplify this large-scale project and define what actions had to take place in each part. Besides, the application was developed by two different programmers, so it made possible each part to be developed independently from the others.

Carrying out the design and development of “Medi Gen HOSPITAL" application, led to the use of unknown tools and technologies. Microsoft SQL server was proved to be a powerful database management system, which made easy the implementation of the database. Enterprise Manager is a very useful tool, as it provides an easy-to-use programming interface, where all the objects of the database can be managed directly. Tables can be easily created and the relationships between them prevent from having redundant data in the database.

The only requirement for using SQL server is building statements with Transact-SQL, which is a really simple programming language, as using four basic statements performs almost all operations in a database.

The SQL stored procedures are a very powerful programming feature of SQL Server, and that is the reason why they were preferred, while developing the application. They are compiled when created, so they execute fast and improve the performance of the application. Their code is a part of SQL Server, so they made the application’s source code less complex. On the other hand, they can be modified without affecting the program’s code, and that was really helpful, as while developing the application often changes had to be made.

The hospital server needs to connect to the hospital database in order to retrieve and process the data stored. The ADO.NET classes provide access to the SQL Server database by establishing the connection and executing the SQL statements and stored procedures.
Visual Studio 2003 provides a friendly programming environment, as it provides tools for building and debugging programs. On the other hand Visual C# was proved to be an easy to learn and use programming language which provides a rich set of classes for developing this application.

The PDA client and the hospital server communicate over TCP protocol. Visual C# provides a rich set of classes that made easy the implementation of the communication. An important remark is that the communication protocol had to be defined, before starting implementing the server. That is because the server needs to know what kind of requests it receives and provide the appropriate services. The communication protocol is an agreement between the client and the server on what kind of messages they exchange.

The NetworkStreams were used for exchanging data between the server and the client, but they were proved to be unreliable. Often, they create problems, as there is no predefined way to flush data from a network stream. So, when using them this problem has to be eliminated or else they fail in carrying out the communication between the client and the server. An easy and effective way to deal with this problem is to make use of acknowledgement messages, as described.

Using multiple threads for implementing a multithreaded hospital server made this application high level and effective. The server is able to serve the requests of many clients at the same time, so it can correspond to the demands of a real hospital. Responsiveness to user is increased and all requests are accomplished. It has to be noted that implementing multiple threads was easy using C#.