NEAR EAST UNIVERSITY

Faculty of Engineering

Department of Computer Engineering

Milk Stock Program With Delphi

Graduation Project
COM400

Student: Ercan Top (20033553)

Supervisor: Assist. Prof. Dr Elbrus Imanov

Nicosia-2008

ACKNOWLEDGEMENT

First of all I am very happy to complete the my project which I had given with blessing of ALLAH and also I am indebted to all my instructors and my friends in my education life who have, supported me, advised me. Taught me and who have always encouraged me to follow my dreams and ambitions.

I especially to thank my supervisor, Assist. Prof. Dr Elbrus İmanov, for intellectual support and encouragement that made this project possible, and his associate for correcting both my mistakes and my software's errors.

And thank my dearest parents who supported me to continue until my undergraduate studies, to my father who have much contributions to my all studies and to my mother who encouraged me along the way.

To all my friends, especially Mr Mustafa Yucel and Yunus Kaba for sharing wonderful moments, advice, and for making me feel at home.

Finally, l wish to thank the all individuals, who made this project possible,dedicating their valuable time and energy to its completion.

ABSTRACT

The Milk stock program is going to prepare to solve the problems and the difficulties that milk stock controlling and employee working that is made by manually. The milk stock software to make easy the works of the cheese manufacturers in large platforms. Because the computers take place in every part of our lives, to equip them with programs that relieve our live is a good idea. This software is going to make, stock controlling and employee working ease and fast.

The main aim of this Project is making the users job easy which provides the stock controlling features and evaluation criteria is based on a small to medium cheese manufacturing company. The complex jobs like calculating the bill, Money movement are calculated by the user in the program.. A scheduled user manual prepared for helping the users to select an appropriate action.

TABLE OF CONTENTS

INTRODUCTION

In end days Information technology is a key investment for many small businesses. Computer software is part of that investment. People will need software to make computer hardware perform useful business functions. It is an important part of your business assets, and should be chosen so that it matches your business needs. Software can help your business work more efficiently and effectively and lead to new ways of working. This program will help you to make easy some complex tasks of the cheese companies and how to choose the most appropriate way for this business.

The companies were doing their jobs manually in this field, such as accounting, stock controlling and etc. But recently Information Technology started to help companies or firms. Then it has been very popular because it is faster, cheaper than manually and so easy work with IT. At this point Milk Stock program will provide easiness and quickness of controlling stock. In my project the main point is making the users job easy. Let us look at together

In chapter one the basic concept of delphi and introduction part were explained. Generally delphi's characterictics and special features were explained. Also you can get basic programming knowledges about the Delphi programming language in this chapter.

In chapter two database concept of paradox was explained and described how to use in delphi programming. Paradox and characterictics of components were explained. And also was given some useful techniques about the creating a well defined table in paradox database system.

In chapter three particularly is consisted of easy comprehensible user guide, interfaces, user manuals and illustrations of all the form which used in the Project. This chapter is very significiant for user because all visual forms and main description are included in this chapter.

And finally appendix part containing source code of the program.

CHAPTER 1

DELPHI

1.1 INTRODUCTION TO DELPHI

The name "Delphi" was never a term with which either Olaf Helmer or Norman Dalkey (the founders of the method) were particular happy. Since many of the early Delphi studies focused on utilizing the technique to make forecasts of future occurrences, the name was first applied by some others at Rand as a joke. However, the name stuck. The resulting image of a priestess, sitting on a stool over a crack in the earth, inhaling sulfur fumes, and making vague and jumbled statements that could be interpreted in many different ways, did not exactly inspire confidence in the method.

The straightforward nature of utilizing an iterative survey to gather information "sounds" so easy to do that many people have done "one" Delphi, but never a second. Since the name gives no obvious insight into the method and since the number of unsuccessful Delphi studies probably exceeds the successful ones, there has been a long history of diverse definitions and opinions about the method. Some of these misconceptions are expressed in statements such as the following that one finds in the literature:

It is a method for predicting future events.

It is a method for generating a quick consensus by a group.

It is the use of a survey to collect information.

It is the use of anonymity on the part of the participants.

It is the use of voting to reduce the need for long discussions.

It is a method for quantifying human judgement in a group setting.

Some of these statements are sometimes true; a few (e.g. consensus) are actually contrary to the purpose of a Delphi. Delphi is a communication structure aimed at producing detailed critical examination and discussion, not at forcing a quick compromise. Certainly quantification is a property, but only to serve the goal of quickly identifying agreement and disagreement in order to focus attention. It is often very common, even today, for people to come to a view of the Delphi method that reflects a particular application with which they are familiar. In 1975 Linstone and Turoff proposed a view of the Delphi method that they felt best summarized both the technique and its objective:

"Delphi may be characterized as a method for structuring a group communication process, so that the process is effective in allowing a group of individuals, as a whole, to deal with complex problems." The essence of Delphi is structuring of the group communication process. Given that there had been much earlier work on how to facilitate and structure face-to-face meetings, the other important distinction was that Delphi was commonly applied utilizing a paper and pencil communication process among groups in which the members were dispersed in space and time. Also, Delphis were commonly applied to groups of a size (30 to 100 individuals) that could not function well in a face-to-face environment, even if they could find a time when they all could get together.

Additional opportunity has been added by the introduction of Computer Mediated Communication Systems (Hiltz and Turoff, 1978; Rice and Associates, 1984; Turoff, 1989; Turoff, 1991). These are computer systems that support group communications in either a synchronous (Group Decision Support Systems, Desanctis et. al., 1987) or an asynchronous manner (Computer Conferencing). Techniques that were developed and refined in the evolution of the Delphi Method (e.g. anonymity, voting) have been incorporated as basic facilities or tools in many of these computer based systems. As a result, any of these systems can be used to carry out some form of a Delphi process or Nominal Group Technique (Delbecq, et. al., 1975).

The result, however, is not merely confusion due to different names to describe the same things; but a basic lack of knowledge by many people working in these areas as to what was learned in the studies of the Delphi Method about how to properly employ these techniques and their impact on the communication process. There seems to be a great deal of "rediscovery" and repeating of earlier misconceptions and difficulties.

Given this situation, the primary objective of this chapter is to review the specific properties and methods employed in the design and execution of Delphi Exercises and to examine how they may best be translated into a computer based environment.

1.2 WHAT IS DELPHI?

Delphi is an object oriented, component based, visual, rapid development environment for event driven Windows applications, based on the Pascal language. Unlike other popular competing Rapid Application Development (RAD) tools, Delphi compiles the code you write and produces really tight, natively executable code for the target platform. In fact the most recent versions of Delphi optimise the compiled code and the resulting executables are as efficient as those compiled with any other compiler currently on the market.The term "visual" describes Delphi very well. All of the user interface development is conducted in a What You See Is What You Get environment (WYSIWYG), which means you can create polished, user friendly interfaces in a very short time, or prototype whole applications in a few hours.

Delphi is, in effect, the latest in a long and distinguished line of Pascal compilers (the previous versions of which went by the name "Turbo Pascal") from the company formerly known as Borland, now known as Inprise. In common with the Turbo Pascal compilers that preceded it, Delphi is not just a compiler, but a complete development environment. Some of the facilities that are included in the "Integrated Development Environment" (IDE) are listed below:

- A syntax sensitive program file editor
- A rapid optimising compiler
- Built in debugging /tracing facilities
- A visual interface developer
- Syntax sensitive help files
- Database creation and editing tools

- Image/Icon/Cursor creation / editing tools

- Version Control CASE tools What's more, the development environment itself is extensible, and there are a number of add ins available to perform functions such as memory leak detection and profiling.

In short, Delphi includes just about everything you need to write applications that will run on an Intel platform under Windows, but if your target platform is a Silicon Graphics running IRIX, or a Sun Sparc running SOLARIS, or even a PC running LINUX, then you will need to look elsewhere for your development tools.

This specialisation on one platform and one operating system, makes Delphi a very strong tool. The code it generates runs very rapidly, and is very stable, once your own bugs have been ironed out!

1.3 WHAT KIND OF PROGRAMMING CAN YOU DO WITH DELPHI?

The simple answer is "more or less anything". Because the code is compiled, it runs quickly, and is therefore suitable for writing more or less any program that you would consider a candidate for the Windows operating system.

You probably won't be using it to write embedded systems for washing machines, toasters or fuel injection systems, but for more or less anything else, it can be used (and the chances are that probably someone somewhere has!)

Some projects to which Delphi is suited:

- Simple, single user database applications

- Intermediate multi-user database applications

- Large scale multi-tier, multi-user database applications

- Internet applications

- Graphics Applications

- Multimedia Applications
- Image processing/Image recognition
- Data analysis
- System tools

This is not intended to be an exhaustive list, more an indication of the depth and breadth of Delphi's applicability. Because it is possible to access any and all of the Windows API, and because if all else fails, Delphi will allow you to drop a few lines of assembler code directly into your ordinary Pascal instructions, it is possible to do more or less anything. Delphi can also be used to write Dynamically Linked Libraries (DLLs) and can call out to DLLs written in other programming languages without difficulty.

Because Delphi is based on the concept of self contained Components (elements of code that can be dropped directly on to a form in your application, and exist in object form, performing their function until they are no longer required), it is possible to build applications very rapidly. Because Delphi has been available for quite some time, the number of pre-written components has been increasing to the point that now there is a component to do more or less anything you can imagine. The job of the programmer has become one of gluing together appropriate components with code that operates them as required.

1.4 VERSIONS ARE THERE AND HOW DO THEY DIFFER?

Borland (as they were then) has a long tradition in the creation of high speed compilers. One of their best known products was Turbo Pascal - a tool that many programmers cut their teeth on. With the rise in importance of the Windows environment, it was only a matter of time before development tools started to appear that were specific to this new environment.

In the very beginning, Windows produced SDKs (software development kits) that were totally non-visual (user interface development was totally separated from the development of the actual application), and required great patience and some genius to get anything working with. Whilst these tools slowly improved, they still required a really good understanding of the inner workings of Windows.

To a great extent these criticisms were dispatched by the release of Microsoft's Visual Basic product, which attempted to bring Windows development to the masses. It achieved this to a great extent too, and remains a popular product today. However,it suffered from several drawbacks:

1) It wasn't as stable as it might have been

2) It was an interpreted language and hence was slow to run

3) It had as its underlying language BASIC, and most "real" programmers weren't so keen!

Into this environment arrived the eye opening Delphi I product, and in many ways the standard for visual development tools for Windows was set. This first version was a 16 bit compiler, and produced executable code that would run on Windows 3.1 and Windows 3.11. Of course, Microsoft have ensured (up to now) that their 32 bit operating systems (Win95, Win98, and Win NT) will all run 16 bit applications, however, many of the features that were introduced in these newer operating systems are not accessible to the 16 bit applications developed with Delphi I.

Delphi 2 was released quite soon after Delphi I, and in fact included a full distribution of Delphi I on the same CD. Delphi 2, (and all subsequent versions) have been 32 bit compilers, producing code that runs exclusively on 32bit Windows platforms. (We ignore for simplicity the WIN32S DLLs which allow Win 3.1x to run some 32 bit applications).

Delphi is currently standing at Version 4.0, with a new release (version 5.0) expected shortly. In its latest version, Delphi has become somewhat feature loaded, and as a result, we would argue, less stable than the earlier versions. However, in its defence, Delphi (and Borland products in general) have always been more stable than their competitors products, and the majority of Delphi 4's glitches are minor and forgivable -

just don't try and copy/paste a selection of your code, midway through a debugging session!

The reasons for the version progression include the addition of new components, improvements in the development environment, the inclusion of more internet related support and improvements in the documentation. Delphi at version 4 is a very mature product, and Inprise has always been responsive in developing the product in the direction that the market requires it to go. Predominantly this means right now, the inclusion of more and more Internet, Web and CORBA related tools and components - a trend we are assured continues with the release of version 5.0

For each version of Delphi there are several sub-versions, varying in cost and features, from the most basic "Developer" version to the most complete (and expensive) "Client Server" version. The variation in price is substantial, and if you are contemplating a purchase, you should study the feature list carefully to ensure you are not paying for features you will never use. Even the most basic "Developer" version contains the vast majority of the features you are likely to need on a day to day basis. Don't assume that you will need Client Server, simply because you are intending to write a large database application - The developer edition is quitcapable ofthis.

1.5 SOME KNOWLEDGE ABOUT DELPHI

Delphi is a Rapid Application Development (RAD) environment. It allows you to drag and drop components on to a blank canvas to create a program. Delphi will also allow you to use write console based DOS like programs.

Delphi is based around the Pascal language but is more developed object orientated derivative. Unlike Visual Basic, Delphi uses punctuation in its basic syntax to make the program easily readable and to help the compiler sort the code. Although Delphi code is not case sensitive there is a generally accepted way of writing Delphi code. The main reason for this is so that any programmer can read your code and easily understand what you are doing, because they write their code like you write yours.

Luckily this isn't the end of the article so we'll actually have a worthwhile program at the end of it. All we need to do is insert some code in the main procedure we have just made.

Every good programmer's first program was 'Hello World' and you'll be no exception. All we need to do is use the WriteLn procedure to write 'Hello World!' to the console, simple.Notice the semicolon at the end of the line, at the end of any statement you need to add a semicolon. Run the program and see the results...

Now I don't know about you but I saw hello world flash up and go away in a second, if you didn't write the program you wouldn't even know what it said. To solve this problem we need to tell the program to leave the console open until the user is ready to close it. We can use ReadLn for this which reads the users input from the console.

Delphi Code:

```
program HelloWorld;

 {$APPTYPE CONSOLE}

 uses

   SysUtils;

 begin

   WriteLn('Hello World!' + #13#10 + #13#10 +

     'Press RETURN to end...');

   ReadLn;

 end.
```

I have added a few extra things into the 'Hello World' string so the user knows what to do to end the program as it could be a bit confusing. '#13#10' is to insert a carriage return as 13 and 10 are the ASCII codes for a carriage return followed by a new line feed. ASCII can be inserted in this way into strings.

1.5.2 Delphi Style

Coding style, the way you format your code and the way in which you present it on the page.At the end of the day who cares about my style, I can read it, and Delphi strips all the spaces out of it and doesn't care if I indent. Why waste my time?

Neatly present code which conforms to the accepted standards not only makes your code much easier for you to read and debug but also but any one else who might read your code to help you, or learn from you can do so with ease. After all which code is easier to follow, example 1 or 2?

Delphi Code:

```
// Example 1

procedure xyz();

var

x,y,z,a:integer;

begin

x:=1;y:=2;

for z:=x to y do begin

a:=power(z,y);

showmessage(inttostr(a));

end;
```

```
end;
```

Delphi Code:

```
// Example 2

procedure XYZ();

var

X,Y,Z,A: Integer;

begin

 X := 1;

 Y := 2;

 for Z := X to Y do

 begin

      A := Power(Z, Y);

      ShowMessage(IntToStr(A));

 end; // for end

end; // procedure end
```

Design patterns are frequently recurring structures and relationships in object-oriented design. Getting to know them can help you design better, more reusable code and also help you learn to design more complex systems.

Much of the ground-breaking work on design patterns was presented in the book Design Patterns: Elements of Reusable Object-Oriented Software by Gamma, Helm, Johnson and Vlissides. You might also have heard of the authors referred to as "the Gang of Four". If you haven't read this book before and you're designing objects, it's an excellent primer to help structure your design. To get the most out of these examples, I recommend reading the book as well.

Another good source of pattern concepts is the book Object Models: Strategies, Patterns and Applications by Peter Coad. Coad's examples are more business oriented and he emphasises learning strategies to identify patterns in your own work.

1.6 HOW DELPHI HELPS YOU DEFINE PATTERNS

Delphi implements a fully object-oriented language with many practical refinements that simplify development.

The most important class attributes from a pattern perspective are the basic inheritance of classes; virtual and abstract methods; and use of protected and public scope. These give you the tools to create patterns that can be reused and extended, and let you isolate varying functionality from base attributes that are unchanging.

Delphi is a great example of an extensible application, through its component architecture, IDE interfaces and tool interfaces. These interfaces define many virtual and abstract constructors and operations.

1.6.1 Delphi Examples of Design Patterns

I should note from the outset, there may be alternative or better ways to implement these patterns and I welcome your suggestions on ways to improve the design. The following patterns from the book *Design Patterns* are discussed and illustrated in Delphi to give you a starting point for implementing your own Delphi patterns.

Pattern Name	*Definition*
Singleton	"Ensure a class has only one instance, and provide a global point of access to it."
Adapter	"Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't

	otherwise because of incompatible interfaces."
Template Method	"Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure."
Builder	"Separate the construction of a complex object from its representation so that the same construction process can create different representations."
Abstract Factory	"Provide an interface for creating families of related or dependant objects without specifying their concrete classes."
Factory Method	"Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory method lets a class defer instantiation to subclasses."

Note: These definitions are taken from *Design Patterns*.

1.6.2 Pattern: Singleton

1.6.2.1 Definition

"Ensure a class has only one instance, and provide a global point of access to it."

This is one of the easiest patterns to implement.

1.6.2.2 Applications in Delphi

There are several examples of this sort of class in the Delphi VCL, such as TApplication, TScreen or TClipboard. The pattern is useful whenever you want a single global object in your application. Other uses might include a global exception handler, application security, or a single point of interface to another application.

1.6.2.3 Implementation Example

To implement a class of this type, override the constructor and destructor of the class to refer to a global (interface) variable of the class.

Abort the constructor if the variable is assigned, otherwise create the instance and assign the variable.

In the destructor, clear the variable if it refers to the instance being destroyed.

Note: To make the creation and destruction of the single instance automatic, include its creation in the initialization section of the unit. To destroy the instance, include its destruction in an ExitProc (Delphi 1) or in the finalization section of the unit (Delphi 2).

1.6.3 Pattern: Adapter

1.6.3.1 Definition

"Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces."

1.6.3.2 Applications in Delphi

A typical example of this is the wrapper Delphi generates when you import a VBX or OCX. Delphi generates a new class which translates the interface of the external control into a Pascal compatible interface. Another typical case is when you want to build a single interface to old and new systems.

Note Delphi does not allow class adaption through multiple inheritance in the way described in Design Patterns. Instead, the adapter needs to refer to a specific instance of the old class.

1.6.3.3 Implementation Example

The following example is a simple (read only) case of a new customer class, an adapter class and an old customer class. The adapter illustrates handling the year 2000 problem, translating an old customer record containing two digit years into a new date format. The client using this wrapper only knows about the new customer class. Translation between classes is handled by the use of virtual access methods for the properties. The old customer class and adapter class are hidden in the implementation of the unit.

1.6.4 Pattern: Template Method

1.6.4.1 Definition

"Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure."

This pattern is essentially an extension of abstract methods to more complex algorithms.

1.6.4.2 Applications in Delphi

Abstraction is implemented in Delphi by abstract virtual methods. Abstract methods differ from virtual methods by the base class not providing any implementation. The descendant class is completely responsible for implementing an abstract method. Calling an abstract method that has not been overridden will result in a runtime error.

1.6.4.3 A typical example of abstraction is the TGraphic class.

TGraphic is an abstract class used to implement TBitmap, TIcon and TMetafile. Other developers have frequently used TGraphic as the basis for other graphics objects such as PCX, GIF, JPG representations. TGraphic defines abstract methods such as Draw, LoadFromFile and SaveToFile which are then overridden in the concrete classes. Other objects that use TGraphic, such as a TCanvas only know about the abstract Draw method, yet are used with the concrete class at runtime.

Many classes that use complex algorithms are likely to benefit from abstraction using the template method approach. Typical examples include data compression, encryption and advanced graphics processing.

1.6.4.4 Implementation Example

To implement template methods you need an abstract class and concrete classes for each alternate implementation. Define a public interface to an algorithm in an abstract base class. In that public method, implement the steps of the algorithm in calls to protected abstract methods of the class. In concrete classes derived from the base class, override each step of the algorithm with a concrete implementation specific to that class.

1.6.5 Pattern: Builder

1.6.5.1 Definition

"Separate the construction of a complex object from its representation so that the same construction process can create different representations."

A Builder seems similar in concept to the Abstract Factory. The difference as I see it is the Builder refers to single complex objects of different concrete classes but containing multiple parts, whereas the abstract factory lets you create whole families of concrete classes. For example, a builder might construct a house, cottage or office. You might employ a different builder for a brick house or a timber house, though you would give them both similar instructions about the size and shape of the house. On the other hand the factory generates parts and not the whole. It might produce a range of windows for buildings, or it might produce a quite different range of windows for cars.

1.6.5.2 Applications in Delphi

The functionality used in Delphi's VCL to create forms and components is similar in concept to the builder. Delphi creates forms using a common interface, through Application.CreateForm and through the TForm class constructor. TForm implements a common constructor using the resource information (DFM file) to instantiate the components owned by the form. Many descendant classes reuse this same construction process to create different representations. Delphi also makes developer extensions easy. TForm's OnCreate event also adds a hook into the builder process to make the functionality easy to extend.

1.6.5.3 Implementation Example

The following example includes a class TAbstractFormBuilder and two concrete classes TRedFormBuilder and TBlueFormBuilder. For ease of development some common functionality of the concrete classes has been moved into the shared TAbstractFormBuilder class.

1.6.6 Pattern: Abstract Factory

1.6.6.1 Definition

"Provide an interface for creating families of related or dependant objects without specifying their concrete classes."

The Factory Method pattern below is commonly used in this pattern.

1.6.6.2 Applications in Delphi

This pattern is ideal where you want to isolate your application from the implementation of the concrete classes. For example if you wanted to overlay Delphi's VCL with a common VCL layer for both 16 and 32 bit applications, you might start with the abstract factory as a base.

1.6.6.3 Implementation Example

The following example uses an abstract factory and two concrete factory classes to implement different styles of user interface components. TOAbstractFactory is a singleton class, since we usually want one factory to be used for the whole application.

At runtime, our client application instantiates the abstract factory with a concrete class and then uses the abstract interface. Parts of the client application that use the factory don't need to know which concrete class is actually in use.

1.6.7 Pattern: Factory Method

1.6.7.1 Definition

"Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory method lets a class defer instantiation to subclasses."

The Abstact Factory pattern can be viewed as a collection of Factory Methods.

1.6.7.2 Applications in Delphi

This pattern is useful when you want to encapsulate the construction of a class and isolate knowledge of the concrete class from the client application through an abstract interface.

One example of this might arise if you had an object oriented business application potentially interfacing to multiple target DBMS. The client application only wants to know about the business classes, not about their implementation-specific storage and retrieval.

1.6.7.3 Implementation Example

In the Abstract Factory example, each of the virtual widget constructor functions is a Factory Method. In their implementation we define a specific widget class to return.

1.7 KEY ELEMENTS OF DELPHI CLASS DEFINITIONS

1.7.1 Unit Structure

Delphi units (.PAS files) allow declaration of interface and implementation sections. The interface defines the part that is visible to other units using that unit. The keyword *uses* can be added to a unit's interface or implementation section to list the other units that your unit uses. This indicates to the compiler that your unit refers to parts of the used unit's interface. Parts of a unit declared in the implementation section are all private to that unit, i.e. never visible to any other unit. Types, functions and procedures declared in the interface of a unit must have a corresponding implementation, or be declared as external (e.g. a call to a function in a DLL).

1.7.2 Class Interfaces

Classes are defined as types in Delphi and may contain fields of standard data types or other objects, methods declared as functions or procedures, and properties. The type declaration of a class defines its interface and the scope of access to fields, methods and properties of the class. Class interfaces are usually defined in the interface of a unit to make them accessible to other modules using that unit. However they don't need to be. Sometimes a type declaration of a class may be used only within the implementation part of a unit.

1.7.3 Properties

Properties are a specialised interface to a field of a defined type, allowing access control through read and write methods. Properties are not virtual, you can replace a property with another property of the same name, but the parent class doesn't know about the new property. It is however possible to make the access methods of a property virtual.

1.7.4 Inheritance

Delphi's inheritance model is based on a single hierarchy. Every class inherits from TObject and can have only one parent.

A descendant class inherits all of the interface and functionality of its parent class, subject to the scope described below.

Multiple inheritance from more than one parent is not allowed directly. It can be implemented by using a container class to create instances one or more other classes and selectively expose parts of the contained classes.

Private, Protected, Public and Published ScopeScope refers to the visibility of methods and data defined in the interface of a class, i.e. what parts of the class are accessible to the rest of the application or to descendant classes.

The default scope is public, for instance the component instances you add to a form at design time. Public says "come and get me"; it makes the data or method visible to everything at runtime.

Published parts of a class are a specialized form of Public scope. They indicate special behaviour for classes derived from TPersistent. A persistent class can save and restore its published properties to persistent storage using Delphi's standard streaming methods. Published properties also interact with Delphi Object Inspector in the IDE. A class must descend from TPersistent in order to use Published. There's also not much point in publishing methods, since you can't store them, although Delphi's compiler doesn't stop you. Published also lets another application access details of the class through Delphi's runtime type information. This would be rarely used, except in Delphi's design time interaction with its VCL.

Encapsulation or information hiding is essential to object orientation, so Protected and Private scope let you narrow the access to parts of a class.

Protected parts are visible only to descendant classes, or to other classes defined in the same unit.

Private parts are visible only to the defining class, or to other classes defined in the same unit.

It's important to note that once something is given public or published scope, it cannot be hidden in descendant classes.

Static, Virtual and Dynamic Methods; Override and Inherited

Methods declared as virtual or dynamic let you change their behaviour using override in a descendant class. You're unlikely to see a virtual method in the private part of a class, since it could only be overridden in the same unit, although Delphi's compiler doesn't stop you from doing this.

Override indicates that your new method replaces the method of the same name from the parent class. The override must be declared with the same name and parameters as the original method.

When a method is overridden, a call to the parent class's method actually executes the override method in the real class of the object.

Static methods on the other hand have no virtual or override declaration. You can replace a method of a class in a descendant class by redeclaring another method, however this is not object oriented. If you reference your descendant class as the parent type and try to call the replaced method, the static method of the parent class is executed. So in most cases, it's a bad idea to replace a static method.

Virtual and dynamic methods can be used interchangeably. They differ only in their treatment by the compiler and runtime library. Delphi's help explains that dynamic methods have their implementation resolved at compile time and run slightly faster, whereas virtual methods are resolved at runtime, resulting in slightly slower access but a smaller compiled program. Virtual is usually the preferred declaration. Delphi's help suggests using dynamic when you have a base class with many descendants that may not override the method.

The inherited directive lets you refer back to a property or method as it was declared in the parent class. This is most often used in the implementation of an override method, to call the inherited method of the parent class and then supplement its behaviour.

1.7.5 Abstract Methods

Abstract is used in base classes to declare a method in the interface and defer its implementation to a descendant class. I.e. it defines an interface, but not the underlying operation. Abstract must be used with the virtual or dynamic directive. Abstract methods are never implemented in the base class and must be implemented in descendant classes to be used. A runtime error occurs if you try to execute an abstract method that is not overridden. Calling inherited within the override implementation of an abstract method will also result in a runtime error, since there is no inherited behaviour.

1.7.6 Messages

Delphi's handling of Windows messages is a special case of virtual methods. Message handlers are implemented in classes that descend from TControl. I.e classes that have a handle and can receive messages. Message handlers are always virtual and can be declared in the private part of a class interface, yet still allow the inherited method to be called. Inherited in a message handler just uses the keyword inherited, there is no need to supply the name of the method to call.

1.7.7 Events

Events are also an important characteristic of Delphi, since they let you delegate extensible behaviour to instances of a class. Events are properties that refer to a method of another object. Events are not inherited in Delphi 1; Delphi 2 extends this behaviour to let you use inherited in an event. . Inherited in an event handler just uses the keyword inherited, there is no need to supply the name of the method to call.

Events are particularly important to component developers, since they provide a hook for the user of the component to modify its behaviour in a way that may not be foreseen at the time the component is written.

1.7.8 Constructors and Destructors

The constructor and destructor are two special types of methods. The constructor initializes a class instance (allocates memory initialized to 0) and returns a reference (pointer) to the object. The destructor deallocates memory used by the object (but not the memory of other objects created by the object).

Classes descended from TObject have a static constructor, Create, and a virtual destructor Destroy.

TComponent introduces a new public property, the Owner of the component and this must be initialized in the constructor. TComponent's constructor is declared virtual, i.e. it can be overridden in descendant classes.It is essential when you override a virtual constructor or destructor in a TComponent descendant to include a call to the inherited method.

1.8 THE VCL TO APPLICATIONS DEVELOPERS

Applications Developers create complete applications by interacting with the Delphi visual environment (as mentioned earlier, this is a concept nonexistent in many other frameworks). These people use the VCL to create their user-interface and the other elements of their application: database connectivity, data validation, business rules, etc..

Applications Developers should know which properties, events, and methods each component makes available. Additionally, by understanding the VCL architecture, Applications Developers will be able to easily identify where they can improve their applications by extending components or creating new ones. Then they can maximize the capabilities of these components, and create better applications.

1.8.1 The VCL to Component Writers

Component Writers expand on the existing VCL, either by developing new components, or by increasing the functionality of existing ones. Many component writers make their components available for Applications Developers to use.

A Component Writer must take their knowledge of the VCL a step further than that of the Application Developer. For example, they must know whether to write a new component or to extend an existing one when the need for a certain characteristic arises. This requires a greater knowledge of the VCL's inner workings.

1.8.2 The VCL is made up of components

Components are the building blocks that developers use to design the user-interface and to provide some non-visual capabilities to their applications. To an Application Developer, a component is an object most commonly dragged from the Component palette and placed onto a form. Once on the form, one can manipulate the component's properties and add code to the component's various events to give the component a specific behavior. To a Component Writer, components are objects in Object Pascal code. Some components encapsulate the behavior of elements provided by the system, such as the standard Windows 95 controls. Other objects introduce entirely new visual or non-visual elements, in which case the component's code makes up the entire behavior of the component.

The complexity of different components varies widely. Some might be simple while others might encapsulate a elaborate task. There is no limit to what a component can do or be made up of. You can have a very simple component like a TLabel, or a much more complex component which encapsulates the complete functionality of a spreadsheet.

1.8.3 Component Types, structure, and VCL hierarchy

Components are really just special types of objects. In fact, a component's structure is based on the rules that apply to Object Pascal. There are three fundamental keys to understanding the VCL.

First, you should know the special characteristics of the four basic component types: standard controls, custom controls, graphical controls and non-visual components.

Second, you must understand the VCL structure with which components are built. This really ties into your understanding of Object Pascal's implementation. Third, you should be familiar with the VCL hierarchy and you should also know where the four component types previously mentioned fit into the VCL hierarchy. The following paragraphs will discuss each of these keys to understanding the VCL.

1.8.4 Component Types

As a component writer, there four primary types of components that you will work with in Delphi: standard controls, custom controls, graphical controls, and non-visual components. Although these component types are primarily of interest to component writers, it's not a bad idea for applications developers to be familiar with them. They are the foundations on which applications are built.

1.8.4.1 Standard Components

Some of the components provided by Delphi 2.0 encapsulate the behavior of the standard Windows controls: TButton, TListbox and Tedit, for example. You will find these components on the *Standard* page of the Component Palette. These components are Windows' common controls with Object Pascal wrappers around them.

Each standard component looks and works like the Windows' common control which it encapsulates. The VCL wrapper's simply makes the control available to you in the form of a Delphi component-it doesn't define the common control's appearance or functionality, but rather, surfaces the ability to modify a control's appearance/functionality in the form of methods and properties. If you have the VCL source code, you can examine how the VCL wraps these controls in the file STDCTRLS.PAS.

If you want to use these standard components unchanged, there is no need to understand how the VCL wraps them. If, however, you want to extend or change one of these components, then you must understand how the Window's common control is wrapped by the VCL into a Delphi component.

For example, the Windows class LISTBOX can display the list box items in multiple columns. This capability, however, isn't surfaced by Delphi's TListBox component (which encapsulates the Windows LISTBOX class). (TListBox only displays items in a single column.) Surfacing this capability requires that you override the default creation of the TListBox component.

This example also serves to illustrate why it is important for Applications Developers to understand the VCL. Just knowing this tidbit of information helps you to identify where enhancements to the existing library of components can help make your life easier and more productive.

1.8.4.2 Custom components

Unlike standard components, custom components are controls that don't already have a method for displaying themselves, nor do they have a defined behavior. The Component Writer must provide to code that tells the component how to draw itself and determines how the component behaves when the user interacts with it. Examples of existing custom components are the TPanel and TStringGrid components.

It should be mentioned here that both standard and custom components are *windowed* controls. A "windowed control" has a window associated with it and, therefore, has a window handle. Windowed controls have three characteristics: they can receive the input focus, they use system resources, and they can be parents to other controls. (Parents is related to containership, discussed later in this paper.) An example of a component which can be a container is the TPanel component.

1.8.4.3 Graphical components

Graphical components are visual controls which cannot receive the input focus from the user. They are non-windowed controls. Graphical components allow you to display something to the user without using up any system resources; they have less "overhead" than standard or custom components. Graphical components don't require a window handle-thus, they cannot can't get focus. Some examples of graphical components are the TLabel and TShape components.

Graphical components cannot be containers of other components. This means that they cannot own other components which are placed on top of them.

1.8.4.4 Non-visual components

Non-visual components are components that do not appear on the form as controls at run-time. These components allow you to encapsulate some functionality of an entity within an object. You can manipulate how the component will behave, at design-time, through the Object Inspector. Using the Object Inspector, you can modify a non-visual component's properties and provide event handlers for its events. Examples of such components are the TOpenDialog, TTable, and TTimer components.

1.8.4.5 Structure of a component

All components share a similar structure. Each component consists of common elements that allow developers to manipulate its appearance and function via properties, methods and events. The following sections in this paper will discuss these common elements as well as talk about a few other characteristics of components which don't apply to all components.

1.8.4.6 Component properties

Properties provide an extension of an object's fields. Unlike fields, properties do not store data: they provide other capabilities. For example, properties may use methods to read or write data to an object field to which the user has no access. This adds a certain level of protection as to how a given field is assigned data. Properties also cause "side effects" to occur when the user makes a particular assignment to the property. Thus what appears as a simple field assignment to the component user could trigger a complex operation to occur behind the scenes.

1.9 PROPERTIES PROVIDE ACCESS TO INTERNAL STORAGE FIELDS

There are two ways that properties provide access to internal storage fields of components: directly or through access methods. Examine the code below which illustrates this process.

```
TCustomEdit = class(TWinControl)
private
 FMaxLength: Integer;
protected
 procedure SetMaxLength(Value: Integer);
```

```
...
published
  property MaxLength: Integer read
     FMaxLength write SetMaxLength default 0;
...
end;
```

The code above is snippet of the TCustomEdit component class. TCustomEdit is the base class for edit boxes and memo components such as TEdit, and TMemo.

TCustomEdit has an internal field FMaxLength of type Integer which specifies the maximum length of characters which the user can enter into the control. The user doesn't directly access the FMaxLength field to specify this value. Instead, a value is added to this field by making an assignment to the MaxLength property.

The property MaxLength provides the access to the storage field FMaxLength. The property definition is comprised of the property name, the property type, a read declaration, a write declaration and optional default value.

The read declaration specifies how the property is used to read the value of an internal storage field. For instance, the MaxLength property has direct read access to FMaxLength. The write declaration for MaxLength shows that assignments made to the MaxLength property result in a call to an *access method* which is responsible for assigning a value to the FMaxLength storage field. This access method is SetMaxLength.

1.9.1 Property-access methods

Access methods take a single parameter of the same type as the property. One of the primary reasons for write access methods is to cause some side-effect to occur as a result of an assignment to a property. Write access methods also provide a method layer over assignments made to a component's fields. Instead of the component user making the assignment to the field directly, the property's write access method will assign the value to the storage field if the property refers to a particular storage field. For example, examine the implementation of the SetMaxLength method below.

```
procedure TCustomEdit.SetMaxLength(Value: Integer);

begin

 if FMaxLength <> Value then

 begin

  FMaxLength := Value;

  if HandleAllocated then

     SendMessage(Handle, EM_LIMITTEXT, Value, 0);

 end;

end;
```

The code in the SetMaxLength method checks if the user is assigning the same value as that which the property already holds. This is done as a simple optimization. The method then assigns the new value to the internal storage field, FMaxLength. Additionally, the method then sends an EM_LIMITTEXT Windows message to the window which the TCustomEdit encapsulates. The EM_LIMITTEXT message places a limit on the amount of text that a user can enter into an edit control. This last step is what is referred to as a *side-effect* when assigning property values. Side effects are any additional actions that occur when assigning a value to a property and can be quite sophisticated.

Providing access to internal storage fields through property access methods offers the advantage that the Component Writer can modify the implementation of a class without modifying the interface. It is also possible to have access methods for the read access of a property. The read access method might, for example, return a type which is different that that of a properties storage field. For instance, it could return the string representation of an integer storage field.

Another fundamental reason for properties is that properties are accessible for modification at run-time through Delphi's Object Inspector. This occurs whenever the declaration of the property appears in the published section of a component's declaration.

1.9.2 Types of properties

Properties can be of the standard data types defined by the Object Pascal rules. Property types also determine how they are edited in Delphi's Object Inspector. The table below shows the different property types as they are defined in Delphi's online help.

Property type	Object Inspector treatment
Simple	Numeric, character, and string properties appear in the Object Inspector as numbers, characters, and strings, respectively. The user can type and edit the value of the property directly.
Enumerated	Properties of enumerated types (including Boolean) display the value as defined in the source code. The user can cycle through the possible values by double-clicking the value column. There is also a drop-down list that shows all possible values of the enumerated type.
Set	Properties of set types appear in the Object Inspector looking like a set. By expanding the set, the user can treat each element of the set as a Boolean value: True if the element is included in the set or False if it's not included.
Object	Properties that are themselves objects often have their own property editors. However, if the object that is a property also has published properties, the Object Inspector allows the user to expand the list of object properties and edit them individually. Object properties must descend from TPersistent.
Array	Array properties must have their own property editors. The Object Inspector has no built-in support for editing array properties.

For more information on properties, refer to the "Component Writers Guide" which ships with Delphi.

1.9.3 Methods

Since components are really just objects, they can have methods. We will discuss some of the more commonly used methods later in this paper when we discuss the different levels of the VCL hierarchy.

1.9.4 Events

Events provide a means for a component to notify the user of some pre-defined occurrence within the component. Such an occurrence might be a button click or the pressing of a key on a keyboard.

Components contain special properties called events to which the component user assigns code. This code will be executed whenever a certain event occurs. For instance, if you look at the events page of a TEdit component, you'll see such events as OnChange, OnClick and OnDblClick. These events are nothing more than pointers to methods.

When the user of a component assigns code to one of those events, the user's code is referred to as an event handler. For example, by double clicking on the events page for a particular event causes Delphi to generate a method and places you in the Code Editor where you can add your code for that method. An example of this is shown in the code below, which is an OnClick event for a TButton component.

It becomes clearer that events are method pointers when you assign an event handler to an event programmatically. The above example was Delphi generated code. To link your own an event handler to a TButton's OnClick event at run time you must first create a method that you will assign to this event. Since this is a method, it must belong to an existing object. This object can be the form which owns the TButton component although it doesn't have to be. In fact, the event handlers which Delphi creates belong to the form on which the component resides. The code below illustrates how you would create an event handler method.

When you define methods for event handlers, these methods must be defined as the same type as the event property and the field to which the event property refers. For instance, the OnClick event refers to an internal data field, FOnClick. Both the property OnClick, and field FOnClick are of the type TNotifyEvent. TNotifyEvent is a procedural type as shown below:

```
TNotifyEvent = procedure (Sender: TObject) of object;
```

Note the use of the of object specification. This tells the compiler that the procedure definition is actually a method and performs some additional logic like ensuring that an implicit Self parameter is also passed to this method when called. Self is just a pointer reference to the class to which a method belongs.

1.9.5 Containership

Some components in the VCL can own other components as well as be parents to other components. These two concepts have a different meaning as will be discussed in the section to follow.

1.9.6 Ownership

All components may be owned by other components but not all components can own other components. A component's Owner property contains a reference to the component which owns it.

The basic responsibility of the owner is one of resource management. The owner is responsible for freeing those components which it owns whenever it is destroyed. Typically, the form owns all components which appear on it, even if those components are placed on another component such as a TPanel. At design-time, the form automatically becomes the owner for components which you place on it. At run-time, when you create a component, you pass the owner as a parameter to the component's constructor. For instance, the code below shows how to create a TButton component at run-time and passes the form's implicit Self variable to the TButton's Create constructor. TButton.Create will then assign whatever is passed to it, in this case Self or rather the form, and assign it to the button's Owner property.

```
MyButton := TButton.Create(self);
```

When the form that now owns this TButton component gets freed, MyButton will also be freed.

You can create a component without an owner by passing nil to the component's Create constructor, however, you must ensure that the component is freed when it is no longer needed. The code below shows you how to do this for a TTable component.

1.9.7 Parenthood

Parenthood is a much different concept from ownership. It applies only to windowed components, which can be parents to other components. Later, when we discuss the VCL hierarchy, you will see the level in the hierarchy which introduces windowed controls.

Parent components are responsible for the display of other components. They call the appropriate methods internally that cause the children components to draw themselves. The Parent property of a component refers to the component which is its parent. Also, a component's parent does not have to be it's owner. Although the parent component is mainly responsible for the display of components, it also frees children components when it is destroyed.

Windowed components are controls which are visible user interface elements such as edit controls, list boxes and memo controls. In order for a windowed component to be displayed, it must be assigned a parent on which to display itself. This task is done automatically by Delphi's design-time environment when you drop a component from the Component Palette onto your form.

CHAPTER 2

DATABASE

Every thing around us has a particular identity. To identify anything system, actor or person in words we need a data or information. So this information is valuable and in this advanced era we can store it in database and access this data by the blink of eye.

For an instant if we go through the definitions of database we may find following definitions.

A database is a collection of related information.

A database is an organized body of related information.

2.1 DEMERITS OF ABSENCE OF DATABASE

A glance on the past will may help us to reveal the drawbacks in case of absence of database.

In the past when there wasn't proper system of database, Much paper work was need to do and to handle great deal of written paper documentation was giant among the problems itself.

In the huge networks to deal with equally bulky data, more workers are needed which affidavit cost much labor expanses.

The old criteria for saving data and making identification was much time consuming such as if we want to search the particular data of a person.

Before the Development of Computer database it was a great problem to search for some thing. Efforts to avoid the headache of search often results in new establishments of data.

Before the development of database it seemed very unsafe to keep the worthy information. In Some situation some big organization had to employee the special persons in order to secure the data.

Before the implementation of database any firm had to face the plenty of difficulties in order to maintain their Management. To hold the check on the expenses of the firm, the manager faced difficulties.

2.2 MERITS OF DATABASE

The modern era is known as the golden age computer sciences and technology. In a simple phrase we can express that the modern age is built on the foundation of database.

If we carefully watch our daily life we can examine that some how our daily life is being connected with database.

There are several benefits of database developments.

Now with the help of computerized database we can access data in a second.

By the development of the database we can make data more secure.

By the development of database we can reduce the cost.

2.3 DATABASE DESIGN

The design of a database has to do with the way data is stored and how that data is related. The design process is performed after you determine exactly what information needs to be stored and how it is to be retrieved.

A collection of programs that enables you to store, modify, and extract information from a database. There are many different types of DBMS ranging from small systems that run on personal computers to huge systems that run on mainframes. The following are examples of database applications:

Computerized library systems

Automated teller machines

Flight reservation systems

Computerized parts inventory systems

From a technical standpoint, DBMS can differ widely. The terms relational, network, flat, and hierarchical all refer to the way a DBMS organizes information internally. The internal organization can affect how quickly and flexibly you can extract information.

Requests for information from a database are made in the form of a query.

Database design is a complex subject. A properly designed database is a model of a business, Country Database or some other in the real world. Like their physical model counterparts, data models enable you to get answers about the facts that make up the objects being modeled. It's the questions that need answers that determine which facts need to be stored in the data model.

In the relational model, data is organized in tables that have the following characteristics: every record has the same number of facts, every field contains the same type of facts (Data) in each record, and there is only one entry for each fact. No two records are exactly the same.

The more carefully you design, the better the physical database meets users' needs. In the process of designing a complete system, you must consider user needs from a variety of viewpoints.

2.4 DATABASE MODELS

Various techniques are used to model data structures. Certain models are more easily implemented by some types of database management systems than others. For any one logical model various physical implementation may be possible. An example of this is the relational model: in larger systems the physical implementation often has indexes which point to the data; this is similar to some aspects of common implementations of the network model. But in small relational database the data is often stored in a set of files, one per table, in a flat, un-indexed structure. There is some confusion below and elsewhere in this article as to logical data model vs. its physical implementation.

2.4.1 Flat Model

The flat (or table) model consists of a single, two dimensional array of data elements, where all members of a given column are assumed to be similar values, and all members of a row are assumed to be related to one another. For instance, columns for name and password might be used as a part of a system security database. Each row would have the specific password associated with a specific user. Columns of the table often have a type associated with them, defining them as character data, date or time information, integers, or floating point numbers. This model is the basis of the spreadsheet.

2.4.2 Network Model

The network model allows multiple datasets to be used together through the use of pointers (or references). Some columns contain pointers to different tables instead of data. Thus, the tables are related by references, which can be viewed as a network structure. A particular subset of the network model, the hierarchical model, limits the relationships to a tree structure, instead of the more general directed graph structure implied by the full network model.

2.4.3 Relational Model

The relational data model was introduced in an academic paper by E.F. Cod in 1970 as a way to make database management systems more independent of any particular application. It is a mathematical model defined in terms of predicate logic and set theory.

Although the basic idea of a relational database has been very popular, relatively few people understand the mathematical definition and only a few obscure DBMSs implement it completely and without extension. Oracle, for example, can be used in a purely relational way, but it also allow tables to be defined that allow duplicate rows an extension (or violation) of the relational model. In common English usage, a DBMS is called relational if it supports relational operational operations, regardless of whether it enforces strict adherence to the relational model. The following is an informal, not-technical explanation of how "relational" database management systems commonly work.

A relational database contains multiple tables, each similar to the one in the "flat" database model. However, unlike network databases, the tables are not linked by pointers. Instead, keys are used to match up rows of data in different tables. A key is just one or more columns in one table that correspond to columns in other tables. Any column can be a key, or multiple columns can be grouped together into a single key. Unlike pointers, it's not necessary to define all the keys in advance; a column can be used as a key even if it wasn't originally intended to be one.

A key that can be used to uniquely identify a row in a table is called a unique key. Typically one of the unique keys is the preferred way to refer to row; this is defined as the table's primary key.

When a key consists of data that has an external, real-world meaning (such as a person's name, a book's ISBN, or a car's serial number), it's called a "natural" key. If no nature key is suitable, an arbitrary key can be assigned (such as by given employees ID numbers). In practice, most databases have both generated and natural keys, because generated keys can be used internally to create links between rows that can't break, while natural keys can be used, less reliably, for searches and for integration with other databases. (For example, records in two independently developed databases could be matched up by social security number, except when the social security numbers are incorrect, missing, or have changed).

2.4.3.1 Why we use a Relational Database Design

Maintaining a simple, so-called flat database consisting of a single table doesn't require much knowledge of database theory. On the other hand, most database worth maintaining are quite a bit more complicated than that. Real life databases often have hundreds of thousands or even millions of records, with data that are very intricately related. This is where using a full-fledged relational database program becomes essential. Consider, for example, the Library of Congress, which has over 16 million books in its collection. For reasons that will become apparent soon, a single table simply will not do for this database.

2.5 RELATIONSHIPS BETWEEN TABLES

When you create tables for an application, you should also consider the relationships between them. These relationships give a relational database much of its power. There are three types of relationships between tables: one-to-one, one-to-many and many-to-many relationships.

2.5.1 One-To-One Relationships

In a one-to-one relationship, each record in one table corresponds to a single record in a second table. This relationship is not very common, but it can offer several benefits. First, you can put the fields from both tables into a single, combined table. One reason for using two tables is that each field is a property of a separate entity, such as owner operators and their tracks. Each operator can operate just one truck at a time, but the fields for the operator and truck tables refer to different entities.

A one-to-one relationship can also reduce the time needed to open a large table by placing some of the table's columns in a second, separate table. This approach makes particular sense when a table has some fields that are used infrequently. Finally, a one-to-one relationship can support in a table requires security, placing them in a separate table lets your application restrict to certain fields. Your application can link the restricted table back to the main table via a one-to-one relationship so that people with proper permissions can edit, delete, and add new records to these fields.

2.5.2 One-To-Many Relationships

A one-to-many relationship, in which a row from one table corresponds to one or more rows from a second table, is more common. This kind of relationship can form the basis for a Many-To-Many relationship as well.

2.6 DATA MODELING

In information system design, data modeling is the analysis and design of the information in the system, concentrating on the logical entities and the logical dependencies between these entities. Data modeling is an abstraction activity in that the details of the values of individual data observations are ignored in favor of the structure, relationships, names and formats of the data of interest, although a list of valid values is frequently recorded. It is by the data model that definitions of what the data means is related to the data structures.

While a common term for this activity is "Data Analysis" the activity actually has more in common with the ideas and methods of synthesis (putting things together), than it does in the original meaning of the term analysis (taking things apart). This is because the activity strives to bring the data structures of interest together in a cohesive, inseparable, whole by eliminating unnecessary data redundancies and relating data structures by relationships.

2.6.1 Database Normalization

Database normalization is a series of steps followed to obtain a database design that allows for consistent storage and efficient access of data in a relational database. These steps reduce data redundancy and the risk of data becoming inconsistent.

However, many relational DBMS lack sufficient separation between the logical database design and the physical implementation of the data store, such that queries against a fully normalized database often perform poorly. In this case de-normalizations are sometimes used to improve performance, at the cost of reduced consistency.

2.6.2 Primary Key

In database design, a primary key is a value that can be used to identify a particular row in a table. Attributes are associated with it. Examples are names in a telephone book (to look up telephone numbers), words in a dictionary (to look up definitions) and Dewey Decimal Numbers (to look up books in a library).

In the relational model of data, a primary key is a candidate key chosen as the main method of uniquely identifying a relation. Practical telephone books, dictionaries and libraries can not use names, words or Dewey Decimal System Numbers as candidate keys because they do not uniquely identify telephone numbers, word definitions or books. In some design situations it is impossible to find a natural key that uniquely identifies a relation. A surrogate key can be used as the primary key. In other situations there may be more than one candidate key for a relation, and no candidate key is obviously preferred. A surrogate key may be used as the primary key to avoid giving one candidate key artificial primacy over the others. In addition to the requirement that the primary key be a candidate key, there are several other factors which may make a particular choice of key better than others for a given relation.

The primary key should generally be short to minimize the amount of data that needs to be stored by other relations that reference it. A compound key is usually not appropriate. (However, this is a design consideration, and some database management systems may be better than others in this regard.)

The primary key should be immutable, meaning its value should not be changed during the course of normal operations of the database. (Recall that a primary key is the means of uniquely identifying a tuple, and that identity by definition, never changes.) This avoids the problem of dangling references or orphan records created by other relations referring to a tuple whose primary key has changed. If the primary key is immutable, this can never happen.

2.6.3 Foreign Key

A foreign key (FK) is a field in a database record under one primary key that points to a key field of another database record in another table where the foreign key of one table refers to the primary key of the other table. This way references can be made to link information together and it is an essential part of database normalization.

For example, a person sending an e-mail needs not to include the entire text of a book in the e-mail. Instead, they can include the ISBN of the book, and interested persons can then use the number to get information about the book, or even the book itself. The ISBN is the primary key of the book, and it is used as a foreign key in the e-mail.

Note that using a foreign key often assumes its existence as a primary key somewhere else. Improper foreign key/primary key relationships are the source of many database problems.

2.6.4 Compound Key

In database design, a compound key (also called a composite key) is a key that consists on 2 or more attributes.

No restriction is applied to the attribute regarding their (initial) ownership within the data model. This means that any one, none or all, of the multiple attributes within the compound key can be foreign keys. Indeed, a foreign key may, itself, be a compound key.

Compound keys almost always originate from attributive or associative entities (tables) within the model, but this is not an absolute value.

CHAPTER THREE

Software Interfaces And Their Usage

3.1 User Login

This screen is user login form. Users can enter their user name and password for enterance of the main program.

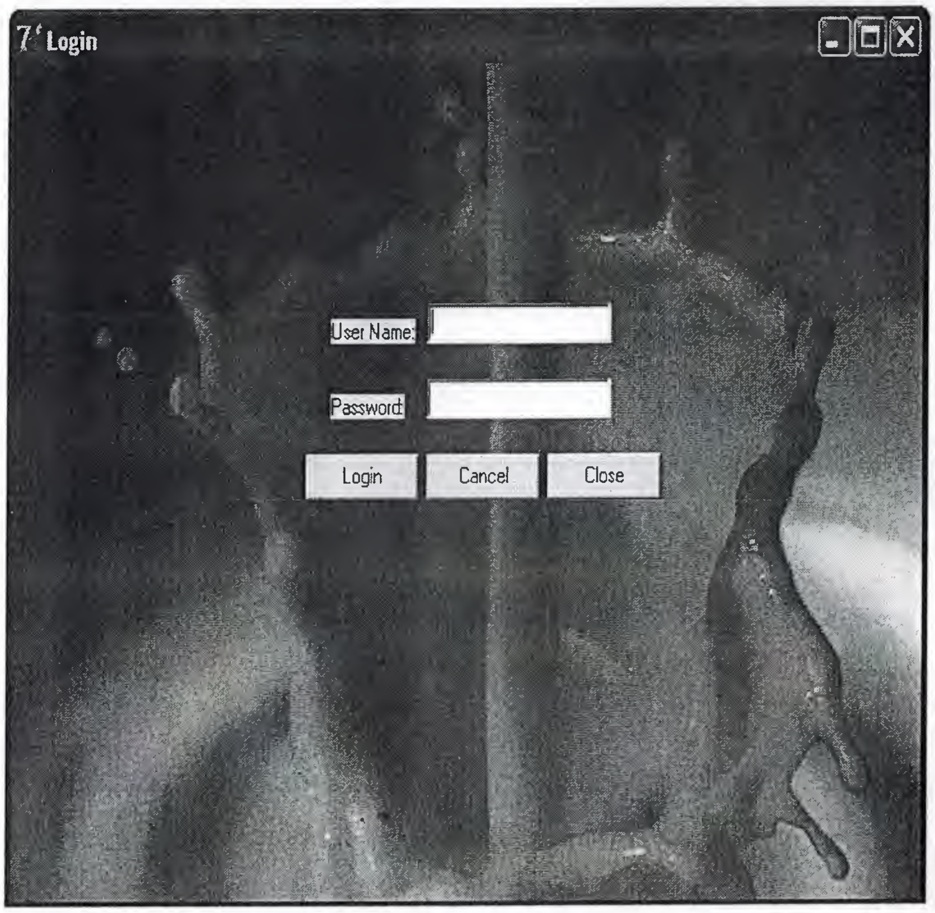
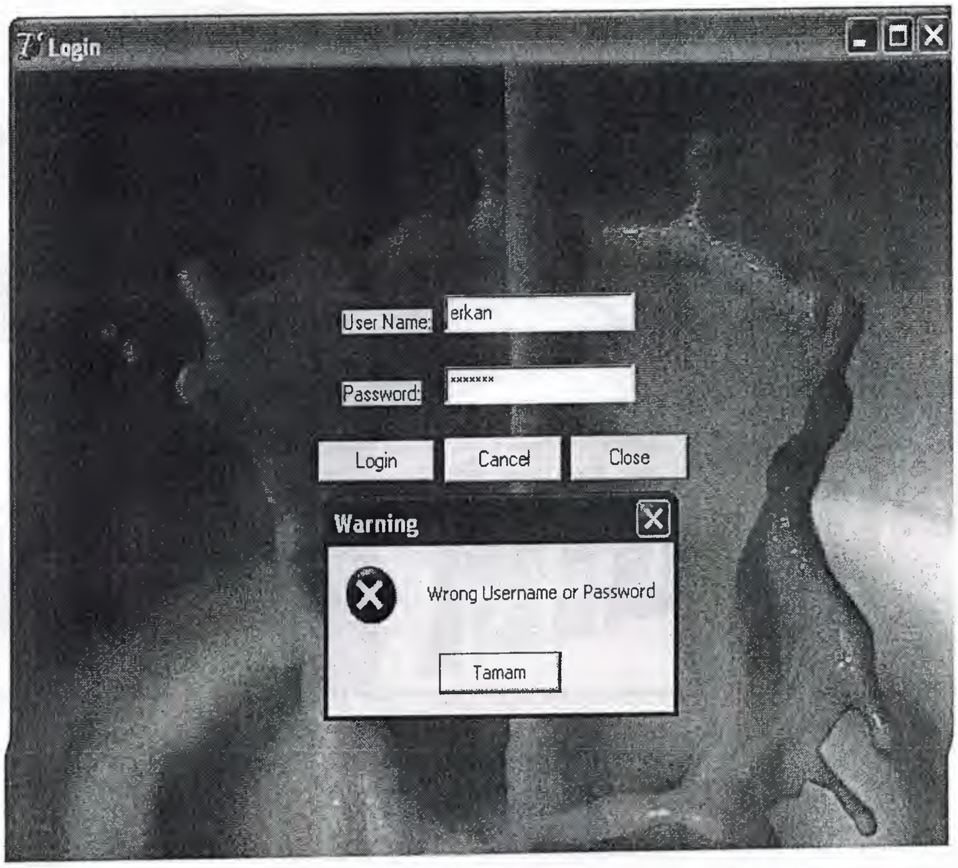


Figure 3.1.1 Login Form

As you see at above; in login panel , there are three buttons on the panel which first buton is Login button for entering to the program, the second one is for canceling the entrennce and third one is close button for closing to the program.

If the user entered to wrong password or user name program auttomatically warn to user and show a message on the screen likes " wrong password or user ".(see Figure 1.2)



Wrong Username or Password

Figure 3.1.2 Login Form

3.2 Main Window And Functions

The Milk Stock Program form is the main window that user can see all functions of the software. At above of the form you will see main menu and some submenus (in main menu), the missions of the main menu is for quick access to other forms. And you will see one panel on the right side of above the form which shows current time and date.

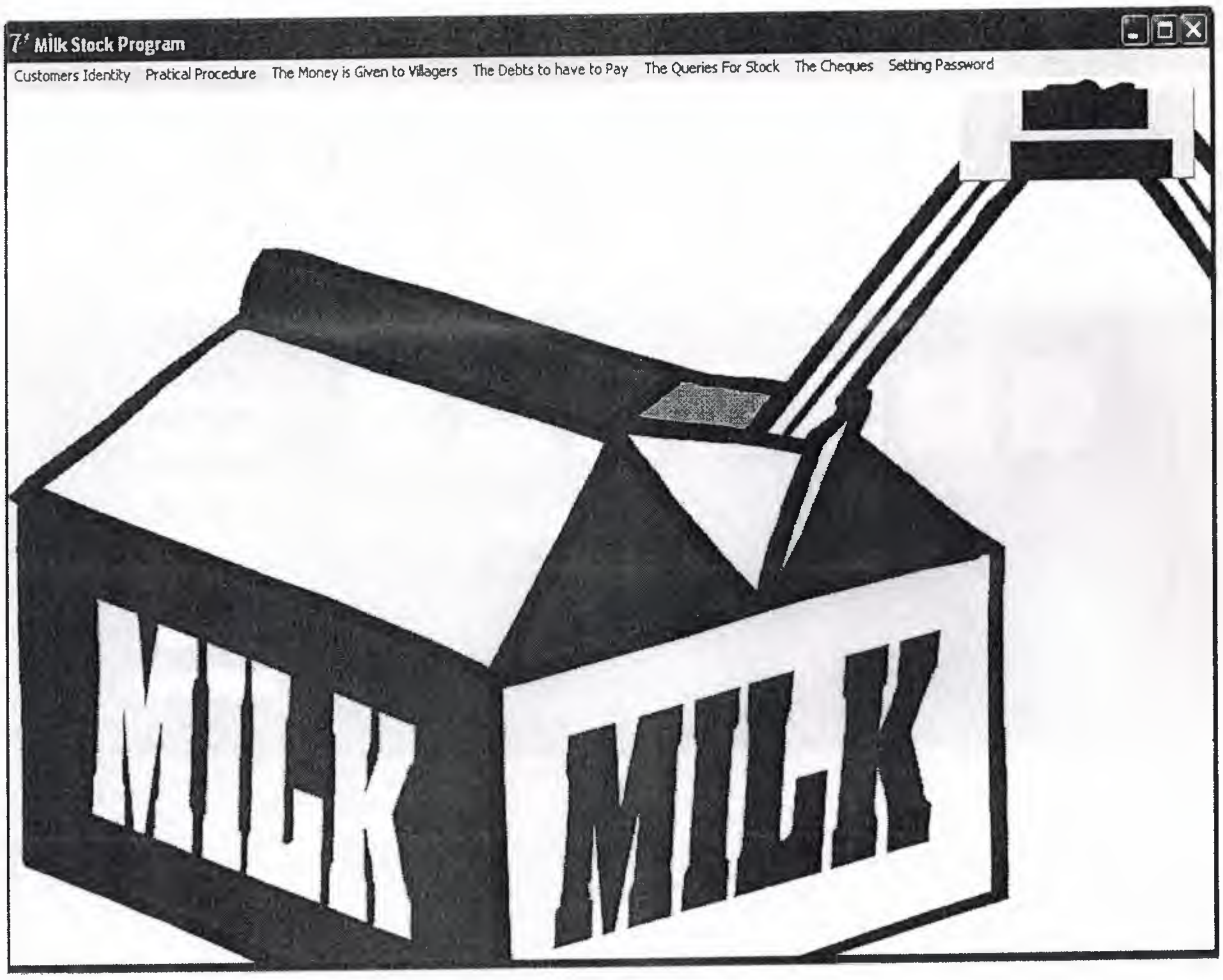


Figure 3.2.1 Milk Stock Main Form

3.2.1 Opening To Customer Identity And Submenu's The Records

As you see at the bottom figure 3.2.1 There are some menu's items and if the user clicks the first item (Customer Identity) then a submenu will open called (The Records) .

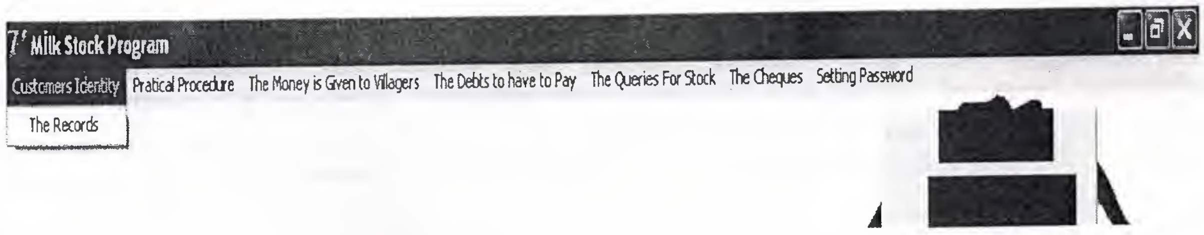
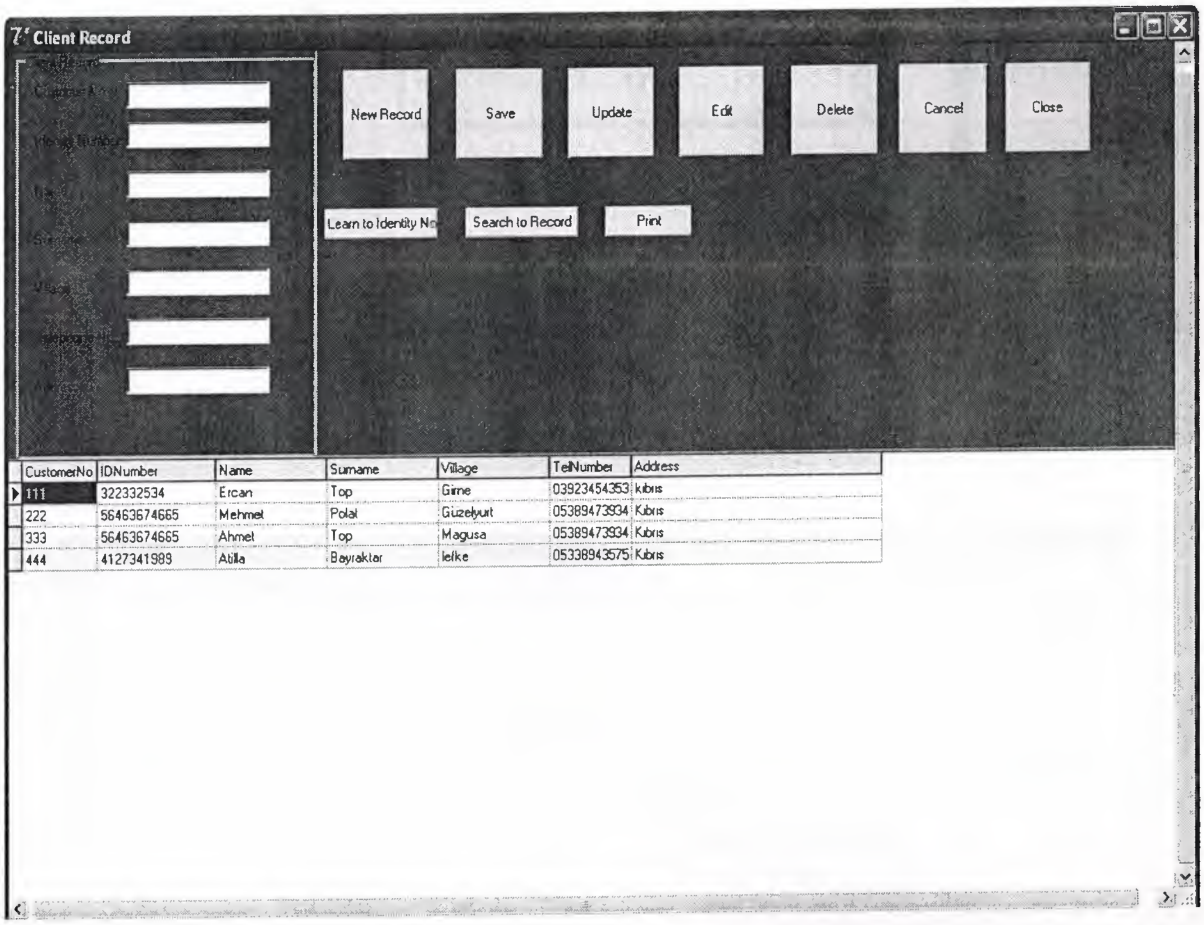


Figure 3.2.1 Milk Stock Menu

3.2.2 Client Record

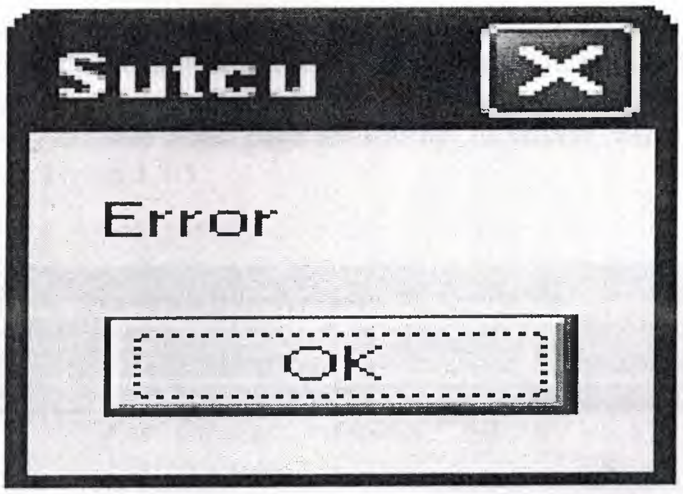
When the user clicks to submenu (The Records) , A window will open which is called Client Record (see figure 3.2.2) .



CustomerNo	IDNumber	Name	Surname	Village	TelNumber	Address
111	322332534	Ercan	Top	Girne	03923454353	kıbrıs
222	56463674665	Mehmet	Polat	Güzelyurt	05389473934	Kıbrıs
333	56463674665	Ahmet	Top	Magusa	05389473934	Kıbrıs
444	4127341989	Atilla	Bayraktar	lefke	05338943575	Kıbrıs

Figure 3.2.2 Client Record Form

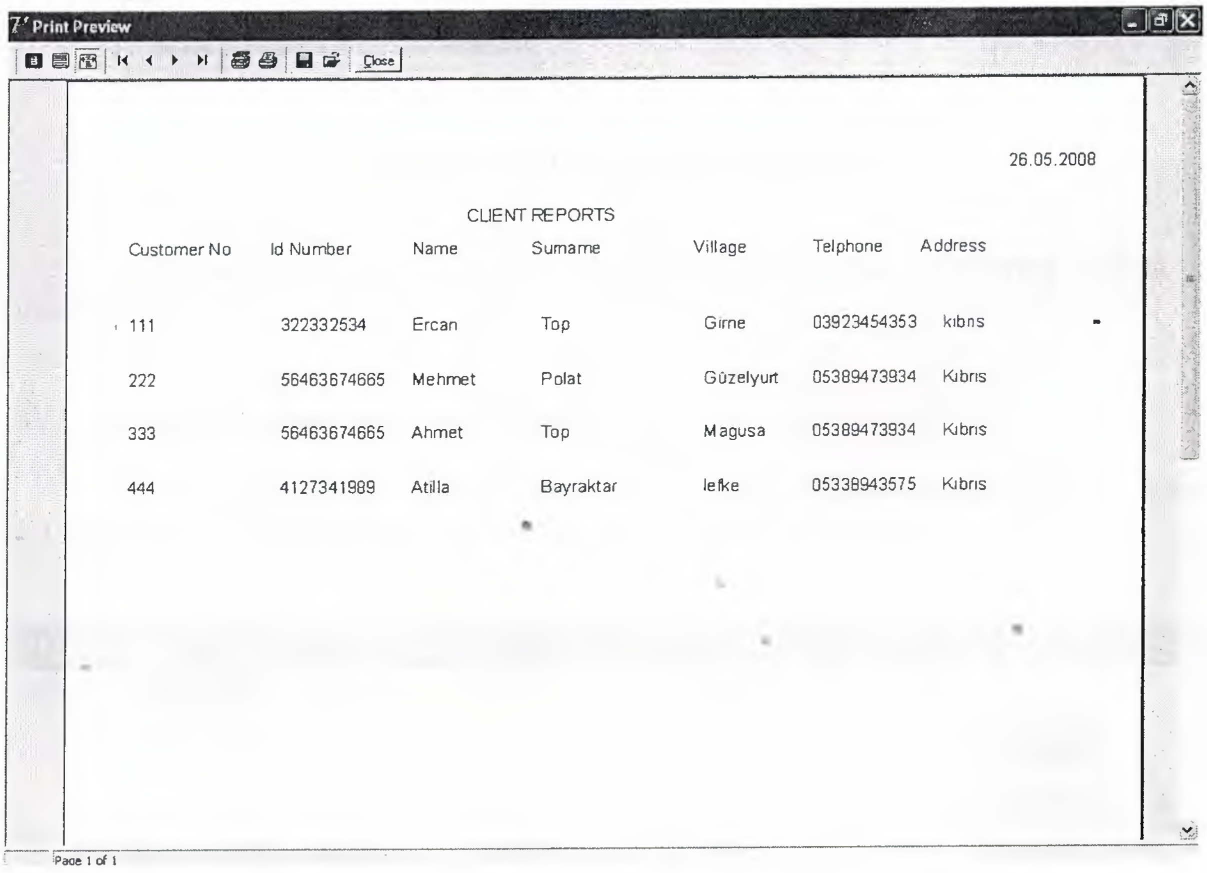
After clicking to The Records you will see this screen which is the above . In the Client Record form , user can enter villager or customer records like CustomerNo, Identity Number, Name, Surname, Village , Telephone Number, Address. If user want to add a client record first of all user presses to new record button then who will enter the identities of the customer after finishing to his writting who will press to Save button. But there is an important point here to be attention if the same CustomerNo exist in the database . The program will give an warning error to user.
Because of the same CustomerNo exist in database.(see figure 3.3.3)



Error

Figure 3.2.3 Erorr Form

And also there are some button on the right side which user can update records or edit,delete,cancel,close the program or print the all records for using these buttons.



26.05.2008

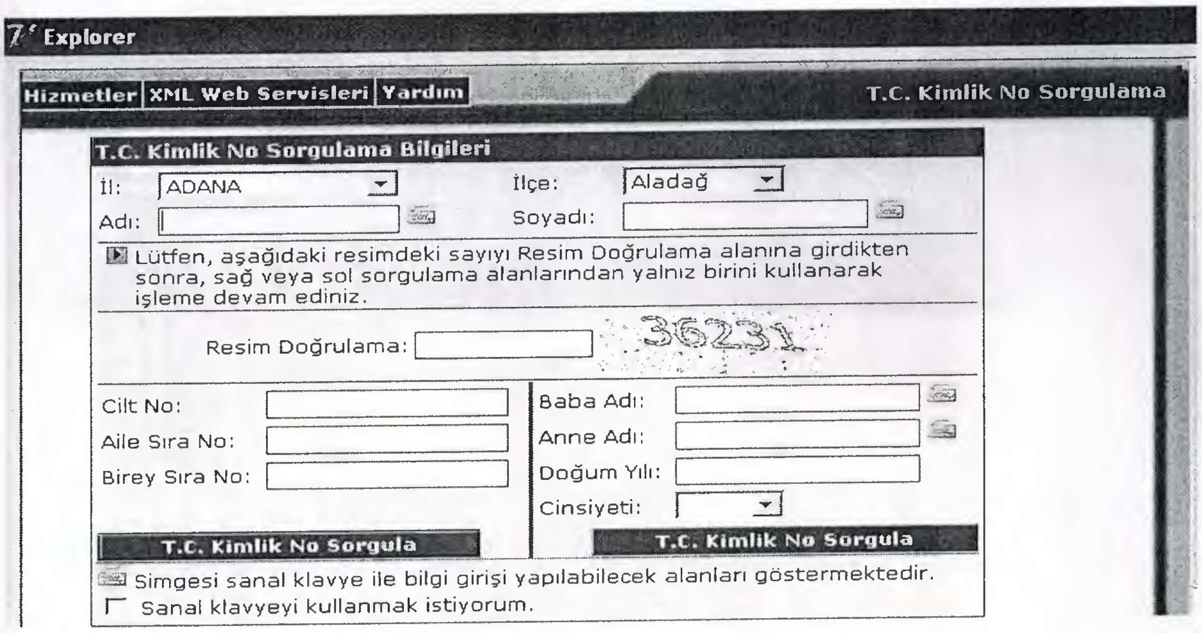
CLIENT REPORTS

Customer No	Id Number	Name	Sumame	Village	Telphone	Address
111	322332534	Ercan	Top	Girne	03923454353	kıbrıs
222	56463674665	Mehmet	Polat	Gûzelyurt	05389473934	Kıbrıs
333	56463674665	Ahmet	Top	Magusa	05389473934	Kıbrıs
444	4127341989	Atilla	Bayraktar	lefke	05338943575	Kıbrıs

Figure 3.2.4 Client Report

At the above you see client reports, you can take print out of the client information.

There are many facillities to included in Milk Stock Program, for example user can learn to customers Identity Number to press to button is called (Learn to Identity No) , If user press to “Learn to Identity No” button which program automatically connect to goverment web XML page for learing to village Identity Number ,see at the bellow shown Figere 3.3.5


Explorer

Hizmetler | XML Web Servisleri | Yardım — T.C. Kimlik No Sorgulama

T.C. Kimlik No Sorgulama Bilgileri

İl: ADANA — İlçe: Aladağ

Adı: — Soyadı:

Lütfen, aşağıdaki resimdeki sayıyı Resim Doğrulama alanına girdikten sonra, sağ veya sol sorgulama alanlarından yalnız birini kullanarak işleme devam ediniz.

Resim Doğrulama: 36231

Cilt No: — Baba Adı:

Aile Sıra No: — Anne Adı:

Birey Sıra No: — Doğum Yılı:

Cinsiyeti:

T.C. Kimlik No Sorgula — T.C. Kimlik No Sorgula

Simgesi sanal klavye ile bilgi girişi yapılabilecek alanları göstermektedir.

Sanal klavyeyi kullanmak istiyorum.

Figure 3.2.5 Goverment Web Page

At this web page user can learn to customer Identity Number for entering to client information.

3.3 Pratical Procedure And Milk Purchasing

When the user cliks to Pratical Procedure on the main window which shows at Figure 3.3.1 then user will see the Milk Purchasing item (see at Figure 3.3.1) .

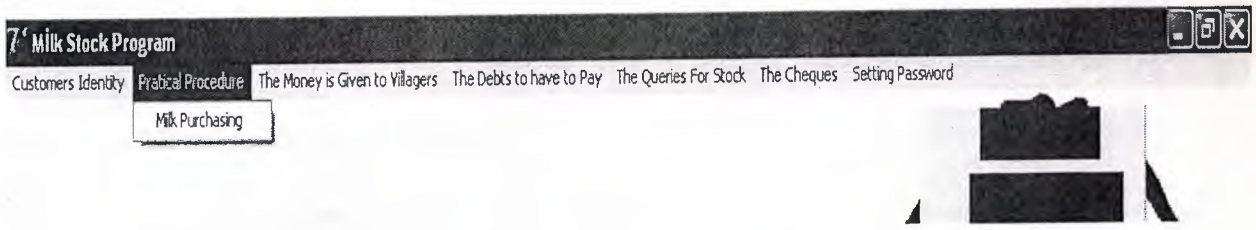
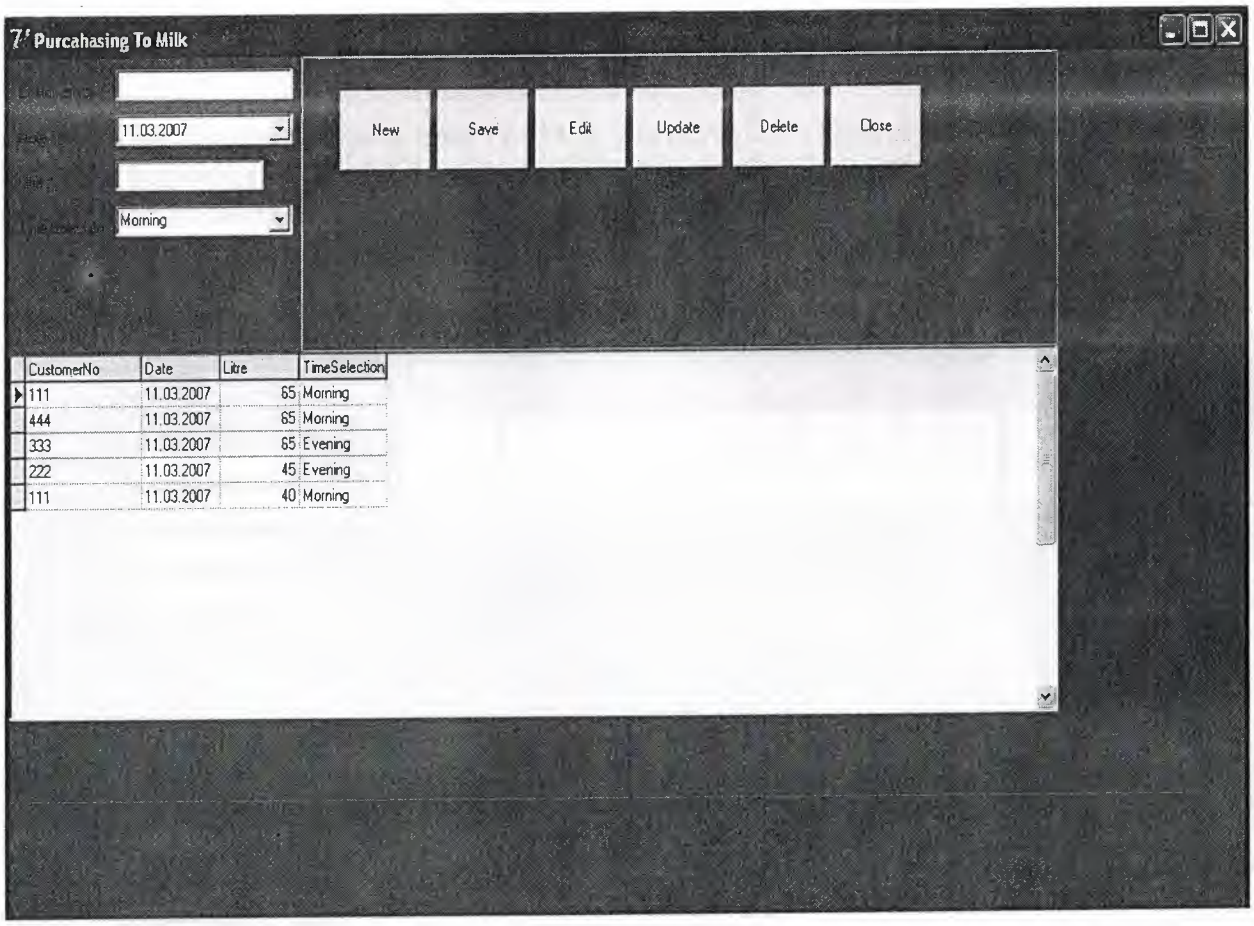


Figure 3.3.1 Milk Stock Menu

After clicking the submenu item Milk Purchasing user will see this screen at below(see Figure 3.3.2) and here user can enter the daily milk litre in stock by collecting from the villagers.



11.03.2007

Morning

	CustomerNo	Date	Litre	TimeSelection
▶	111	11.03.2007	65	Morning
	444	11.03.2007	65	Morning
	333	11.03.2007	65	Evening
	222	11.03.2007	45	Evening
	111	11.03.2007	40	Morning

Figure 3.3.2 Entering Daily Purchasing Form

In this form user can enter the CustomerNo ,Date,Litre and select the Time Selection daily for the stock. For saving, editing, updating, deleting, closing of the stock information user uses some buttons on the top of the right side(see at Figure 3.3.2).

3.4 The Money is Given To Villagers

At this form when the user cliks to The Money is Given To Villagers item on the main window which shows at Figure 3.4.1 then user will see the Milk Purchasing item (see at *Figure 3.4.2)* .

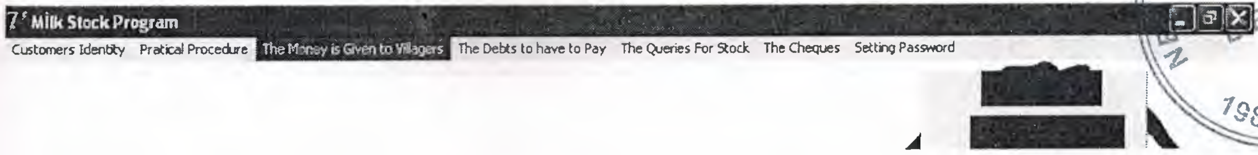
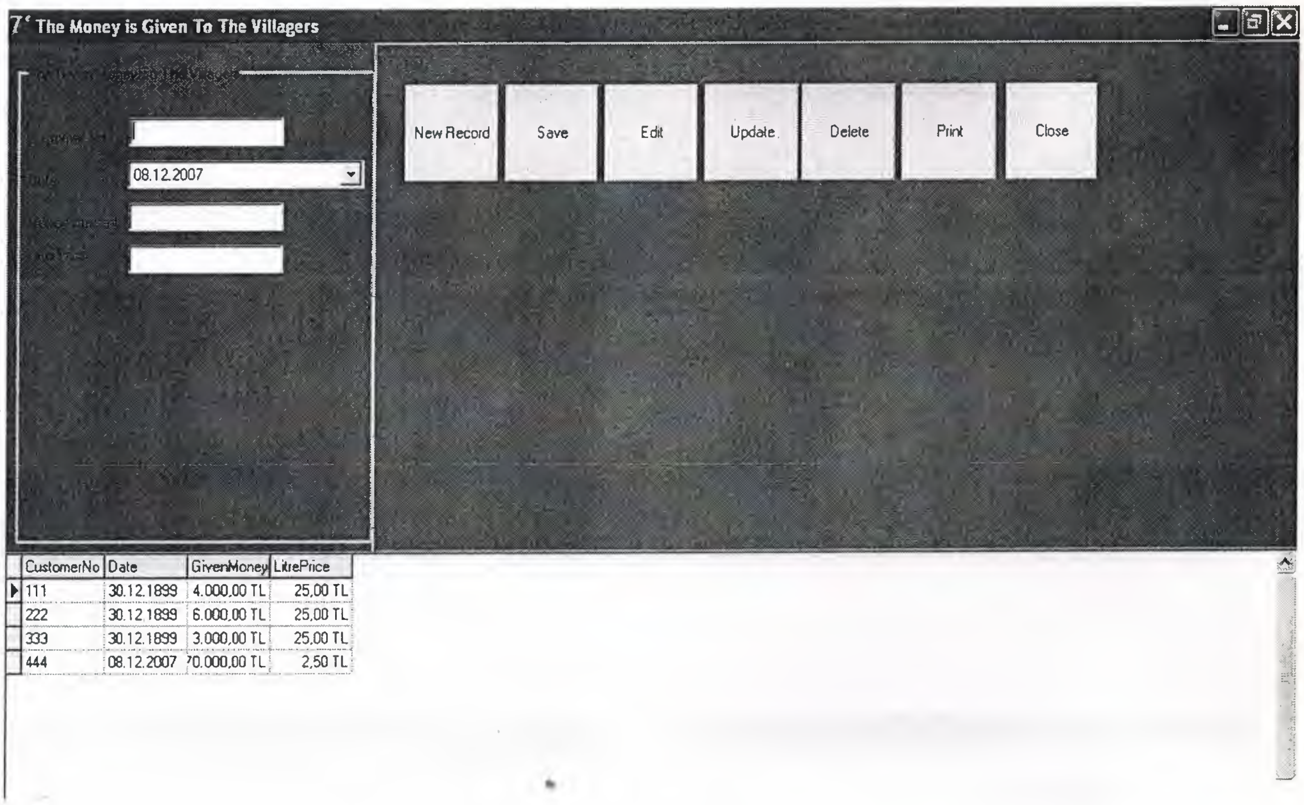


Figure 3.4.1 Milk Stock Menu

After clicking the menu item The Money is Given To Villagers user will see this screen at below(see Figure 3.4.2) and in this field user can enter the Money amount which is given to the villagers and litre price after bargaining with villagers.

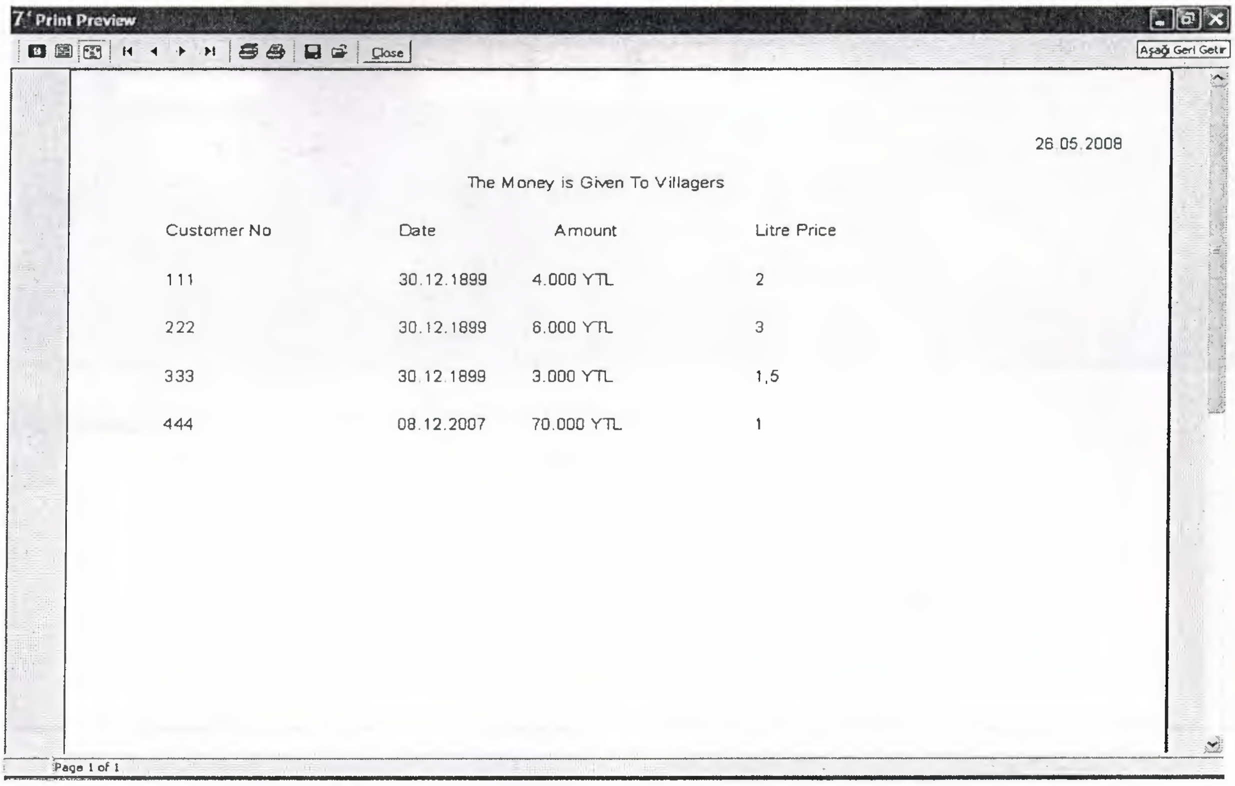


Date: 08.12.2007

	CustomerNo	Date	GivenMoney	LitrePice
▶	111	30.12.1899	4.000,00 TL	25,00 TL
	222	30.12.1899	6.000,00 TL	25,00 TL
	333	30.12.1899	3.000,00 TL	25,00 TL
	444	08.12.2007	70.000,00 TL	2,50 TL

Figure 3.4.2 The Money is Given To Villagers Form

When the writting process finish , user can save, edit, update, delete,close or taking print out from the records (see at Figure 3.4.2).



26.05.2008

The Money is Given To Villagers

Customer No	Date	Amount	Litre Price
111	30.12.1899	4.000 YTL	2
222	30.12.1899	6.000 YTL	3
333	30.12.1899	3.000 YTL	1,5
444	08.12.2007	70.000 YTL	1

Figure 3.4.2 The Money is Given To Villagers Report

3.5 The Debts to have to pay

When the user cliks to menu item is called The Debts to have to pay on the main window which shows at Figure 3.5.1 then The Quantity Of Money Have To Pay window will be on the screen (see at Figure 3.5.2) .

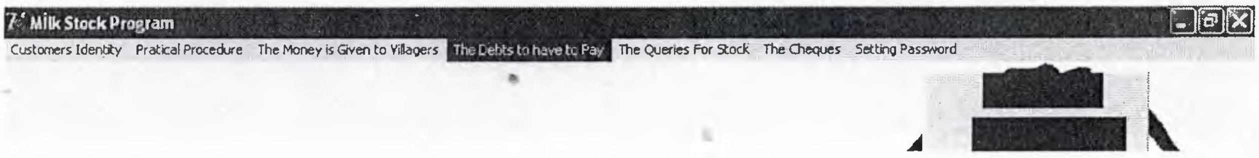
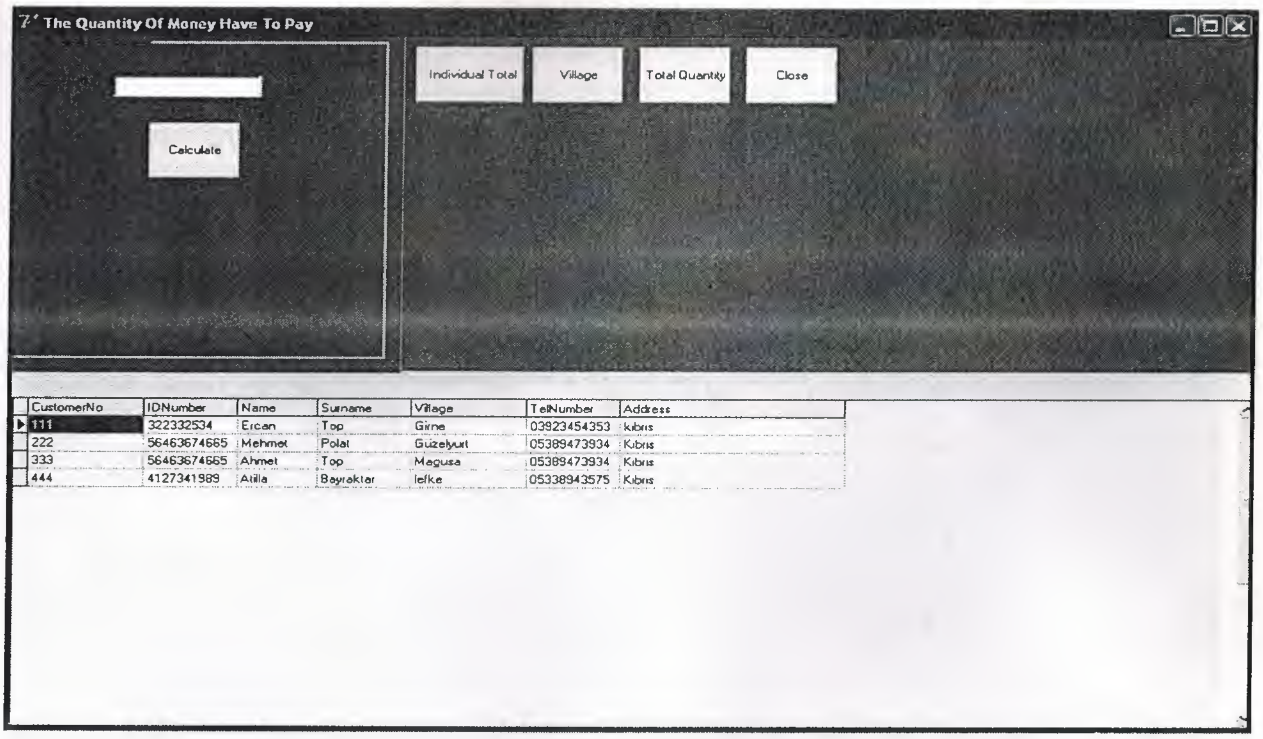


Figure 3.5.1 Milk Stock Menu

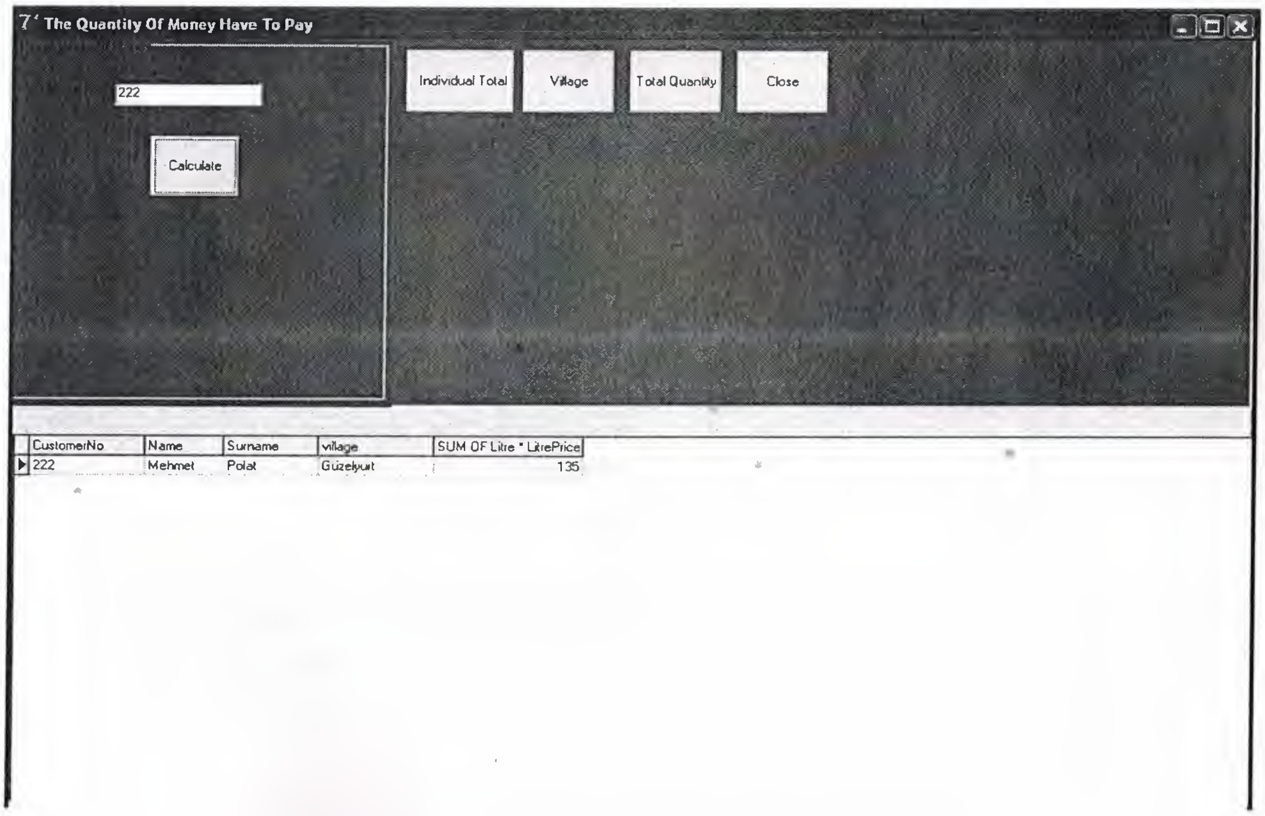
After clicking the menu item The Debts to have to pay then user will see this screen at below(see Figure 3.5.2) and at this form user can learn to the Money amount which have to give the villagers.



	CustomerNo	IDNumber	Name	Surname	Village	TelNumber	Address
▶	111	322332534	Ercan	Top	Girne	03923454353	kıbrıs
	222	56463674665	Mehmet	Polat	Guzelyurt	05389473934	Kıbrıs
	333	56463674665	Ahmet	Top	Magusa	05389473934	Kıbrıs
	444	4127341989	Atilla	Bayraktar	lefke	05338943575	Kıbrıs

Figure 3.5.2 The Quantity Of Money Have To Pay Form

Also user can calculate the villagers account by entering to CustomerNo and press to calculate button. For example user enters to 222 of villager's CustomerNo then you can see Figure 3.5.3 at the below.

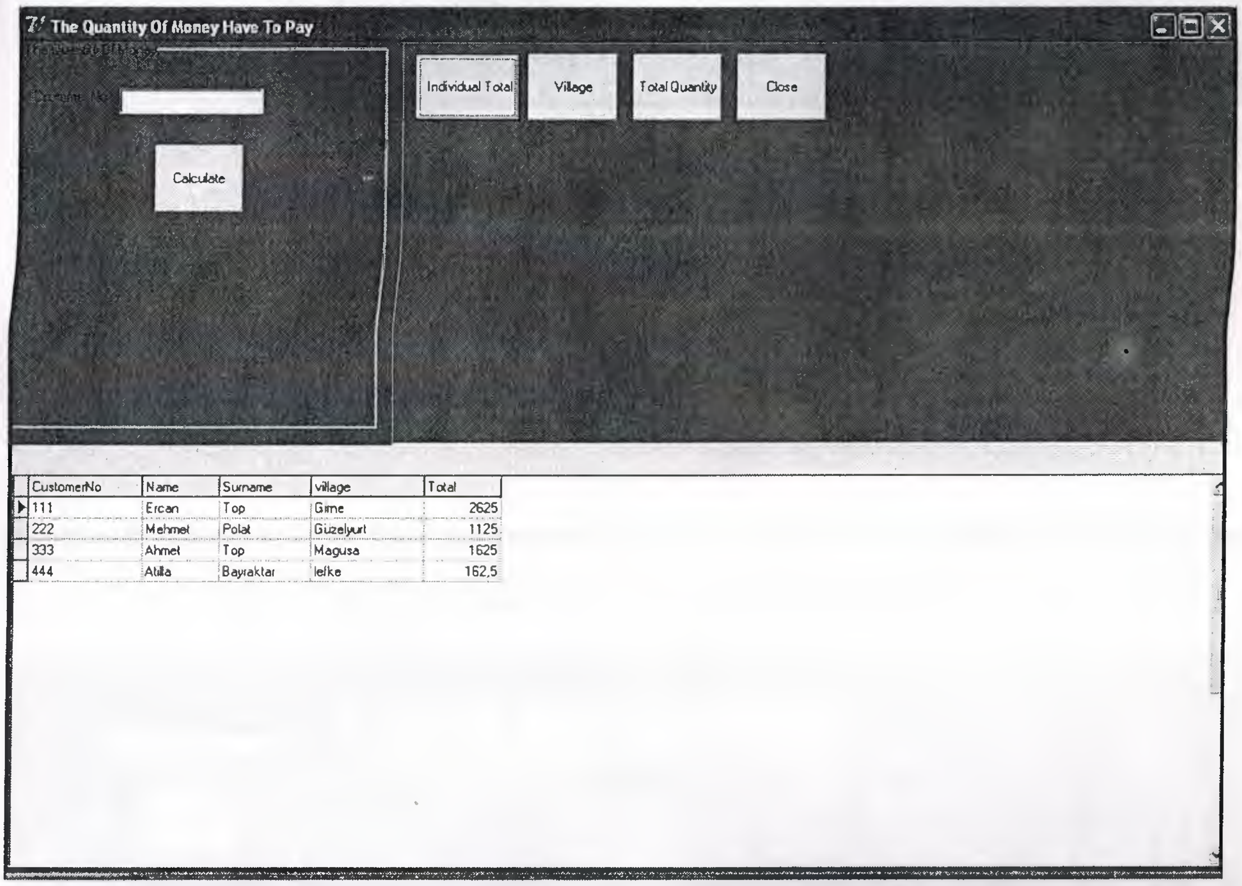


	CustomerNo	Name	Surname	village	SUM OF Litre * LitrePrice
▶	222	Mehmet	Polat	Guzelyurt	135

Figure 3.5.3 The Quantity Of Money Have To Pay Form

Now you can see when user entered 222 of villager's CustomerNo. You see account of M.r Mehmet Polat on the figure 3.5.3.

At this form there are some buttons on the top of the form. User can use these button to make their work easy. For example if user press to Total Individuals button, you will see figure 3.5.4 . You can see all villagers' individual total account on the form.

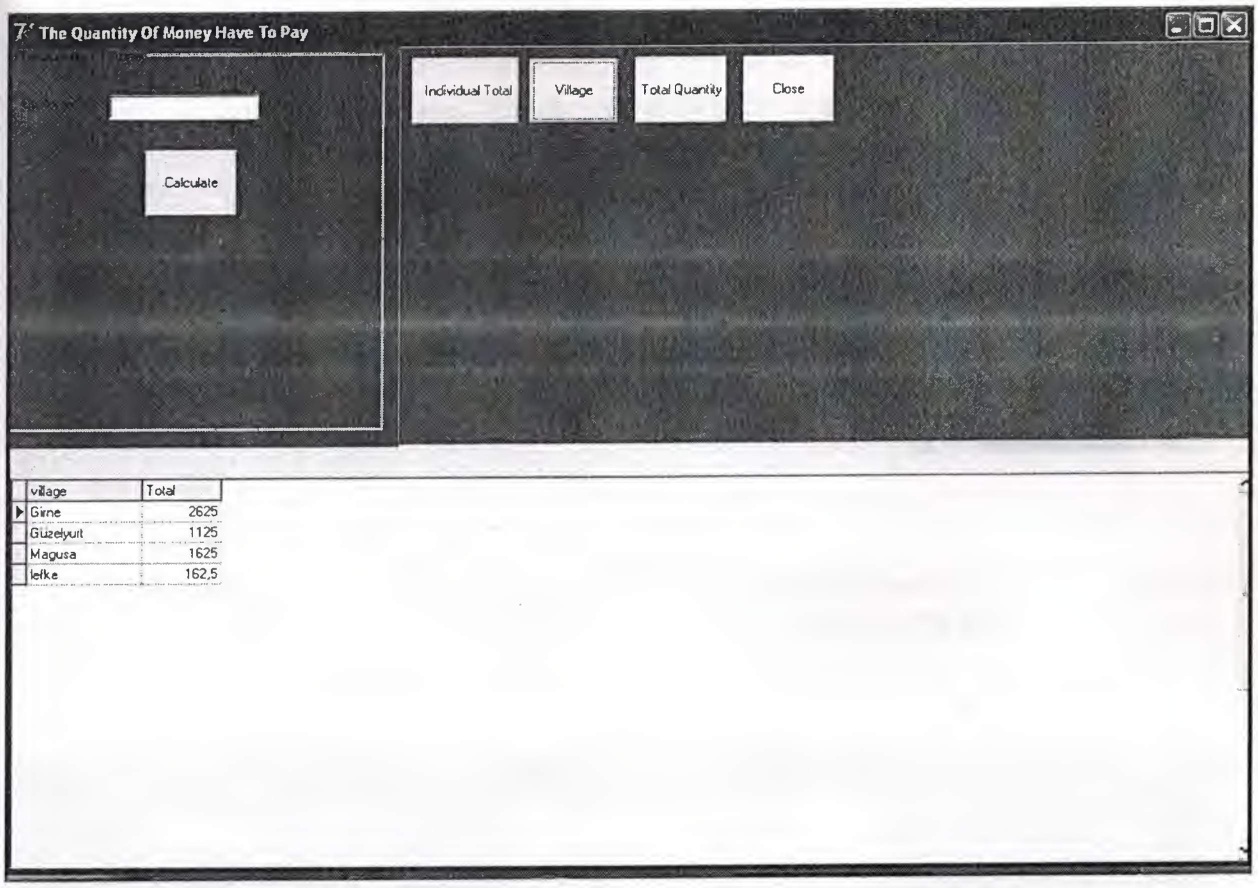


CustomerNo	Name	Surname	village	Total
111	Ercan	Top	Gime	2625
222	Mehmet	Polat	Guzelyurt	1125
333	Ahmet	Top	Magusa	1625
444	Atilla	Bayraktar	lefke	162,5

Figure 3.5.4 The Quantity Of Money Have To Pay Form

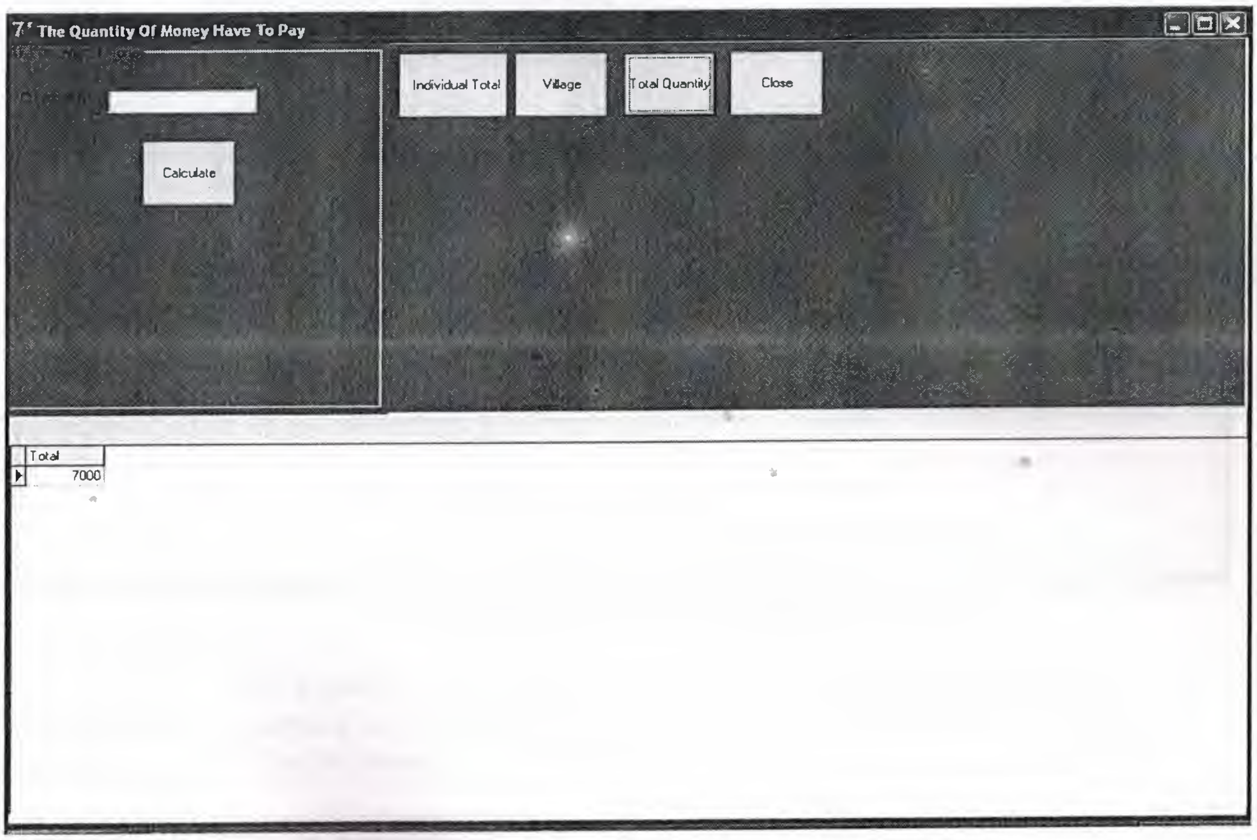
If user want to see totol quantity of money are given to by all villages. User will press to Village button and program will give automatically all the Money are distributed to villages which is shown figure 3.5.5 at the below.

And if user want to see total quatity of money that is given to villagers. User will press to Total Quantity button and program will give an output total of the money which is given all the villagers(see figure 3.5.6) .



village	Total
Girne	2625
Guzelyurt	1125
Magusa	1625
lefke	162,5

Figure 3.5.5 The Quantity Of Money Have To Pay Form



Total
7000

Figure 3.5.6 The Quantity Of Money Have To Pay Form

3.6 The Queries For Stock

When the user cliks to menu item is called The The Queries For Stock on the main window which shows at figure 3.6.1 then Stock Procedure window will be open on the screen (see at figure 3.6.2) .

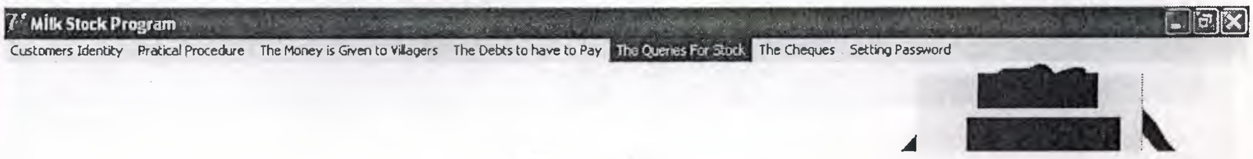
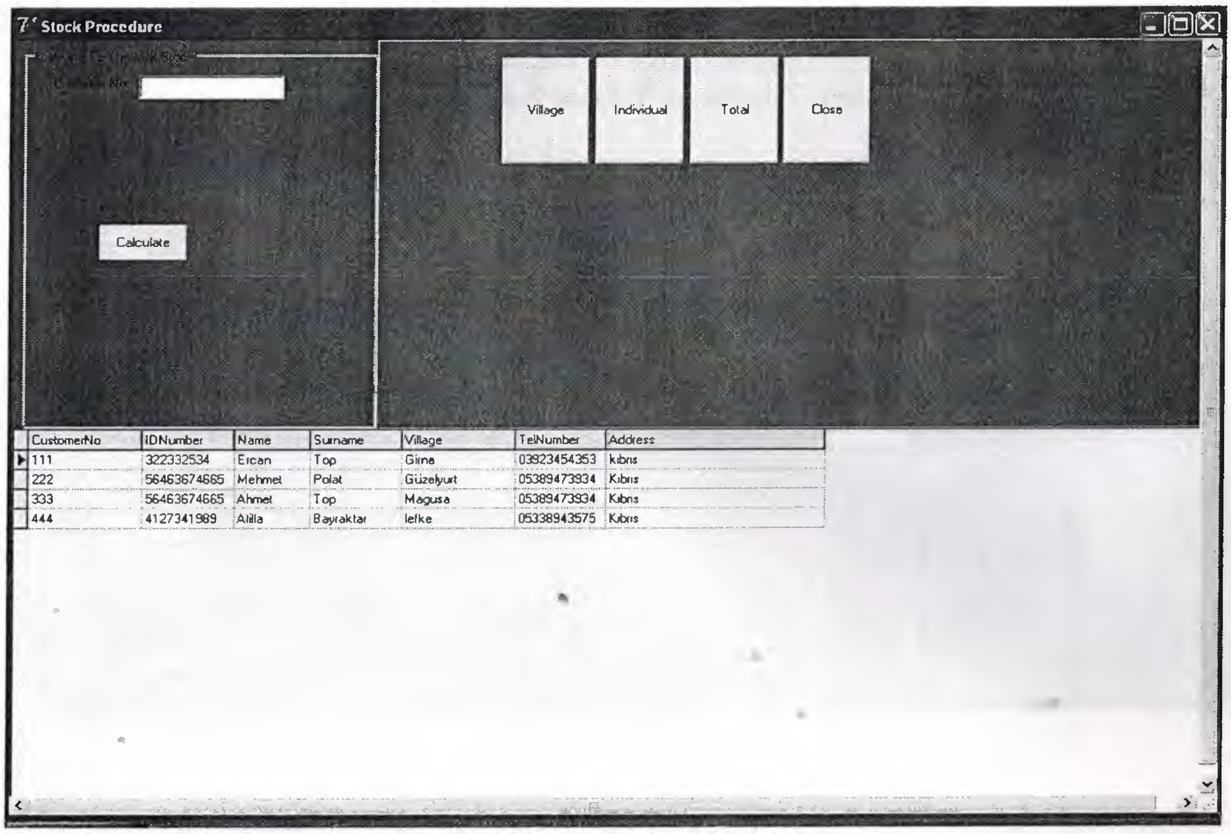


Figure 3.6.1

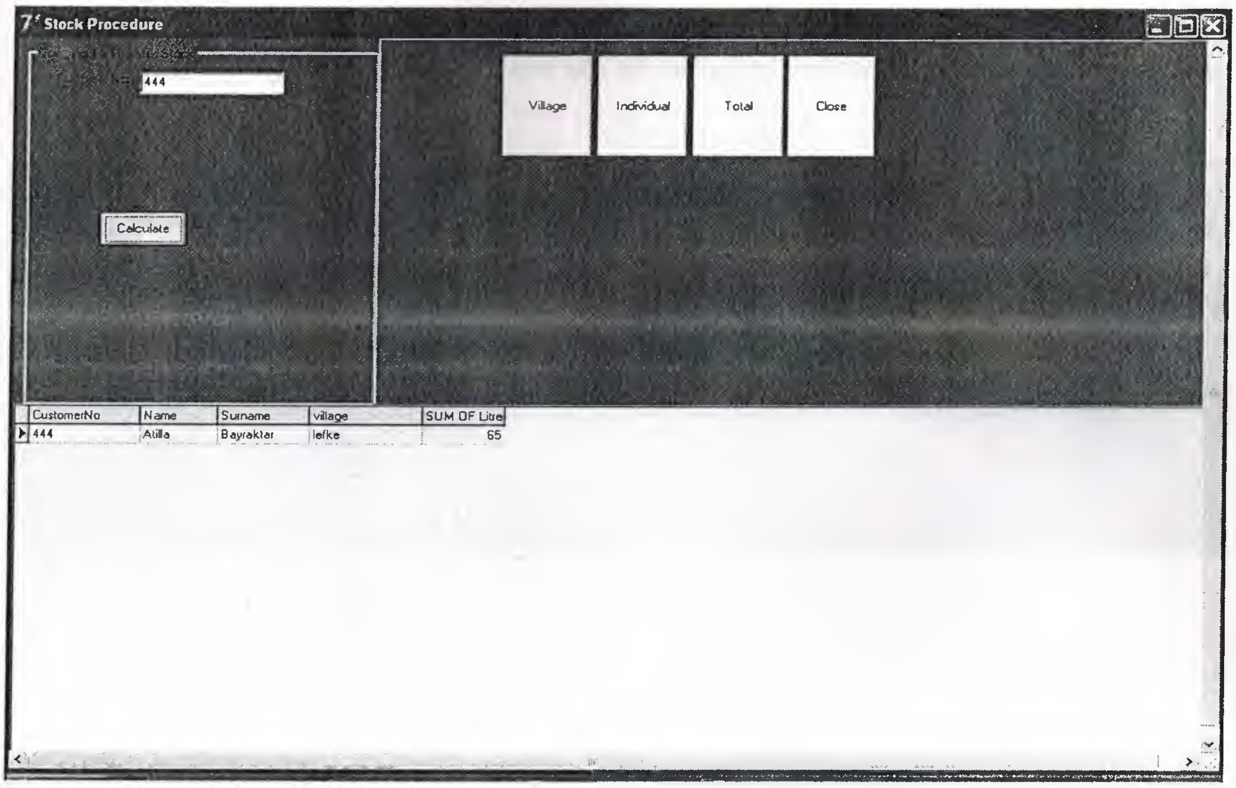
After clicking the menu item The Debts to have to pay then user will see this screen at below(see Figure 3.5.2) and at this form user can learn to total of the milk litre which are collected from the villagers.



	CustomerNo	IDNumber	Name	Surname	Village	TelNumber	Address
▶	111	322332534	Ercan	Top	Girne	03923454353	kıbrıs
	222	56463674665	Mehmet	Polat	Güzelyurt	05389473934	Kıbrıs
	333	56463674665	Ahmet	Top	Magusa	05389473934	Kıbrıs
	444	4127341989	Atilla	Bayraktar	lefke	05338943575	Kıbrıs

Figure 3.6.2 Stock Procedure Form

Also user can calculate the villagers' total milk litre by entering to CustomerNo and press to calculate button. For example user enters to 444 of villager's CustomerNo then you can see Figure 3.6.3 at the below.

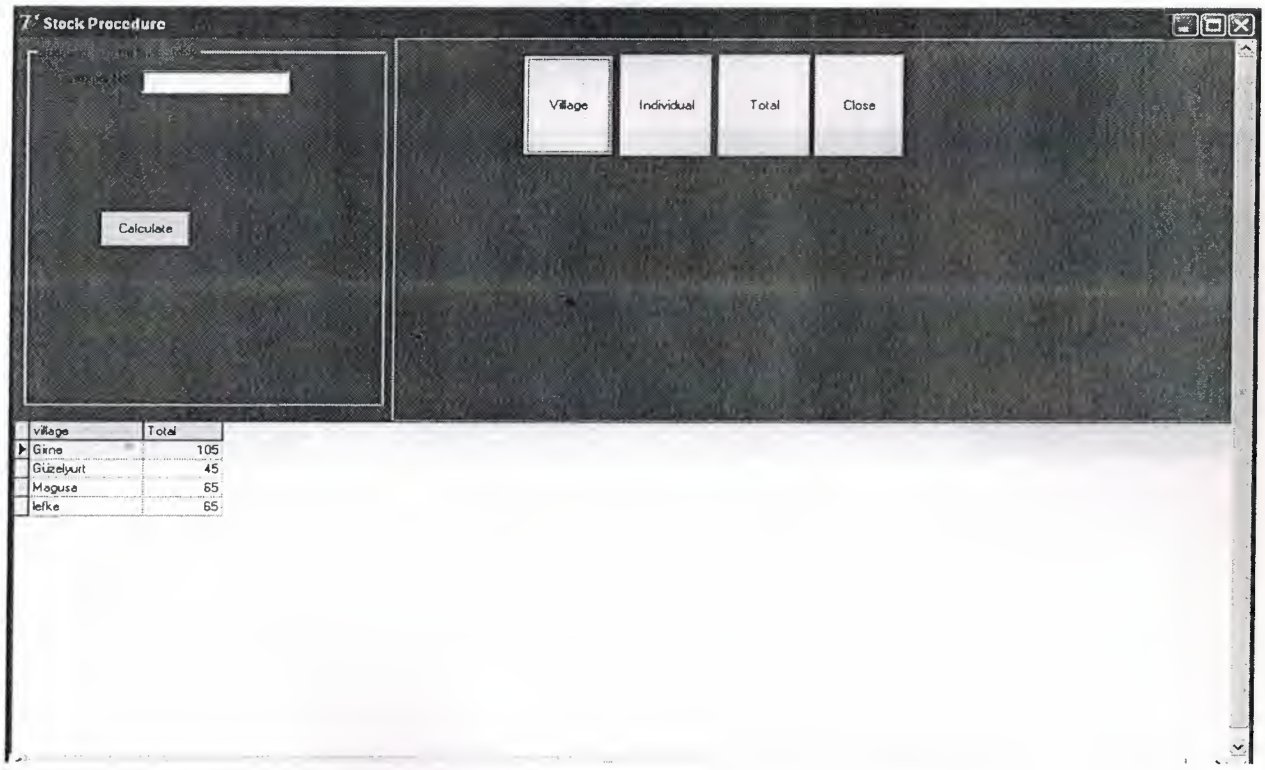


444

	CustomerNo	Name	Surname	village	SUM OF Litre
▶	444	Atilla	Bayraktar	lefke	65

Figure 3.6.3 Stock Procedure Form

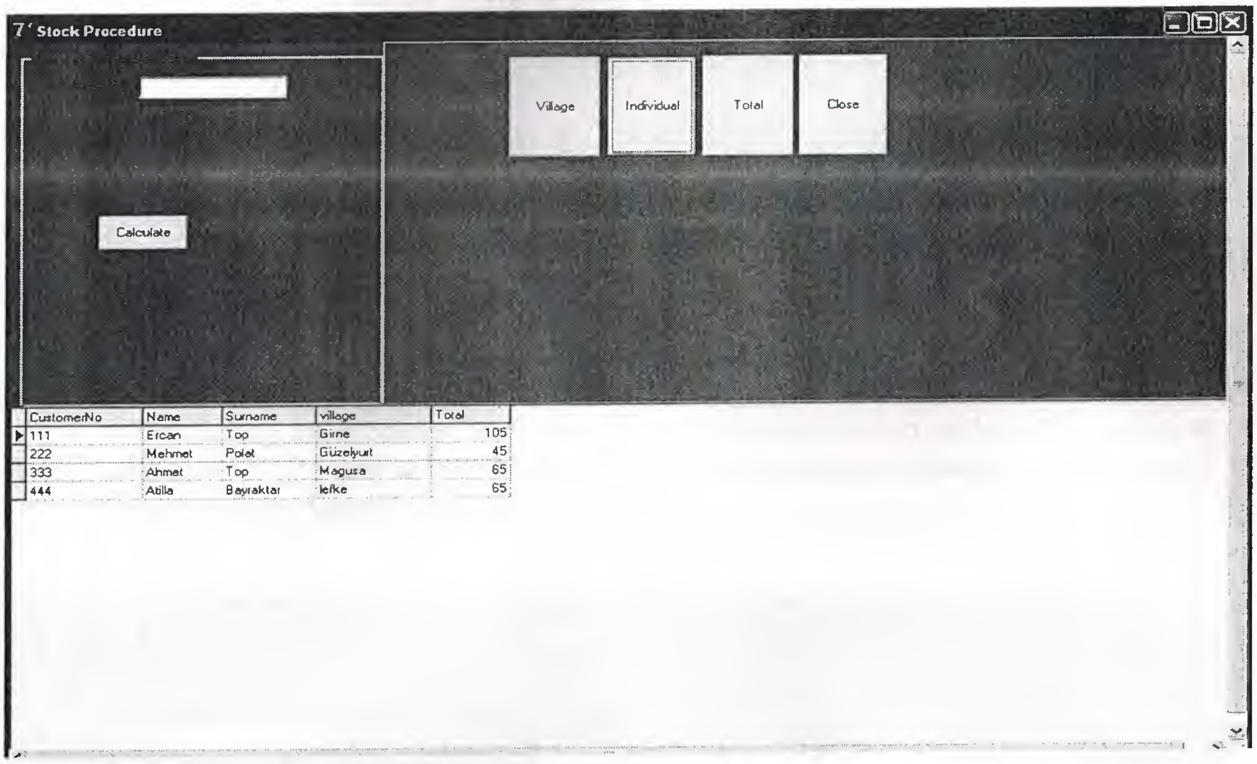
If user want to see totol milk litre are collected to by all villages. User will press to Village button and program will give automatically all the total milk are distributed by villages which is shown figure 3.6.4 at the below.



	village	Total
▶	Girne	105
	Güzelyurt	45
	Magusa	65
	lefke	65

Figure 3.6.4 Stock Procedure Form

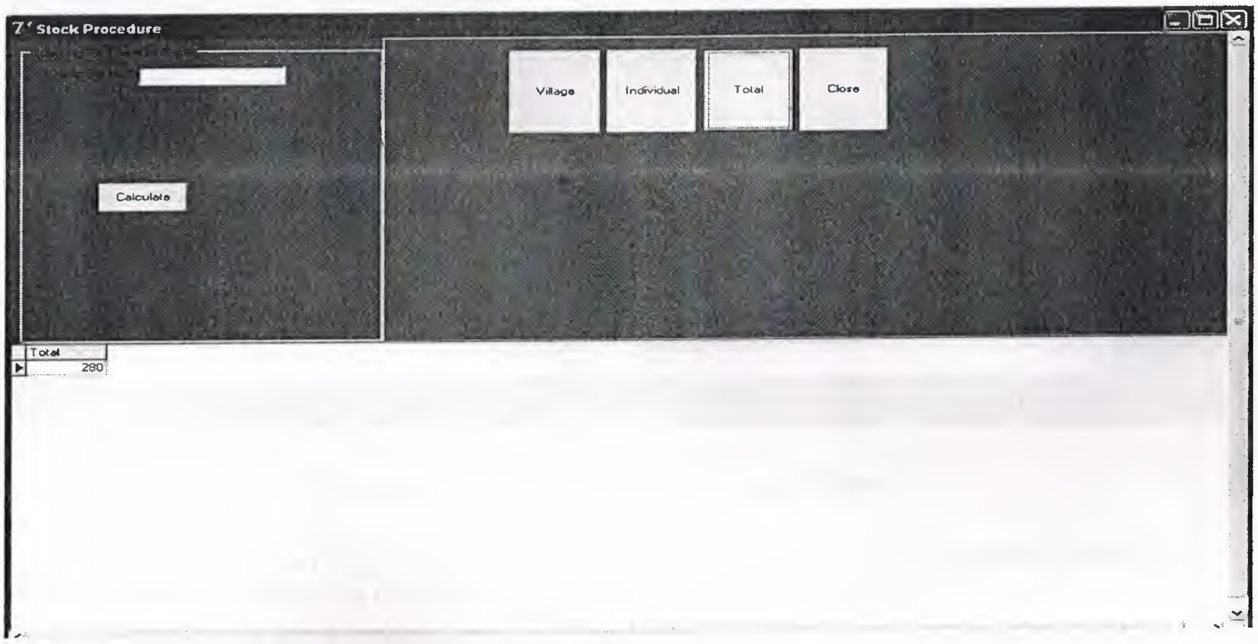
if user press to Individuals button then program automatically calculate total milk litre, you will see figure 3.5.4 . You can see all villagers' individual total milk litre on the table.



CustomerNo	Name	Surname	village	Total
111	Ercan	Top	Girne	105
222	Mehmet	Polat	Güzelyurt	45
333	Ahmet	Top	Magusa	65
444	Atilla	Bayraktar	lefke	65

Figure 3.6.4 Stock Procedure Form

And if user want to see total litre of the milk which is collected from the villagers. User will press to Total button and program will give an output total litre of the milk which is collected all the villagers(see figure 3.6.5) .



Total
280

Figure 3.6.5 Stock Procedure Form

3.7 The Cheques

When the user cliks to menu item which is called here The Cheques on the main window which shows at Figure 3.7.1. If user click to The Cheques and then Entering the Cheques's item and Learning to the Cheques' item submenus will be on the screen (see at Figure 3.7.1).

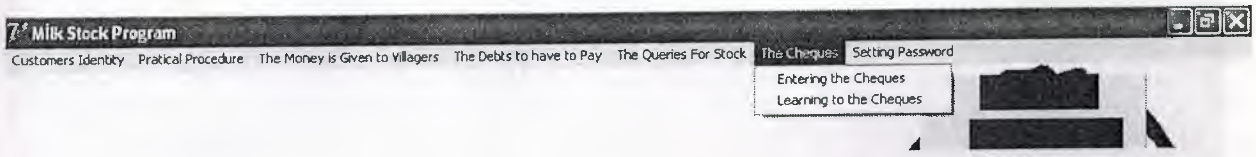
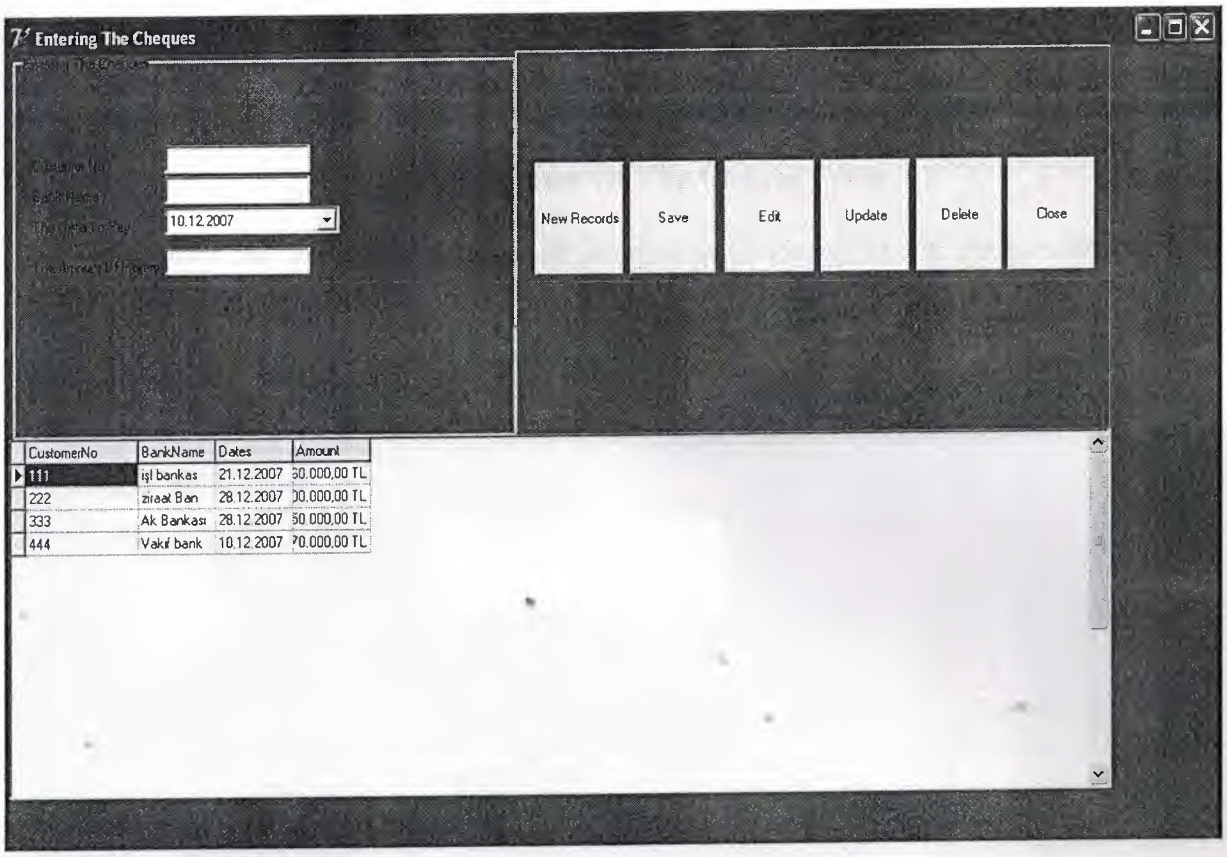


Figure 3.7.1 Milk Stock Menu

After clicking the submenu item Entering the Cheques then user will see this screen at below(see Figure 3.7.2) and at this form user can enter CustomerNo ,Bank Name, Date and to the Money amount which have to give the villagers.

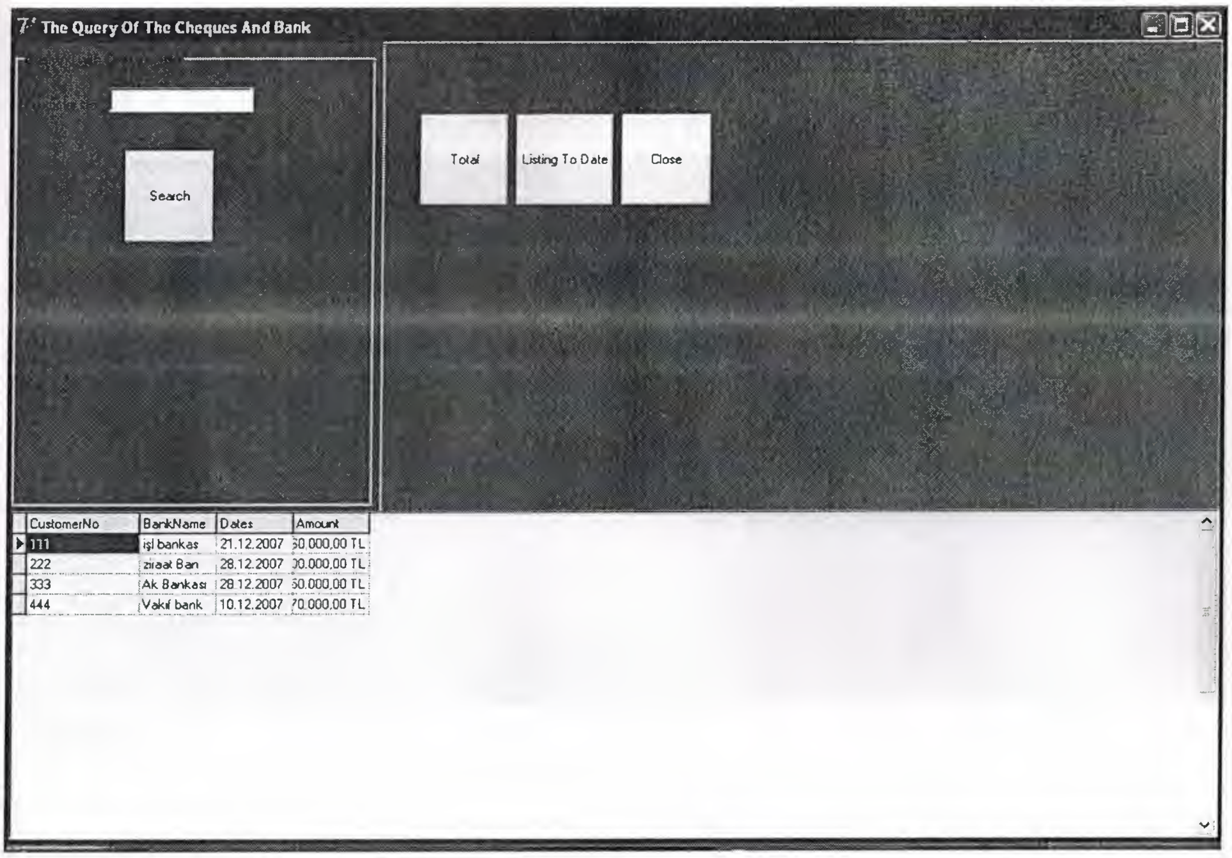


CustomerNo	BankName	Dates	Amount
111	işl bankas	21.12.2007	30.000,00 TL
222	ziraat Ban	28.12.2007	00.000,00 TL
333	Ak Bankası	28.12.2007	50.000,00 TL
444	Vakıf bank	10.12.2007	70.000,00 TL

Figure 3.7.2 Entering The Cheques Form

And also for saving, editing, updating, deleting, closing of the Cheques information user uses some buttons on the top of the right side(see at Figure 3.7.2).

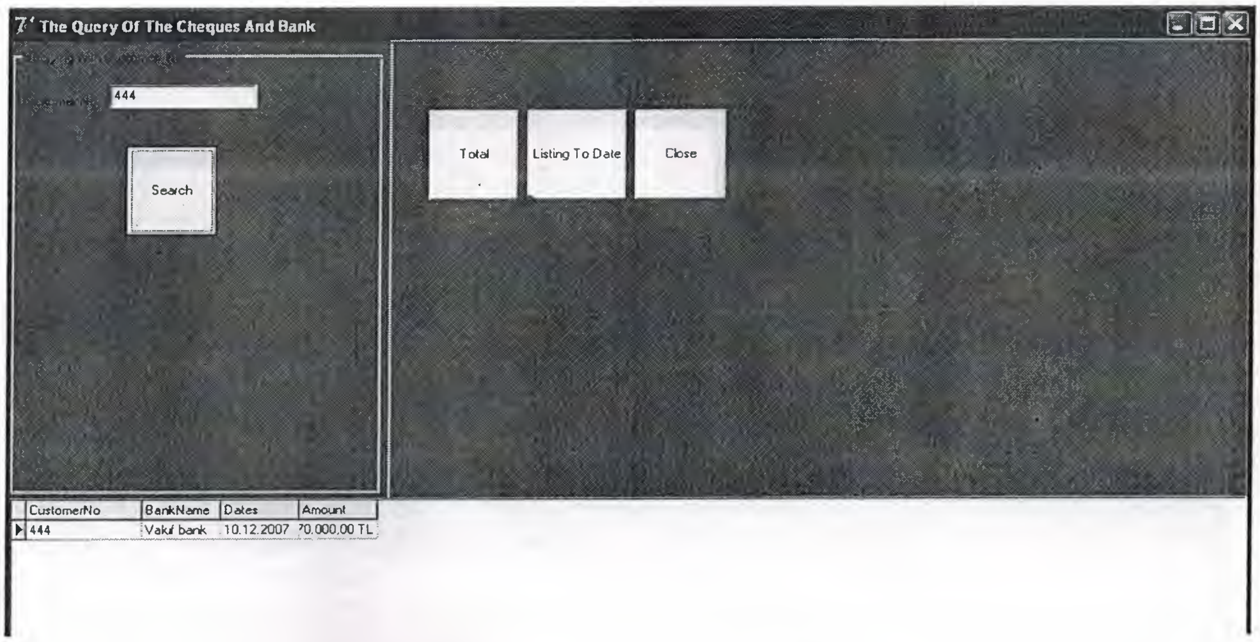
If user clicks the submenu item Learning to the Cheques' item then user will see this screen at below(see Figure 3.7.3) and in this area user can see CustomerNo, Bank Name, Date and to the Money amount which have to pay banks.



	CustomerNo	BankName	Dates	Amount
▶	111	işl bankas	21.12.2007	50.000,00 TL
	222	ziraat Ban	28.12.2007	00.000,00 TL
	333	Ak Bankası	28.12.2007	50.000,00 TL
	444	Vakıf bank	10.12.2007	70.000,00 TL

Figure 3.7.4 Query Of The Cheque Form

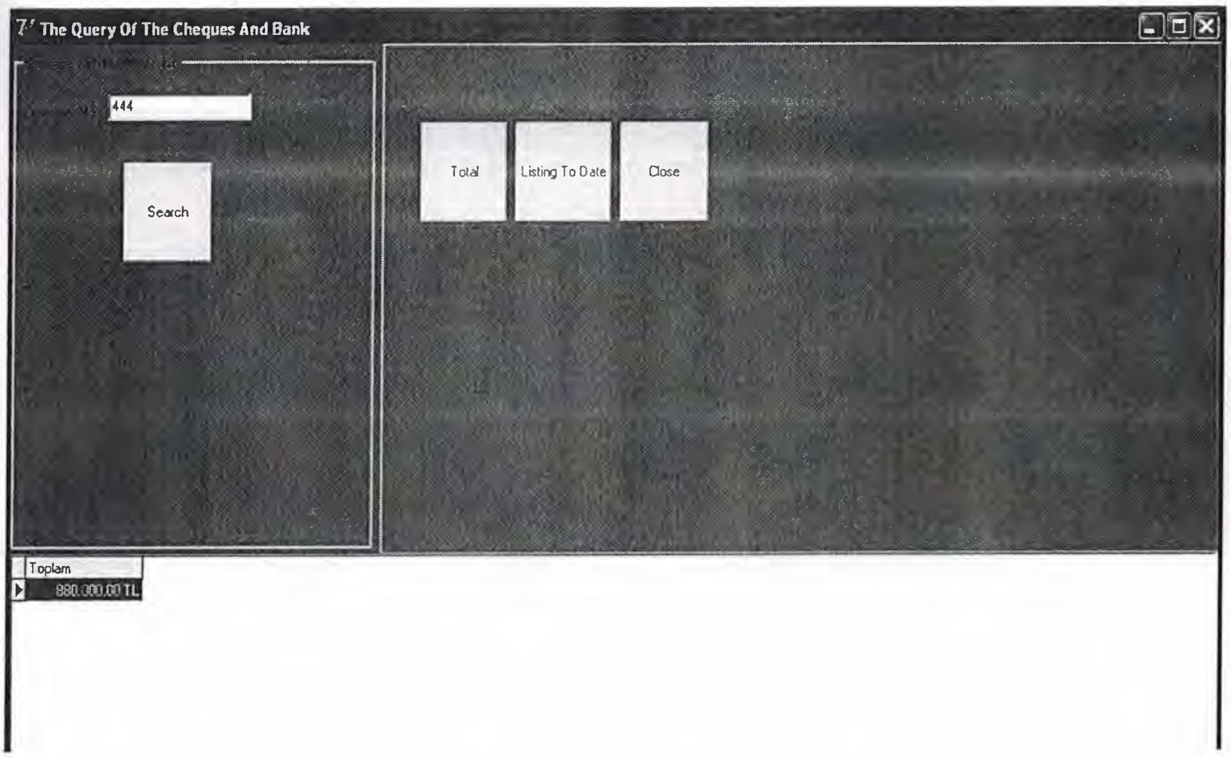
Also user can see all the cheques by entering to CustomerNo and press to Search button. For example user enters to 444 of villager's CustomerNo then you can see Figure 3.6.3 at the below.



	CustomerNo	BankName	Dates	Amount
▶	444	Vakıf bank	10.12.2007	70.000,00 TL

Figure 3.7.5 Query Of The Cheque Form

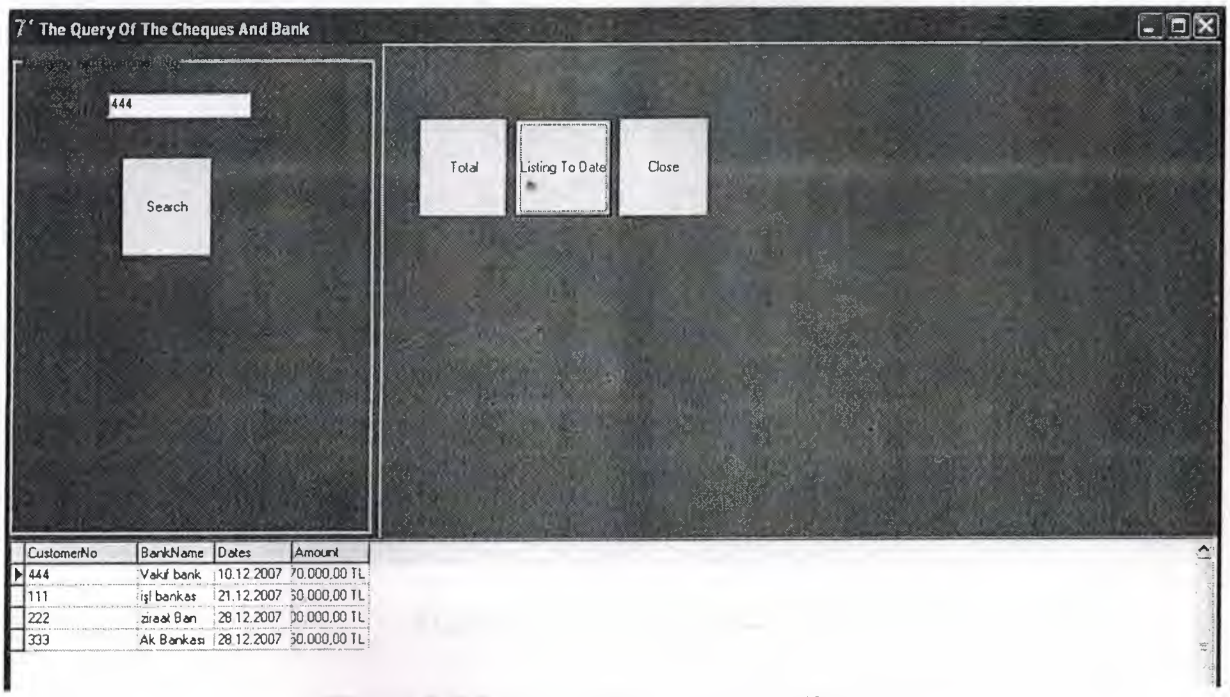
If user want to see total quatity of cheques that are given to villagers. User will press to Total button and program will give an output total of the money which is given all the villagers(see figure 3.7.6) .



Toplam
880.000,00 TL

Figure 3.7.6 Query Of The Cheque Form

And if user want to see list of dates of the cheques that is given to villagers. User will press to List To Dates button and program will give an output list of the dates of the cheques which is given all the villagers(see figure 3.7.7).



CustomerNo	BankName	Dates	Amount
444	Vakıf bank	10.12.2007	70.000,00 TL
111	işl bankas	21.12.2007	50.000,00 TL
222	ziraat Ban	28.12.2007	00.000,00 TL
333	Ak Bankası	28.12.2007	50.000,00 TL

Figure 3.7.7 Query Of The Cheque Form

3.8 Setting Password

When the user cliks to menu item is called here Setting Password on the main window which shows at Figure 3.8.1 then Setting Password window will be on the screen (see at Figure 3.8.2) .

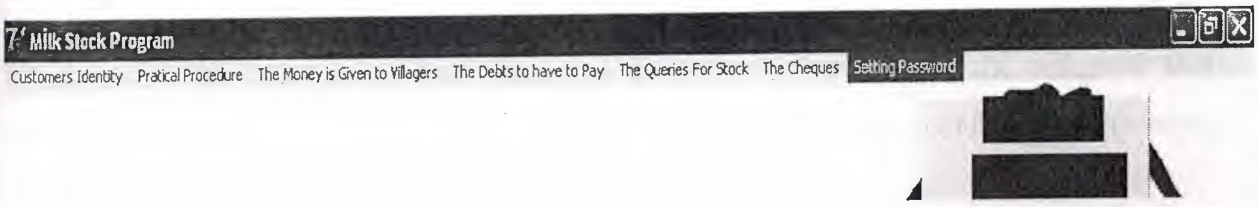


Figure 3.8.1 Milk Stock Menu

After clicking the menu item Setting Password then user will see this screen at below(see Figure 3.8.2) and at this form you can add new user and password or change to passwor or name, delete to user.

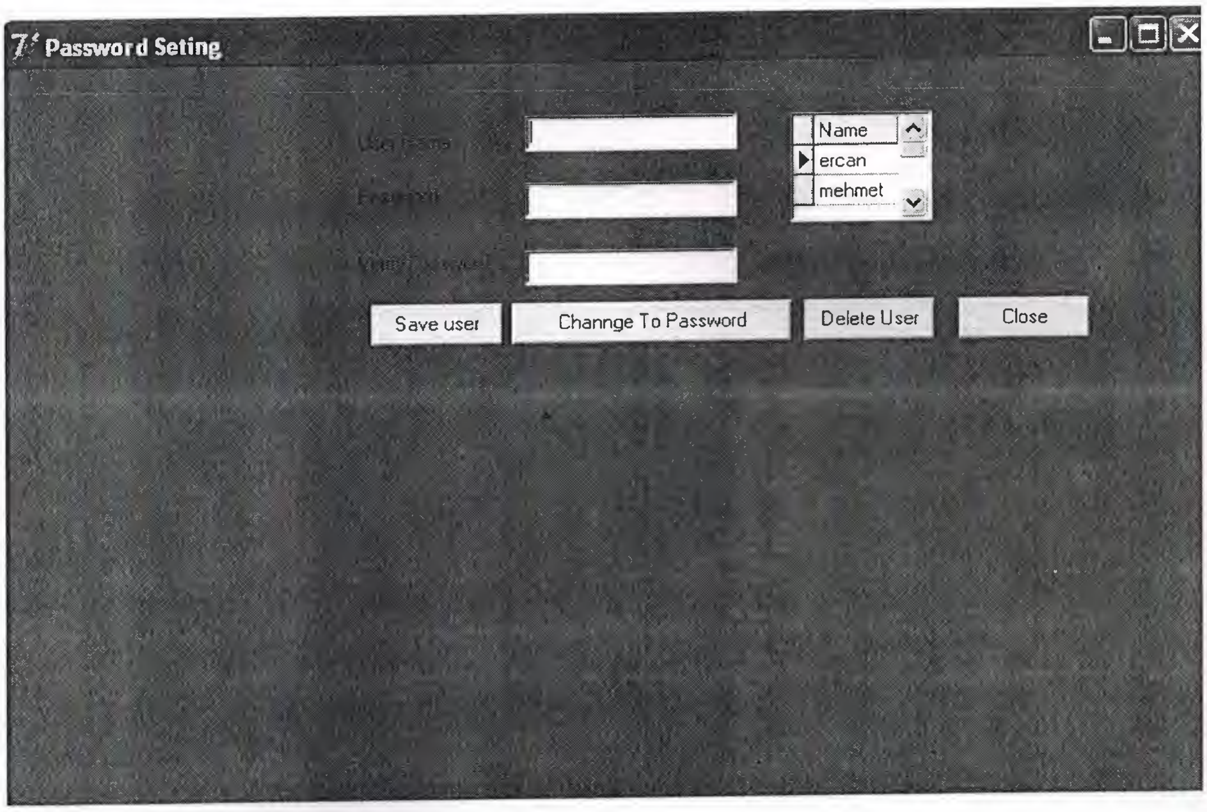


Figure 3.8.2 Password Setting Form

CONCLUSION

The aim of this project was supplying a comfortable situation and easy way of collecting milk in a stock. In the early years, the stock information were writing by hand on some papers and combined later. But it was not a good way and also very hard for employees. After that period it is started to keep the information in the computer like programs notepad etc. Nowadays it is started to store these information in databases by using Milk stock programs as this Project.

Not only for keeping the customer information or stock conditions are included in this programs. At this point Milk Stock program will provide easiness and quickness of controlling stock. In this project the main point is making the users job easy. In other words it is used for all Cheese and milk companies.

At last Automation technology develops very fast and following it is too hard. In the next years it can be seen that every job is done only by a chip card (today it has examples). Let us see what the future shows!

REFERENCES

[1]. Delphi Programming Explorer Book

[2]. Borland Delphi 7 – İhsan Karagülle (Türkmen Kitabevi)

[3]. www.barcodewiz.com

[4]. www.borland.com

[5]. www.delphiturk.com

[6]. www.programmersheaven.com

[7]. www.programlama.com

[8] . http://www.wikipedia.com

[9]. www.prestigeturk.com

APPENDIX : SOURCE CODE OF THE PROGRAM

FORM 1. Main Window And Functions

```
unit Unit1;

interface

uses

  Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms,
  Dialogs, Menus, StdCtrls, ExtCtrls;

type

  TForm1 = class(TForm)
    MainMenu1: TMainMenu;
    M1: TMenuItem;
    Pratikilemleri1: TMenuItem;
    Stalm1: TMenuItem;
    Alacaklar1: TMenuItem;
    Borlar1: TMenuItem;
    Stokilemleri1: TMenuItem;
    Kayt1: TMenuItem;
    bekleyenmterieksenetleri1: TMenuItem;
    eksenetgir1: TMenuItem;
    eksenetde1: TMenuItem;
    Timer1: TTimer;
    Panel1: TPanel;
    Label1: TLabel;
    Label2: TLabel;
    Image1: TImage;
    SettingPassword1: TMenuItem;

    procedure Kayt1Click(Sender: TObject);
    procedure Stalm1Click(Sender: TObject);
    procedure Alacaklar1Click(Sender: TObject);
    procedure Borlar1Click(Sender: TObject);
    procedure eksenetgir1Click(Sender: TObject);
    procedure Stokilemleri1Click(Sender: TObject);
    procedure FormCreate(Sender: TObject);
    procedure Timer1Timer(Sender: TObject);
    procedure eksenetde1Click(Sender: TObject);
    procedure SettingPassword1Click(Sender: TObject);
    procedure FormClose(Sender: TObject; var Action: TCloseAction);

private
    { Private declarations }
  public
  { Public declarations }
 end;

var
 Form1: TForm1;

implementation

uses unit2,unit3,unit4,unit5,unit6,unit7,unit10,unit16;


{$R *.dfm}


procedure TForm1.Kayt1Click(Sender: TObject);

begin

        Form2.Show;
        Form1.Hide;
end;

procedure TForm1.Stalm1Click(Sender: TObject);
begin
        Form1.Hide;
        form3.Show;

end;

procedure TForm1.Alacaklar1Click(Sender: TObject);
begin

        Form4.Show;
        Form1.Hide;
end;

procedure TForm1.Borlar1Click(Sender: TObject);
begin
        Form5.Show;
        Form1.Hide;

end;

procedure TForm1.eksenetgir1Click(Sender: TObject);
begin
        Form6.Show;
        Form1.Hide;

end;
```

```
procedure TForm1.Stokilemleri1Click(Sender: TObject);
begin
        Form7.Show;
        form1.Hide;

end;

procedure TForm1.FormCreate(Sender: TObject);
begin

        Label2.Caption:=DateToStr(date);

end;

procedure TForm1.Timer1Timer(Sender: TObject);
begin

        Label1.Caption:=TimeToStr(time);

end;

procedure TForm1.eksenetde1Click(Sender: TObject);
begin

        Form1.Hide;
        Form10.Show;

end;

procedure TForm1.SettingPassword1Click(Sender: TObject);
begin

        Form1.Hide;
        Form16.Show;

end;

procedure TForm1.FormClose(Sender: TObject; var Action: TCloseAction);
begin

        halt;

end;

end.
```

FORM 2. Client Record

```
unit Unit2;

interface

uses

 Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms,
 Dialogs, StdCtrls, Grids, DBGrids, DB, DBTables, ExtCtrls;

type
 TForm2 = class(TForm)
   DBGrid1: TDBGrid;
   DataSource1: TDataSource;
   Query1: TQuery;
   Table1: TTable;
   GroupBox1: TGroupBox;
   Label1: TLabel;
   Label2: TLabel;
   Label3: TLabel;
   Label4: TLabel;
   Label5: TLabel;
   Label6: TLabel;
   Edit1: TEdit;
   Edit2: TEdit;
   Edit3: TEdit;
   Edit4: TEdit;
   Edit5: TEdit;
   Edit6: TEdit;

   Panel1: TPanel;
   Button1: TButton;
   Button2: TButton;
   Button3: TButton;
   Button4: TButton;
   Button5: TButton;
   Button6: TButton;
   Button7: TButton;
   Button8: TButton;
   Button9: TButton;
   Button10: TButton;
   Edit7: TEdit;
   Table1CustomerNo: TStringField;
   Table1IDNumber: TStringField;
   Table1Name: TStringField;
   Table1Surname: TStringField;
   Table1Village: TStringField;
```

```
    Table1TelNumber: TStringField;
    Table1Address: TStringField;
    Label7: TLabel;

    procedure Button4Click(Sender: TObject);
    procedure FormShow(Sender: TObject);
    procedure FormClose(Sender: TObject; var Action: TCloseAction);
    procedure Button1Click(Sender: TObject);
    procedure Button5Click(Sender: TObject);
    procedure Button6Click(Sender: TObject);
    procedure Button3Click(Sender: TObject);
    procedure Button7Click(Sender: TObject);
    procedure Button2Click(Sender: TObject);
    procedure Button8Click(Sender: TObject);
    procedure Button9Click(Sender: TObject);
    procedure Button10Click(Sender: TObject);
    procedure FormCreate(Sender: TObject);

  private
    x: Integer;
    { Private declarations }
  public
    { Public declarations }
  end;

var

Form2: TForm2;

implementation

uses unit1,unit9,unit11;

{$R *.dfm}

procedure TForm2.Button4Click(Sender: TObject);
begin

        Form1.Show;
        Form2.Hide;

end;

procedure TForm2.FormShow(Sender: TObject);
begin

        Query1.SQL.Text := 'Select * from  customer';
        Query1.Open;
        x := 0;
```

```
end;

procedure TForm2.FormClose(Sender: TObject;
  var Action: TCloseAction);
begin

        Query1.Close;
        Form1.Show;

end;

procedure TForm2.Button1Click(Sender: TObject);
begin

        Edit7.Text:='';
        Edit1.Text:='';
        Edit2.Text:='';
        Edit3.Text:='';
        Edit4.Text:='';
        Edit5.Text:='';
        Edit6.Text:='';
        Edit1.SetFocus;


end;

procedure TForm2.Button5Click(Sender: TObject);
begin

        Explorer.Show;
        Explorer.WebBrowser1.Navigate('http://tckimlik.nvi.gov.tr/Web/QueryIdentityNumbe
r.aspx');
        Form2.Hide;

end;

procedure TForm2.Button6Click(Sender: TObject);
begin

        Table1.SetKey;
        Table1.FieldByName('CustomerNo').AsString:=Edit7.Text;
        Table1.GotoKey
        if Table1.GotoKey=false Then
                begin

                        Table1.Insert;
                         Table1CustomerNo.AsString:=Edit7.Text;
                         Table1IDNumber.AsString:=Edit1.Text;
                         Table1Name.AsString:=Edit2.Text;
                         Table1Surname.AsString:=edit3.Text;
```

```
                    Table1Village.AsString:=Edit4.Text;
                    Table1TelNumber.AsString:=Edit5.Text;
                    Table1Address.AsString:=Edit6.Text;
                    Table1.Post;
               end

      else ShowMessage('Error');


end;

procedure TForm2.Button3Click(Sender: TObject);

var
   mesaj:integer;

begin

      mesaj:=Application.MessageBox('Are you sure to delete',
' Delete',MB_ICONSTOP+MB_YESNO);
        if mesaj=mryes then
                begin

                    Table1.Delete;
                    Table1.Edit;
                    Edit7.Text:=Table1CustomerNo.Text;
                    Edit1.Text:=Table1IDNumber.Text;
                    Edit2.Text:=Table1Name.Text;
                    Edit3.Text:=Table1Surname.Text;
                    Edit4.Text:=Table1Village.Text;
                    Edit5.Text:=Table1TelNumber.Text;
                    Edit6.Text:=Table1Address.Text;

                    ShowMessage('Deleted);

        end

 else

                    ShowMessage('Canceled');

end;

procedure TForm2.Button7Click(Sender: TObject);
begin
        Table1.Cancel;
        Table1.Edit;
          Edit7.Text:=Table1CustomerNo.Text;
        Edit1.Text:=Table1IDNumber.Text;
        Edit2.Text:=Table1Name.Text;
        Edit3.Text:=Table1Surname.Text;
        Edit4.Text:=Table1Village.Text;
        Edit5.Text:=Table1TelNumber.Text;
         Edit6.Text:=Table1Address.Text;

end;

procedure TForm2.Button2Click(Sender: TObject);
begin

        Table1.Edit;
        Table1.Edit;
        Edit7.Text:=Table1CustomerNo.Text;
        Edit1.Text:=Table1IDNumber.Text;
        Edit2.Text:=Table1Name.Text;
         Edit3.Text:=Table1Surname.Text;
         Edit4.Text:=Table1Village.Text;
         Edit5.Text:=Table1TelNumber.Text;
        Edit6.Text:=Table1Address.Text;

end;

procedure TForm2.Button8Click(Sender: TObject);
begin

        Table1.ClearFields;
        Table1CustomerNo.AsString:=Edit7.Text;
        Table1IDNumber.AsString:=Edit1.Text;
        Table1Name.AsString:=Edit2.Text;
        Table1Surname.AsString:=edit3.Text;
        Table1Village.AsString:=Edit4.Text;
        Table1TelNumber.AsString:=Edit5.Text;
        Table1Address.AsString:=Edit6.Text;
        Table1.Post;
end;

procedure TForm2.Button9Click(Sender: TObject);

var
ara:boolean;
begin

        if (Edit7.Text='') and (Edit2.Text='') Then
                ShowMessage('Please enter the  name and surname')
        else
        begin
                ara:=Table1.Locate('Name;Surname',VarArrayOf([Edit2.Text,Edit3.Text]),[]);

end;
```

```
end;

procedure TForm2.Button10Click(Sender: TObject);
begin

      Form11.QuickRep1.Preview;

end;

procedure TForm2.FormCreate(Sender: TObject);
begin

end;

end.
```

FORM 3. Pratical Procedure And Milk Purchasing

```
unit Unit3;

interface

uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms,
  Dialogs, StdCtrls, ComCtrls, Grids, DBGrids, DB, DBTables, ExtCtrls;

type
  TForm3 = class(TForm)
    DBGrid1: TDBGrid;
    Label1: TLabel;
    Label2: TLabel;
    Label3: TLabel;
    edit1: TEdit;
    combobox1: TComboBox;
    datetimepicker1: TDateTimePicker;
    Panel1: TPanel;
    Button1: TButton;
    Button2: TButton;
    Button3: TButton;
    Button4: TButton;
    Button5: TButton;
    Table1: TTable;
    DataSource1: TDataSource;
    Button6: TButton;
```

```
   Table1CustomerNo: TStringField;
   Table1Date: TDateField;
   Table1Litre: TFloatField;
   Table1TimeSelection: TStringField;
   Edit2: TEdit;

   procedure FormClose(Sender: TObject; var Action: TCloseAction);
   procedure Button1Click(Sender: TObject);
   procedure Button3Click(Sender: TObject);
   procedure Button4Click(Sender: TObject);
   procedure Button2Click(Sender: TObject);
   procedure Button5Click(Sender: TObject);
   procedure Button6Click(Sender: TObject);
   procedure FormCreate(Sender: TObject);

 private
   { Private declarations }
  public
   { Public declarations }
  end;

var
 Form3: TForm3;

implementation

uses unit1;

{$R *.dfm}

procedure TForm3.FormClose(Sender: TObject; var Action: TCloseAction);
begin

        Form1.Show;

end;

procedure TForm3.Button1Click(Sender: TObject);
begin

        Form1.Show;
        Form3.Hide;

end;

procedure TForm3.Button3Click(Sender: TObject);
begin

        Table1.Insert;
        Table1CustomerNo.AsString:=Edit1.Text;
        Table1Date.AsDateTime:=DateTimePicker1.DateTime;
        Table1Litre.AsString:=Edit2.Text;
        Table1TimeSelection.AsString:=ComboBox1.Text;
        Table1.Post;

end;

procedure TForm3.Button4Click(Sender: TObject);
begin

        Edit1.Text:='';
        Edit2.Text:='';
        ComboBox1.Text:='';
end;

procedure TForm3.Button2Click(Sender: TObject);
begin

        Edit1.Text:=Table1CustomerNo.Text;
        DateTimePicker1.DateTime:=Table1Date.AsDateTime;
        Edit2.Text:=Table1Litre.Text;
        ComboBox1.Text:=Table1TimeSelection.Text;
        Table1.Edit;

end;

procedure TForm3.Button5Click(Sender: TObject);
begin

        Table1.ClearFields;
        Table1CustomerNo.AsString:=Edit1.Text;
        Table1Date.AsDateTime:=DateTimePicker1.DateTime;
        Table1Litre.AsString:=Edit2.Text;
        Table1TimeSelection.AsString:=ComboBox1.Text;
        Table1.Post;

end;

procedure TForm3.Button6Click(Sender: TObject);

var

mesaj:integer;

begin

        mesaj:=Application.MessageBox('Are you sure to delete',
        'Delete',MB_ICONSTOP+MB_YESNO);

                if mesaj=mryes then
```

```
                    begin
                         Table1.Delete;
                         Edit1.Text:=Table1CustomerNo.Text;
                         DateTimePicker1.DateTime:=Table1Date.AsDateTime;
                         Edit2.Text:=Table1Litre.Text;
                         ComboBox1.Text:=Table1TimeSelection.Text;
                          ShowMessage('Deleted');

                    end

               else

          ShowMessage('Canceled');

end;

procedure TForm3.FormCreate(Sender: TObject);
begin

end;

end.
```

Form 4. The Money is Given To Villagers

```
unit Unit4;

interface

uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms,
  Dialogs, StdCtrls, Grids, DBGrids, DB, DBTables, ComCtrls, ExtCtrls;

type
  TForm4 = class(TForm)
    GroupBox1: TGroupBox;
    Label1: TLabel;
    Label4: TLabel;
    Label5: TLabel;
    Edit1: TEdit;
    DateTimePicker1: TDateTimePicker;
    Panel1: TPanel;
    Button1: TButton;
    Button2: TButton;
    Button3: TButton;
    Button4: TButton;
    Button5: TButton;
    Button6: TButton;
    Button7: TButton;
    Table1: TTable;
    DataSource1: TDataSource;
    DBGrid1: TDBGrid;
    Table1CustomerNo: TStringField;
    Table1Date: TDateField;
    Table1GivenMoney: TCurrencyField;
    Edit2: TEdit;
    Edit3: TEdit;
    Label2: TLabel;
    Table1LitrePrice: TCurrencyField;

    procedure Button1Click(Sender: TObject);
    procedure FormClose(Sender: TObject; var Action: TCloseAction);
    procedure Button5Click(Sender: TObject);
    procedure Button2Click(Sender: TObject);
    procedure Button3Click(Sender: TObject);
    procedure Button4Click(Sender: TObject);
    procedure Button6Click(Sender: TObject);
    procedure Button7Click(Sender: TObject);
    procedure FormCreate(Sender: TObject);

private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  Form4: TForm4;

implementation

uses unit1,unit17;
{$R *.dfm}

procedure TForm4.Button1Click(Sender: TObject);
begin

        form1.Show;
        Form4.Hide;

end;
```

```
procedure TForm4.FormClose(Sender: TObject; var Action: TCloseAction);
begin

        form1.Show;

end;

procedure TForm4.Button5Click(Sender: TObject);
begin

        Edit1.Text:='';
        Edit2.Text:='';
        Edit3.Text:='';

end;

procedure TForm4.Button2Click(Sender: TObject);
begin

        Table1.SetKey;
        Table1.FieldByName('CustomerNo').AsString:='';
         Table1.GotoKey;
        if Table1.GotoKey=false Then
                begin

                        Table1.Insert;
                        Table1CustomerNo.AsString:=Edit1.Text;
                        Table1Date.AsDateTime:=DateTimePicker1.DateTime;
                        Table1GivenMoney.AsCurrency:=StrToCurr(Edit2.Text);
                        Table1GivenMoney.DisplayFormat:='###,###,##  YTL';
                        Table1LitrePrice.AsFloat:=StrToFloat(Edit3.Text);
                        Table1.Post;
                End

         Else

                        ShowMessage('Error');

end;

procedure TForm4.Button3Click(Sender: TObject);
begin

        Edit1.Text:=Table1CustomerNo.Text;
        DateTimePicker1.Date:=Table1Date.AsDateTime;
        Edit2.Text:=Table1GivenMoney.Text;
        Edit3.Text:=Table1LitrePrice.Text;
        Table1.Edit;

end;
```

```
procedure TForm4.Button4Click(Sender: TObject);
begin

        table1.ClearFields;
        Table1CustomerNo.AsString:=Edit1.Text;
        Table1Date.AsDateTime:=DateTimePicker1.DateTime;
        Table1GivenMoney.AsCurrency:=StrToCurr(Edit2.Text);
        Table1LitrePrice.AsFloat:=StrToFloat(Edit3.Text);
        Table1.Post;

end;

procedure TForm4.Button6Click(Sender: TObject);

var
  mesaj:integer;

begin

        mesaj:=Application.MessageBox('Are you sure to delete',
        'Delete',MB_ICONSTOP+MB_YESNO);
          if mesaj=mryes then
                begin

                        Table1.Delete;
                        Edit1.Text:=Table1CustomerNo.Text;
                        DateTimePicker1.Date:=Table1Date.AsDateTime;
                        Edit2.Text:=Table1GivenMoney.Text;
                        Edit3.Text:=Table1LitrePrice.Text;

                       ShowMessage('Deleted');

                end

          else

                ShowMessage('Canceled');


end;


procedure TForm4.Button7Click(Sender: TObject);
begin

        Form17.QuickRep1.Preview;

end;
```

```
procedure TForm4.FormCreate(Sender: TObject);
begin

end;

end.
```

Form 5. The Debts to have to pay

```
unit Unit5;

interface

uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms,
  Dialogs, Grids, DBGrids, StdCtrls, DB, DBTables, ExtCtrls;

type
  TForm5 = class(TForm)
    Panel1: TPanel;
    DBGrid1: TDBGrid;
    GroupBox1: TGroupBox;
    Button2: TButton;
    Edit1: TEdit;
    Label1: TLabel;
    ScrollBox1: TScrollBox;
    Button1: TButton;
    Button3: TButton;
    Button4: TButton;
    Button5: TButton;
    DataSource1: TDataSource;
    Query1: TQuery;
    Label3: TLabel;

    procedure Button1Click(Sender: TObject);
    procedure Button2Click(Sender: TObject);
    procedure FormCreate(Sender: TObject);
    procedure Button3Click(Sender: TObject);
    procedure Button4Click(Sender: TObject);
    procedure Button5Click(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var

Form5: TForm5;
  a:Integer;

implementation

uses unit1;
{$R *.dfm}

procedure TForm5.Button1Click(Sender: TObject);
begin

        Form1.Show;
        Form5.Hide

end;


procedure TForm5.Button2Click(Sender: TObject);
begin

if Edit1.Text='' Then

        ShowMessage('Please enter the Customer No')

else
        begin

                Query1.SQL.Clear;
                Query1.SQL.Add('Select
                c.CustomerNo,c.Name,c.Surname,c.village,SUM(p.Litre)*g.LitrePrice From
                customer c,praticalProcedure p,GivenMoney g where
                c.CustomerNo=p.CustomerNo and c.CustomerNo=g.CustomerNo and
                c.CustomerNo=:n Group By c.CustomerNo,c.Name,c.Surname,c.village');
                Query1.ParamByName('n').AsString:=Edit1.Text;
                Query1.Open;

        end;

end;


procedure TForm5.FormCreate(Sender: TObject);
```

```
begin

        Query1.SQL.Clear;
        Query1.DatabaseName:='sutcu';
        Query1.Close;
        Query1.SQL.Clear;
        Query1.SQL.Add('sELECT * fROM customer');
        Query1.Open;

end;

procedure TForm5.Button3Click(Sender: TObject);
begin

        Query1.Close;

        Query1.DatabaseName:='sutcu';
        Query1.SQL.Clear;
        Query1.SQL.Add('Select c.village,SUM(p.Litre)*g.LitrePrice Total From customer
        c,praticalProcedure p,GivenMoney g where c.CustomerNo=p.CustomerNo and
        c.CustomerNo=g.CustomerNo Group By c.village');
        Query1.Open;
end;

procedure TForm5.Button4Click(Sender: TObject);
begin

        Query1.Close;
        Query1.DatabaseName:='sutcu';
        Query1.SQL.Clear;
        Query1.SQL.Add('Select
        c.CustomerNo,c.Name,c.Surname,c.village,SUM(p.Litre)*g.LitrePrice Total From
        customer c,praticalProcedure p,GivenMoney g where c.CustomerNo=p.CustomerNo
        and c.CustomerNo=g.CustomerNo Group By
        c.CustomerNo,c.Name,c.Surname,c.village');
        Query1.Open;
end;

procedure TForm5.Button5Click(Sender: TObject);
begin

        Query1.Close;
        Query1.DatabaseName:='sutcu';
        Query1.SQL.Clear;
        Query1.SQL.Add(' select SUM(p.Litre)*g.LitrePrice Total From  praticalProcedure
        p,GivenMoney g where p.CustomerNo=g.CustomerNo');
```

```
      Query1.Open;

end;

end.
```

Form 6. The Cheques

```
unit Unit6;

interface

uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms,
  Dialogs, Grids, DBGrids, StdCtrls, ComCtrls, DB, DBTables, ExtCtrls;

type
  TForm6 = class(TForm)
    DBGrid1: TDBGrid;
    Table1: TTable;
    DataSource1: TDataSource;
    GroupBox1: TGroupBox;
    Label1: TLabel;
    Label4: TLabel;
    Label5: TLabel;
    Label3: TLabel;
    Edit1: TEdit;
    DateTimePicker1: TDateTimePicker;
    Panel1: TPanel;
    Button1: TButton;
    Button2: TButton;
    Button3: TButton;
    Button4: TButton;
    Button5: TButton;
    Button6: TButton;
    Edit2: TEdit;
    Edit3: TEdit;
    Table1CustomerNo: TStringField;
    Table1BankName: TStringField;
    Table1Dates: TDateField;
    Table1Amount: TCurrencyField;

    procedure Button1Click(Sender: TObject);
    procedure Button5Click(Sender: TObject);
    procedure Button4Click(Sender: TObject);
    procedure Button2Click(Sender: TObject);
    procedure Button3Click(Sender: TObject);
    procedure Button6Click(Sender: TObject);
    procedure FormCreate(Sender: TObject);

  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  Form6: TForm6;

implementation

 uses unit1;
{$R *.dfm}

procedure TForm6.Button1Click(Sender: TObject);
begin

        Table1.Insert;
        Table1CustomerNo.AsString:=Edit1.Text;
        Table1BankName.AsString:=Edit2.Text;
        Table1Dates.AsDateTime:=DateTimePicker1.Date;
        Table1Amount.AsCurrency:=StrToCurr(Edit3.Text);
        Table1Amount.DisplayFormat:='###,###,##  YTL';
         Table1.Post;

end;


procedure TForm6.Button5Click(Sender: TObject);
begin

         Form1.Show;
        Form6.Hide;

end;


procedure TForm6.Button4Click(Sender: TObject);
begin

         Edit1.Text:='';
         Edit2.Text:='';
         Edit3.Text:='';
```

```
end;

procedure TForm6.Button2Click(Sender: TObject);
begin

        Edit1.Text:=Table1CustomerNo.Text;
        Edit2.Text:=Table1BankName.Text;
        DateTimePicker1.DateTime:=Table1Dates.AsDateTime;
        Edit3.Text:=Table1Amount.Text;
        Table1.Edit;

end;

procedure TForm6.Button3Click(Sender: TObject);
var
mesaj:integer;

begin
        mesaj:=Application.MessageBox('Are you sure to delete',
        'Delete',MB_ICONSTOP+MB_YESNO);
 if mesaj=mryes then
   begin

        Table1.Delete;
        Edit1.Text:=Table1CustomerNo.Text;
        Edit2.Text:=Table1BankName.Text;
        DateTimePicker1.DateTime:=Table1Dates.AsDateTime;
        Edit3.Text:=Table1Amount.Text;

         ShowMessage('Deleted');

 End

 Else

       ShowMessage('Canceled');

end;


procedure TForm6.Button6Click(Sender: TObject);
begin

      Table1.ClearFields;
      Table1CustomerNo.AsString:=Edit1.Text;
      Table1BankName.AsString:=Edit2.Text;
      Table1Dates.AsDateTime:=DateTimePicker1.Date;
      Table1Amount.AsCurrency:=StrToCurr(Edit3.Text);
```

```
        Table1Amount.DisplayFormat:='###,###,##  YTL';

        Table1.Post;

end;


procedure TForm6.FormCreate(Sender: TObject);
begin

end;

end.
```

Form 7. Query Of The Cheque Form

```
unit Unit7;

interface

uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms,
  Dialogs, StdCtrls, Grids, DBGrids, ComCtrls, DB, DBTables, ExtCtrls;

type
  TForm7 = class(TForm)
    DBGrid1: TDBGrid;
    DataSource1: TDataSource;
    Query1: TQuery;
    GroupBox1: TGroupBox;
    Label1: TLabel;
    Edit1: TEdit;
    Panel1: TPanel;
    Button3: TButton;
    Button2: TButton;
    Button4: TButton;
    Button5: TButton;
    Button1: TButton;

    procedure Button3Click(Sender: TObject);
    procedure Button1Click(Sender: TObject);
    procedure FormCreate(Sender: TObject);
    procedure Button2Click(Sender: TObject);
    procedure Button4Click(Sender: TObject);
    procedure Button5Click(Sender: TObject);
    procedure FormActivate(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  Form7: TForm7;

implementation
uses unit1;

{$R *.dfm}

procedure TForm7.Button3Click(Sender: TObject);
begin

        form1.Show;
        Form7.Hide;

end;


procedure TForm7.Button1Click(Sender: TObject);
begin

        if Edit1.Text='' Then

                ShowMessage('Please enter the Customer No')

        else

        begin
                Query1.SQL.Clear;
                Query1.SQL.Add('Select
                c.CustomerNo,c.Name,c.Surname,c.village,SUM(p.Litre) From customer
                c,praticalProcedure p,GivenMoney g where c.CustomerNo=p.CustomerNo and
                c.CustomerNo=g.CustomerNo and c.CustomerNo=:n Group By
                c.CustomerNo,c.Name,c.Surname,c.village');
        Query1.ParamByName('n').AsString:=Edit1.Text;
        Query1.Open;

        end;

end;
```

```
procedure TForm7.FormCreate(Sender: TObject);
begin

        Query1.SQL.Clear;
        Query1.DatabaseName:='sutcu';
        Query1.Close;
        Query1.SQL.Clear;
        Query1.SQL.Add('sELECT * fROM sutalim');
        Query1.Open;

end;

procedure TForm7.Button2Click(Sender: TObject);
begin

        Query1.Close;
        Query1.DatabaseName:='sutcu';
        Query1.SQL.Clear;
        Query1.SQL.Add('Select c.village,SUM(p.Litre) Total From customer
        c,praticalProcedure p,GivenMoney g where c.CustomerNo=p.CustomerNo and
        c.CustomerNo=g.CustomerNo Group By c.village');
        Query1.Open;


end;


procedure TForm7.Button4Click(Sender: TObject);
begin

        Query1.Close;
        Query1.DatabaseName:='sutcu';
        Query1.SQL.Clear;
        Query1.SQL.Add('Select c.CustomerNo,c.Name,c.Surname,c.village,SUM(p.Litre)
        Total From customer c,praticalProcedure p,GivenMoney g where
        c.CustomerNo=p.CustomerNo and c.CustomerNo=g.CustomerNo Group By
        c.CustomerNo,c.Name,c.Surname,c.village');
        Query1.Open
end;

procedure TForm7.Button5Click(Sender: TObject);
begin

        Query1.Close;
        Query1.DatabaseName:='sutcu';
        Query1.SQL.Clear;
        Query1.SQL.Add(' select SUM(p.Litre) Total From  praticalProcedure p,GivenMoney
        g where p.CustomerNo=g.CustomerNo');
        Query1.Open;

end;


procedure TForm7.FormActivate(Sender: TObject);
begin

        Query1.SQL.Clear;
        Query1.DatabaseName:='sutcu';
        Query1.Close;
        Query1.SQL.Clear;
        Query1.SQL.Add('sELECT * fROM customer');
        Query1.Open;

end;

end.
```

Form 10. Query Of The Cheque Form

```
unit Unit10;

interface

uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms,
  Dialogs, Grids, DBGrids, StdCtrls, DB, DBTables, ExtCtrls, ComCtrls;

type
  TForm10 = class(TForm)
    Query1: TQuery;
    DataSource1: TDataSource;

    DBGrid1: TDBGrid;
    GroupBox1: TGroupBox;
    Label1: TLabel;
    Edit1: TEdit;
    Button4: TButton;
    Panel1: TPanel;
    Button3: TButton;
    Button1: TButton;
    Button2: TButton;

    procedure Button1Click(Sender: TObject);
    procedure Button2Click(Sender: TObject);
    procedure Button3Click(Sender: TObject);
    procedure Button4Click(Sender: TObject);
    procedure FormActivate(Sender: TObject);
    procedure FormCreate(Sender: TObject);

  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  Form10: TForm10;

implementation
 uses unit1;

{$R *.dfm}

procedure TForm10.Button1Click(Sender: TObject);
begin

        Query1.Close;
        Query1.DatabaseName:='sutcu';
        Query1.SQL.Clear;
        Query1.SQL.Add('Select SUM(Amount) Toplam From Cheques  ');
        Query1.Open;

end;


procedure TForm10.Button2Click(Sender: TObject);
begin

        Query1.Close;
        Query1.DatabaseName:='sutcu';
        Query1.SQL.Clear;
        Query1.SQL.Add('Select CustomerNo,BankName,Dates,Amount From cheques order
by Dates');
        Query1.Open;

end;


procedure TForm10.Button3Click(Sender: TObject);
begin

        Form1.Show;
        Form10.Hide;
```

```
end;

procedure TForm10.Button4Click(Sender: TObject);
begin
        if Edit1.Text='' Then

                ShowMessage('Please enter the Customer No')
        Else

        begin
                Query1.SQL.Clear;
                Query1.SQL.Add('Select CustomerNo,BankName,Dates,Amount From
                cheques where CustomerNo=:n Group By
                CustomerNo,BankName,Dates,Amount');
                Query1.ParamByName('n').AsString:=Edit1.Text;
                Query1.Open;
        End

end;

procedure TForm10.FormActivate(Sender: TObject);
begin

        Query1.Close;
        Query1.DatabaseName:='sutcu';
        Query1.SQL.Clear;
        Query1.SQL.Add('Select *  From Cheques  ');
        Query1.Open;

end;

procedure TForm10.FormCreate(Sender: TObject);
begin

end;

end.
```

Form 11. Client Record Report

```
unit Unit11;

interface

uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms,
  Dialogs, DB, DBTables, ExtCtrls, QuickRpt, QRCtrls;

type
  TForm11 = class(TForm)
    QuickRep1: TQuickRep;
    Table1: TTable;
    TitleBand1: TQRBand;
    QRLabel1: TQRLabel;


    ColumnHeaderBand1: TQRBand;
    QRLabel2: TQRLabel;
    QRLabel3: TQRLabel;
    QRLabel4: TQRLabel;
    QRLabel5: TQRLabel;
    QRLabel6: TQRLabel;
    QRLabel7: TQRLabel;
    DetailBand1: TQRBand;
    QRDBText1: TQRDBText;
    QRDBText2: TQRDBText;
    QRDBText3: TQRDBText;
    QRDBText4: TQRDBText;
    QRDBText5: TQRDBText;
    QRDBText6: TQRDBText;
    PageHeaderBand1: TQRBand;
    QRSysData1: TQRSysData;
    PageFooterBand1: TQRBand;
    QRSysData2: TQRSysData;
    QRShape2: TQRShape;
    QRShape1: TQRShape;
    QRLabel8: TQRLabel;
    QRDBText7: TQRDBText;

  procedure FormCreate(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;


var
```

```
  Form11: TForm11;

implementation

{$R *.dfm}

procedure TForm11.FormCreate(Sender: TObject);
begin

end;

end.
```

Form 15. Login

```
unit Unit15;

interface

uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms,
  Dialogs, DB, DBTables, StdCtrls, jpeg, ExtCtrls;

type

TForm15 = class(TForm)
   Edit1: TEdit;
   Edit2: TEdit;
   Button1: TButton;
   Button2: TButton;
   Button3: TButton;
   Table1: TTable;
   DataSource1: TDataSource;
   Image1: TImage;
   Label1: TLabel;
   Label2: TLabel;
   Table1Name: TStringField;
   Table1Password: TStringField;

   procedure Button1Click(Sender: TObject);
   procedure Button2Click(Sender: TObject);
   procedure Button3Click(Sender: TObject);
   procedure FormCreate(Sender: TObject);
```

```
private
   { Private declarations }
  public
   { Public declarations }
  end;

var
  Form15: TForm15;

implementation
 uses unit1;

{$R *.dfm}

procedure TForm15.Button1Click(Sender: TObject);
var

ad:String;
sifre:String;

begin
        Table1.First;
        while not Table1.Eof do
        begin
        ad:=Table1Name.AsString;
        sifre:=Table1Password.AsString;
        if (Edit1.Text=ad) And (Edit2.Text=sifre) Then
                begin

                        Form1.Show;
                        Form15.Hide;
                        exit;

                end;

        Table1.Next;
end;

application.MessageBox('Wrong Username or Password ','Warning',16);

        Edit1.Clear;
        Edit2.Clear;
        Edit1.SetFocus;
end;
```

```
procedure TForm15.Button2Click(Sender: TObject);
begin

        Edit1.Clear;
        Edit2.Clear;
        Edit1.SetFocus;

end;

procedure TForm15.Button3Click(Sender: TObject);
begin

        halt;

end;

procedure TForm15.FormCreate(Sender: TObject);
begin

        Edit2.PasswordChar:='*';

end;

end.
```

Form 16. Setting Password

```
unit Unit16;

interface

uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms,
  Dialogs, StdCtrls, DB, DBTables, Grids, DBGrids;

Type

  TForm16 = class(TForm)
    Edit1: TEdit;
    Edit2: TEdit;
    Edit3: TEdit;
    Button1: TButton;
    Button2: TButton;
    Button3: TButton;
    Label1: TLabel;
    Label2: TLabel;
    Label3: TLabel;
    DataSource1: TDataSource;
    DBGrid1: TDBGrid;
    Table1: TTable;
    Button4: TButton;
    Table1Name: TStringField;
    Table1Password: TStringField;

    procedure Button1Click(Sender: TObject);
    procedure Button3Click(Sender: TObject);
    procedure FormCreate(Sender: TObject);

    procedure Button4Click(Sender: TObject);
    procedure Button2Click(Sender: TObject);

  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  Form16: TForm16;

implementation
  uses unit1;

{$R *.dfm}

procedure TForm16.Button1Click(Sender: TObject);
begin

        if Edit2.Text=Edit3.Text then
                begin
                        Table1.Insert;
                        Table1Name.AsString:=Edit1.Text;
                        Table1Password.AsInteger:=StrToInt(Edit2.Text);
                        Table1.Post
                end
        else
                begin

                        ShowMessage('Enter a valid passport or user name try again');

        Edit1.Clear;
        Edit2.Clear;
        Edit3.Clear;
        Edit1.SetFocus;

end;
```

```
end;

procedure TForm16.Button3Click(Sender: TObject);
var
mesaj:Integer;

begin
        mesaj:=Application.MessageBox('Are sure to delete this user',
      'Kayıt sil',MB_ICONSTOP+MB_YESNO);

end;

procedure TForm16.FormCreate(Sender: TObject);
begin

      Edit2.PasswordChar:='*';
      Edit3.PasswordChar:='*';

end;

procedure TForm16.Button4Click(Sender: TObject);
begin

      Form1.Show;
      Form16.Hide;

end;

procedure TForm16.Button2Click(Sender: TObject);
begin

      Edit1.Text:=Table1Name.Text;
      Edit2.Text:=Table1Password.Text;
      Table1.Edit;

end;

end.
```

Form 17. The Money is Given To Villagers Report

```
unit Unit17;

interface

uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms,
  Dialogs, DB, DBTables, QRCtrls, QuickRpt, ExtCtrls;
type
  TForm17 = class(TForm)
    QuickRep1: TQuickRep;
    PageHeaderBand1: TQRBand;
    QRSysData1: TQRSysData;
    TitleBand1: TQRBand;
    QRLabel1: TQRLabel;
    ColumnHeaderBand1: TQRBand;
    QRLabel2: TQRLabel;
    QRLabel3: TQRLabel;
    QRLabel4: TQRLabel;
    QRLabel5: TQRLabel;
    DetailBand1: TQRBand;
    QRDBText1: TQRDBText;
    QRDBText2: TQRDBText;
    QRDBText3: TQRDBText;
    QRDBText4: TQRDBText;
    Table1: TTable;
    PageFooterBand1: TQRBand;
    QRSysData2: TQRSysData;
    procedure FormCreate(Sender: TObject);

  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  Form17: TForm17;

implementation

{$R *.dfm}

procedure TForm17.FormCreate(Sender: TObject);
begin

end;
end.
```