Poul Klausen

C# 6

ADO.NET and Database Applications

Software Development

bookboon
LEARNING

POUL KLAUSEN

C# 6: ADO.NET AND DATABASE APPLICATIONS

SOFTWARE DEVELOPMENT

C# 6: ADO.NET and Database Applications: Software Development
1st edition

ISBN 978-87-403-3509-5

CONTENTS

FOREWORD

This book is the sixth in a series of books on software development. In the previous books I have generally dealt with object-oriented programming in C#. This book mainly deals with database programming, but seen from C# and how to write a program that uses a database. The book thus assumes that the reader has a basic knowledge of databases and including SQL. With regard to the latter, the book has an appendix which provides a brief introduction to SQL. The book also deals with WPF and data binding and related to the design pattern *Model-View-ViewModel*, as these concept are very closely linked to the development of database programs which often use the WPF component *DataGrid* to present data.

As the title says this series of books deals with software development, and the goal is to teach the reader how to develop applications in C#. You can learn that by reading about the subject and by studying complete sample programs, but most importantly by yourself to do it, and write your own programs from scratch. Therefore, an important part of the books is exercises and problems, where the reader has to write programs that correspond to the substance treated in the books. All books in the series are built around the same skeleton and will consist of text and examples, exercises and problems that are placed in the text where they naturally belongs. The difference between exercises and problems is that the exercises largely deals with repetitions of the substance presented in the text, and furthermore it is relatively accurately described what to do. Problems in turn, are more loosely described and are typically a little bigger and there is rarely any clear best solution. These are books to be read from start to finish, but the many code examples, including exercises and problems plays a central role, and it is important that the reader predict in detail studying the code for the many examples and also solves the exercises and problems or possibly just studying the recommended solutions.

All books ends with a larger sample program, which focus primarily is on the process and an explanation of how the program is written. On the other hand appears the code only to a limited extent, if at all, and the reader should instead study the finished program code, perhaps while testing the program. In addition to show the development of programs that are larger than the examples, which otherwise is presented, the aim of the concluding examples also is to show program examples from varying fields of application.

All sample programs are developed and tested on a Windows machine, and the development tool is Visual Studio. Therefore, you must have Visual Studio installed and Microsoft provides a free version that is fully adequate. Visual Studio is an integrated development environment that provides all the tools needed to develop programs. I do not want to deal with the use of Visual Studio, which is quite straightforward, and is a program that you quickly become comfortable with.

Finally a little about what the books are not. It is not "a how to write" or for that matter a reference manuals to C#, but it is as the title says books on software development. It is my hope that the reader when reading the books and through the many examples can find inspiration for how to write good programs, but also can be used as a source collection with a number of examples of solutions to concrete everyday programming problems that you regularly face as a software developer.

1 INTRODUCTION

This book deals with how to write programs that uses databases. The book is in no way a review of databases and design of databases, including SQL, and as mentioned in the foreword, the book requires knowledge of the most common concepts regarding SQL. If the reader does not have this knowledge, the book's appendix gives a brief introduction to SQL. In relation to .NET and C#, access to databases is done by means of a relatively large number of classes, which are collectively referred to as ADO, and the vast majority of the book therefore deals with how to use these classes. It is a bit like learning WPF, where a large part of the task consists of learning about classes in a framework and how they are used.

All programs work on data, and to a large extent it involves loading data stored on the hard disk, manipulating that data and storing them again. In principle, you could (as I did in the previous books in this series) use ordinary files, but for programs that have to manipulate a lot of data (and many programs have), you will in practice always save data in databases. Database programming is thus one of the key topics in programming.

In order to write database programs, there are two things that you need to have in place. You must have a database system and you must have a database. As for the first, there are many options, similar to the fact that there are several database providers, and as examples I can mention

- DB2
- Oracle
- SQL Server
- MySQL

Of course, these database systems are different, but they are all SQL databases, and if you stick to standard SQL, the programmer does not see the big difference, and the same program can be used with very few changes against a different database than the one the program is written for - yes not enough, you can actually write the program so that it is independent of the underlying database product. Since this book deals with C#, I primarily want to use Microsoft's database products called Microsoft SQL Server. There are more versions, but for this book and the following books you can use *Local SQL Server* that is a free version used on a local machine for program developers. I will later use other database products, but the following assumes that you have Local SQL Server installed and also *SQL Server Management Studio*, which is a tool for creating and maintaining databases.

This book deals with the development of database programs and including the most important WPF components and program architecture that are also needed. There is no room to deal with all database concepts, and some that the concepts are postponed to the next book.

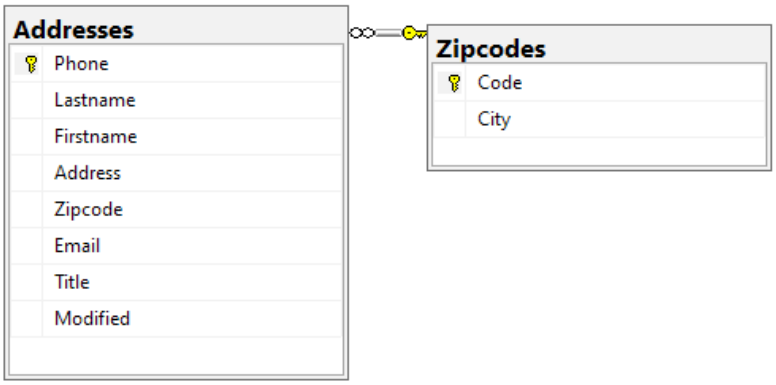
Then there is the databases and the examples in this book will use two

- *Contacts*
- *Books*

Both are relatively simple databases, but conversely sufficiently complex to show how to write database programs.

1.1 CONTACTS

This database consists of two tables:



Below are parts of a script that creates the database and the two tables:

```
create table Zipcodes (
  Code nchar(4) primary key,
  City nvarchar(30) not null
);

create table Addresses (
  Phone nchar(8) primary key,
  Lastname nvarchar(30) not null,
  Firstname nvarchar(50),
  Address nvarchar(50),
  Zipcode nchar(4),
  Email nvarchar(100),
  Title nvarchar(50),
  Modified timestamp,
  foreign key (Zipcode) references Zipcodes
);
```

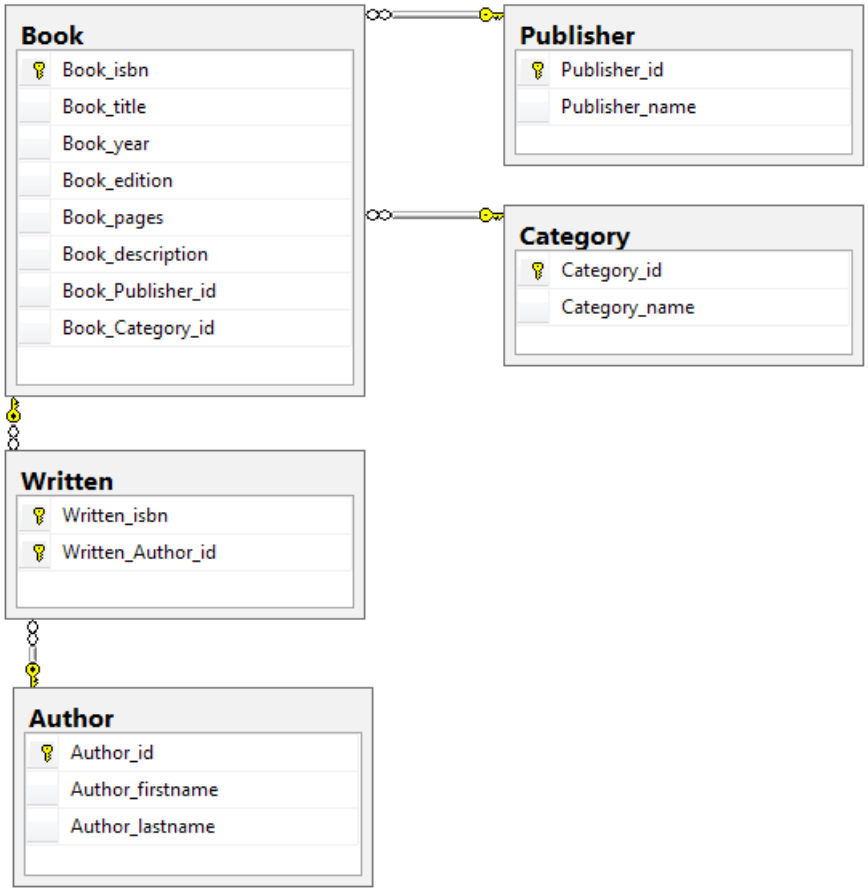
From the script you can see what types each columns has, as well as the *Zipcode* column in the *Addresses* table is a foreign key to the *Zipcodes* table. You should also note the primary keys for the two tables.

The database is used for personal contacts and the meaning of the individual columns should appear from the name - except for the last one in the *Addresses* table, which you simply have to ignore so far. The *Zipcodes* table contains Danish zip codes, while the *Addresses* table contains approx. 10000 people (the same persons as used in the previous book). The large number of addresses is selected to illustrate programs that extract large amounts of data. It should be mentioned that these are not real persons.

The script is called *CreateContacts.sql* and to fill data in the database the script has an SQL INSERT statement for each zip code and each person. It is thus a very comprehensive script.

1.2 BOOKS

This database has 5 tables and is a database containing books:



The primary table is *Book* which via foreign keys is related to tables with respectively *publishers* and *categories*. This corresponds to the fact that each book is published by a specific publisher and that each book is associated with a specific category. Finally, a book can have one or more authors, so the table *Written* is a many-to-many relationship between *Book* and *Author*.

Expressed in SQL, the database can be created as follows:

```
create table Publisher
(
  Publisher_id                      int identity(1,1),
  Publisher_name                    nvarchar(100) not null,
  primary key (Publisher_id)
);
```

```
create table Author
(
  Author_id               int identity(1,1),
  Author_firstname        nvarchar(100),
  Author_lastname         nvarchar(50) not null,
  primary key (Author_id)
);

create table Category
(
  Category_id             int identity (1,1),
  Category_name           nvarchar(100) not null,
  primary key (Category_id)
);

create table Book
(
  Book_isbn               nvarchar(20),
  Book_title              nvarchar(100) not null,
  Book_year               int,
  Book_edition            int,
  Book_pages              int,
  Book_description        ntext,
  Book_Publisher_id       int,
  Book_Category_id        int,
  primary key (Book_isbn),
  foreign key (Book_Publisher_id) references Publisher,
  foreign key (Book_Category_id) references Category
);

create table Written
(
  Written_isbn            nvarchar(20),
  Written_Author_id       int,
  primary key (Written_isbn, Written_Author_id),
  foreign key (Written_isbn) references Book,
  foreign key (Written_Author_id) references Author
);
```

From the script you can see the type of the individual columns, keys, etc.

The content of the database is 135 books. This time it is a matter of real data - even if it is a bit old, but it does not matter what follows.

To create the database and fill it with data you can use a backup which exists as a file in the book's source folder.

1.3 ADO

As mentioned, .NET has a framework of classes, so you as a programmer can work with databases from a program, and it is this API that is called ADO. It consists of several namespaces and the aim of the following is to give an overview of the most important and give some examples of how to apply the classes.

System.Data is a namespace that contains all the basic types that are independent of specific databases and that give the programmer an object-oriented approach to databases. Most database applications will use types from this namespace.

At some point, the types have to become concrete and target specific products. For example there is a namespace *System.Data.SqlClient* that contains types that specifically target SQL Server. Each of these namespaces contains types for a so-called data provider, similar to containing types for a specific database standard / database product, and ADO.NET also contains several and including a provider for Oracle. The types of these provider namespaces are derived from the same base types that are gathered in a namespace named *System.Data.Common*. This namespace is important as it allows you to write code that is independent of a specific provider and thus applications that can connect to databases from different suppliers.

If you now have to use a database for which the framework does not have provider software, in almost all cases you will be able to download and install the necessary software. An example could be MySQL. After the software is installed, the database in question is used in exactly the same way as the databases that the framework was born to support.

The individual namespaces contain types for two very different ways of accessing a database called respectively connected and disconnected access. The two ways cover how the application connects to the physical database and each way has its advantages and disadvantages. Connected access means that the application connects to the underlying database and that connection is kept open while the application performs the desired operations, after which it is the job of the application to close and disconnect. However, if disconnected access is used, some form of adapter is inserted between the application and the database. This adapter establishes the connection to the database and extracts data that it fills into memory objects, after which the connection is automatically closed, and data can then be manipulated in memory without any network traffic. After data is manipulated, all changes can be written back to the database and the adapter will automatically open the connection and close it again after data is saved. In the next two chapters I will look at connected ADO alone.

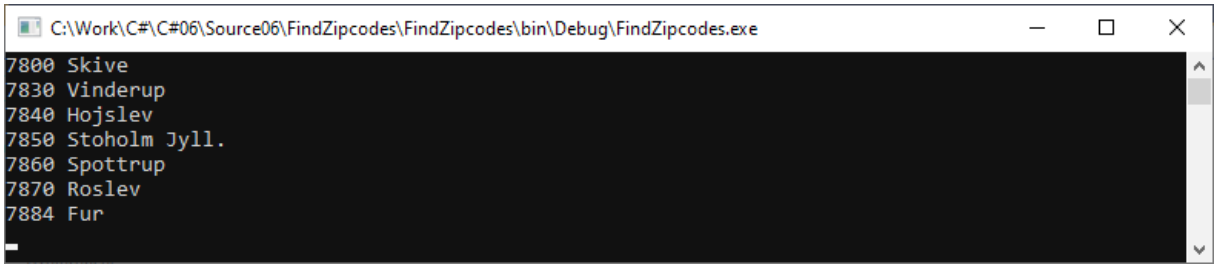
2 DATABASE PROGRAMS

To get started, I will begin with some simple console programs that have only one purpose, namely to show how to perform database operations from a C# program, and thus what you have to write.

The following program uses the *Contacts* database and extracts all rows from the *Zipcodes* table, where the zip code starts with 78. The program should only show how to open a connection to the database, execute a SQL SELECT statement and print the result on the screen. The program is a console application and consists solely of a *Main()* method, thus it is the simplest database application one can imagine. The example is also typical of what it means to use connected database access and also what types of ADO it requires.

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

namespace FindZipcodes
{
  class Program
  {
    static void Main(string[] args)
    {
      SqlConnection connection = new SqlConnection(
        "Data Source=PA1;Initial Catalog=Contacts;Integrated
        Security = True");
      SqlCommand cmd =
        new SqlCommand("SELECT * FROM Zipcodes WHERE
        Code LIKE '78%'", connection);
      connection.Open();
      SqlDataReader reader = cmd.ExecuteReader();
      while (reader.Read())
        Console.WriteLine("{0} {1}", reader["Code"], reader["City"]);
      connection.Close();
      Console.ReadLine();
    }
  }
}
```

If you run the program, the result is the following as there are 7 zip codes where the code starts with 78:



First, note the namespace *System.Data.SqlClient* that contains types for a SQL Server database. Three classes from this namespace are used:

- *SqlConnection*, representing a connection to a SQL Server database
- *SqlCommand*, which represents a SQL command
- *SqlDataReader*, which represents the result of a SQL SELECT statement

To open a connection to a database, you must specify a connection string, which generally consists of several key / value pairs separated by semicolons. The contents of this string depend on the database and in this case there are three parts:

- *PA1*, which is the name of a SQL server on my machine (to test the program on your machine you have to choose a different name which is the name of database server on your machine)
- *Contacts*, which is the name of the database
- that *integrated* security must be used, which means access rights for the current user of the machine

You can specify this connection string as a parameter to the constructor in the class *SqlConnection* (used above) or assign it to a property.

After a connection object is defined, an SQL command, that is represented as a *SqlCommand* object must be defined. The SQL expression is passed as a parameter to the constructor (or by using a property on the object), and the SQL expression must be specified as a string, as well as the connection to which the command is to be sent. Before the command can be executed, the database connection must be opened and the command is then executed using the method *ExecuteReader()*. This method returns a so-called data reader, which here has the type *SqlDataReader*, and it is an object that represents the data (rows) extracted by the database. The object allows you to traverse the rows, but it is important to note that the object is readonly, so that the individual elements cannot be modified, and that you can only traverse forwards. Notice how to use the method *Read()* to test whether you have reached the end of the reader. Also note how to refer to the individual elements in a row as an index operator override and where to specify column names from the database as index. However, it is also possible to specify a numerical index. Finally, the connection to the database must be closed. It is important to remember that.

The above is an example of connected access. This means that the connection to the database is kept open until it is explicitly closed (closed). It means that the connection is open while traversing the reader. The reader does not load all data into memory so that you can then traverse it, but read directly from the physical database. Therefore, one cannot close the connection within the while loop. If the task is to execute a SQL SELECT statement and traverse the data being read (and you often need that), the use of a data reader is the most efficient way. Just be aware that you only have the opportunity to read data.

As a final comment on the first database example, the program has two shortcomings. Firstly, the connection string is hard-coded which means that you cannot move the database without changing the code. Of course, it's unfortunate. The other problem is that you cannot change provider without reprogramming the program (use other types). For example, if you change from a SQL Server to a MySQL database, it is necessary to change the code. The program is too closely tied to the specific provider. Both problems can be solved and become what follows, and when they are not above, it is just to keep it all as simple as possible. Finally, it should also be mentioned that in practice the code requires an exception handling.

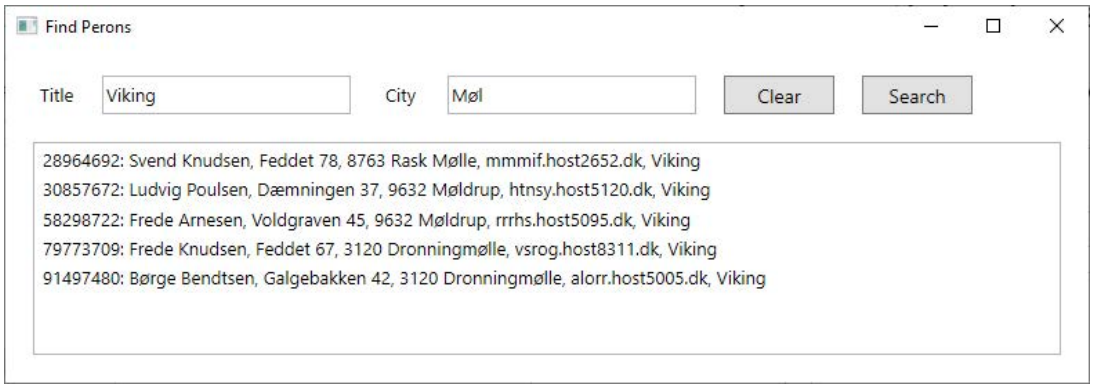
EXERCISE 1: FIND BOOKS

You must write a program which finds and prints the titles and number of pages for all books in the *Books* database where the category name is *Matematik* (math books). The program is basically identical to the above, but you need to use a different database and this is a more complicated SQL expression as it is necessary to join between two tables. When traversing the reader object, this time refer to the individual columns using a numerical index.

EXERCISE 2: FIND PERSONS

Write a program that opens the following window:

The program must search persons in the *Contacts* database. If the user enter text in the two *TextBox* controls the program must fine all persons where title and city name contains the search text. An example could be:



2.1 OPEN DATABASES

The project *FindZipcodes1* is exactly the same program as the first program in this chapter with the only difference that the code is written slightly differently:

```
class Program
{
  static void Main(string[] args)
  {
    string connectionStr =
    "Data Source=PA1;Initial Catalog=Contacts;
    Integrated Security = True";
  using (SqlConnection connection = new SqlConnection(connectionStr))
  {
    try
    {
      string sql = "SELECT * FROM Zipcodes WHERE Code LIKE '78%'";
      SqlCommand cmd = new SqlCommand(sql, connection);
      connection.Open();
      using (SqlDataReader reader = cmd.ExecuteReader())
      {
        while (reader.Read())
          Console.WriteLine("{0} {1}", reader["Code"], reader["City"]);
        }
      }
      catch (Exception ex)
      {
        Console.WriteLine(ex.Message);
      }
    }
    Console.ReadLine();
  }
}
```

The connection string is this time defined as a variable instead of directly typed inline as an actual parameter. The rationale is readability only, as connection strings in practice and for other database products can be long and complex. Next the connection is created in a *using* statement which ensures that an open connection is automatically closed. In the same way the *SqlDataReader* object is created in a *using* statement that ensures that this object also automatically is closed. If a database connection is closed an open reader for this connection will also be closed, but if you has to open another reader for the same open connection, you must first ensure that the first connection is closed and if you always opens a reader with a *using* statement you can be sure it is closed when no longer needed.

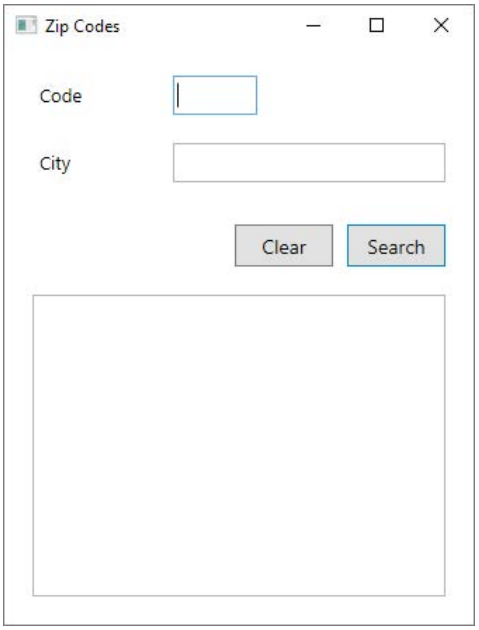
In the above example the code in the first *using* statement are paced in a *try / catch*. One can discuss it reasonably in that since the only thing that can cause an exception is that I have written the SQL term incorrectly. Still, it is recommended to test for an exception as the database and its names may change outside of the program. Also note that this time I have defined the SQL code as a variable. The reason is again readability as SQL expressions can become very long ok complex.

3 THE BASIC SQL STATEMENTS

The basic database operations are

- SELECT
- INSERT
- UPDATE
- DELETE

and above I have shown how to perform a SQL SELECT. It is absolutely the most complex (has a very comprehensive SQL syntax) of the four SQL statements, but seen from a C# program there is nothing more to add than what the above examples shows, but as an example I will show a program to search the *Zipcodes* table in the database *Contacts*. The program does not add anything new but shows some guidelines for using a *DataReader*. The program is called *Zipcodes* and opens the following window:



The window has two fields for search criteria and if you click on *Search* the program updates the lower *ListBox* with the zip codes where the code starts with the search criteria and the city name contains the search criteria.

I should not show the XML code for the Window here as there is nothing new. When searching the database a *DataReader* returns a collection of rows and to use these rows in a program you will usually define a type that represents the individual rows as objects. In this case, I have added a class *Zipcode*:

```
class Zipcode
{
  public string Code { get; set; }
  public string City { get; set; }

  public Zipcode()
  {
  }
  public override string ToString()
  {
    return Code + " " + City;
  }
}
```

In this case it is a very simple class, but it is clear that in other cases the class could be more comprehensive and have more properties and methods. The program must perform database operations and in this case only one, an operation which search the database and then performs a SQL SELECT. It is recommended to place these database operations into one or more database classes, as this can facilitate program maintenance. I have therefore added the following class to the program:

```
using System.Collections.Generic;
using System.Data;
using System.Data.SqlClient;

namespace Zipcodes
{
  class DB
  {
    private SqlConnection connection = new SqlConnection(
      "Data Source=PA1;Initial Catalog=Contacts;
      Integrated Security = True");
```

```
    public List<Zipcode> Find(string code, string city)
    {
      List<Zipcode> list = new List<Zipcode>();
      try
      {
        SqlCommand cmd = new SqlCommand("SELECT *
        FROM Zipcodes WHERE Code LIKE '"
          + code + "%' AND city LIKE '%" + city + "%'", connection);
        connection.Open();
        SqlDataReader reader = cmd.ExecuteReader();
        while (reader.Read()) list.Add(
          new Zipcode { Code = reader.GetString(0),
          City = reader.GetString(1) });
      }
      catch
      {
        list.Clear();
      }
      finally
      {
        if (connection.State == ConnectionState.
        Open) connection.Close();
      }
      return list;
    }
  }
}
```

The class has only one variable which is the connection to the database, and there is only one method. The method is simple enough, but there are still some things to keep in mind:

1. The method returns a list with objects, one object for each row found in the database table.
2. The SELECT statement is build using *string* operations, and it can be demanding to do it without error, especially when building more complicated SELECT statements. I'll show you later how to do this using SQL statements with parameters.
3. A *DataReader* has many methods and for example *GetString()* which returns the value of a column for a given index as a *string*. There are other methods for the other simple data types and using these methods you don't have to type cast the result from the index operator.
4. The method *Find()* opens the connection to the database and to ensure the connection is closed after the operation is performed it is closed in the *finally* block.

With these two classes it is simple to write program code:

```
public partial class MainWindow : Window
{
  private DB db = new DB();

  public MainWindow()
  {
    InitializeComponent();
  }

  private void cmdClear_Click(object sender, RoutedEventArgs e)
  {
    txtCode.Clear();
    txtCity.Clear();
    txtCode.Focus();
  }

  private void cmdSearch_Click(object sender, RoutedEventArgs e)
  {
    lstCodes.ItemsSource = null;
    lstCodes.ItemsSource = db.Find(txtCode.
    Text.Trim(), txtCity.Text.Trim());
    txtCode.Focus();
  }

  private void Window_Activated(object sender, EventArgs e)
  {
    txtCode.Focus();
  }
}
```

There is not much to note, but note that the class has an instance of the class *DB*, and also how the object is used in the event handler for the button *Search*. Note also the first statement in this event handler. It should not be necessary, but should be there as I have to deal with data binding.

In the rest of this chapter I will present how to use the other database operations.

3.1 PARAMETRIZED SQL COMMANDS

I will start with a new console application program which will be used in the rest of this chapter. The project is called *SqlCommands*. First I have added the classes *Zipcode* and *DB* from the above program. Then I have added the following class representing a *Person*:

```
class Person
{
  public string Phone { get; set; }
  public string Firstname { get; set; }
  public string Lastname { get; set; }
  public string Address { get; set; }
  public Zipcode Zipcode { get; set; }
  public string Mail { get; set; }
  public string Title { get; set; }

  public Person()
  {
  }

  public override string ToString()
  {
    StringBuilder builder = new StringBuilder();
    builder.Append(Phone);
    builder.Append(": ");
    builder.Append(Firstname);
    builder.Append(' ');
    builder.Append(Lastname);
    builder.Append(", ");
    builder.Append(Address);
    builder.Append(", ");
    builder.Append(Zipcode.Code);
    builder.Append(' ');
    builder.Append(Zipcode.City);
    builder.Append(", ");
    builder.Append(Mail);
    builder.Append(", ");
    builder.Append(Title);
    return builder.ToString();
  }
}
```

The class is a simple model class for a *Person*.

As the next I have expanded the class *DB* with a method

```
public Person GetPerson(string phone)
```

That is a method which return the person with a specific phone number and then the person with a specific key. There is at most one person with this phone number when *Phone* is primary key.

It is quite simple to write a SQL expression for this query, but this time I use a parameterized SQL expression. A SQL parameter is an object with the type *SqlParameter*, and the following method creates such a parameter:

```
private SqlParameter CreateParam(string name,
object value, SqlDbType type)
{
  SqlParameter param = new SqlParameter(name, type);
  param.Value = value;
  return param;
}
```

where *name* is the name of the parameter, *value* is the value for the parameter and *type* is the type of the parameter which is the type of a column in a database table. You can then write the method *GetPerson()* as:

```
public Person GetPerson(string phone)
{
  try
  {
    SqlCommand command = new SqlCommand("SELECT
    Phone, Firstname, Lastname,
      Address, Zipcode, City, EMail, Title FROM Addresses, Zipcodes
      WHERE Phone = @Phone AND Zipcode = Code", connection);
    command.Parameters.Add(CreateParam("@
    Phone", phone, SqlDbType.NVarChar));
    connection.Open();
    SqlDataReader reader = command.ExecuteReader();
    if (reader.Read()) return new Person { Phone = reader.GetString(0),
      Firstname = reader.GetString(1), Lastname = reader.GetString(2),
      Address = reader.GetString(3), Zipcode =
      new Zipcode { Code = reader.GetString(4),
      City = reader.GetString(5) },
      Mail = reader.GetString(6), Title = reader.GetString(7) };
  }
  catch
  {
  }
  finally
  {
    if (connection.State == ConnectionState.Open) connection.Close();
  }
  throw new Exception("Person not found");
}
```

The syntax for a parameter in a SQL expression is for example *@Phone* and is used in the WHERE part of the expression. The parameter must be assign a value before the expression is performed and it happens to add a *SqlParameter* object to collection for a *SqlCommand* object. It is here I use the method *CreateParam()*. Here you must note the type *SqlDbType* which defines types used for columns in the database tables. When the parameters are initialized (and in this case only one) the command is performed as before and the result is a *DataReader*. If the reader has rows, and there can only be one a *Person* object is created and returned. If no rows are found ether because of an error or because the database does not contain a person with the current phone number the method throws an exception.

The following test method uses the method *GetPerson()*:

```
static void Test1()
{
  try
  {
    DB db = new DB();
    Console.WriteLine(db.GetPerson("10075420"));
  }
  catch (Exception ex)
  {
    Console.WriteLine(ex.Message);
  }
}
```

It may not be easy to see the benefits of using SQL statement parameterization, but there are several good reasons to do so. First, it may help to write the term syntactically correct, and although it requires extra lines of code, it may pay off. In addition, parameters in principle give better performance, as you remove string operations, which are otherwise performed every time the SQL statement is executed, but the most important thing is that with parameters it increases security. The values to be inserted into a SQL statement typically come from user entries and can therefore contain anything and especially SQL code, which can be a security risk to the database. If parameters are used, the database system will check that the text does not contain illegal code. Therefore, it is considered good programming behavior to always parameterize SQL statements.

3.2 SQL INSERT

In this section I will show how to perform a SQL INSERT statement, and the class *DB* is expanded with a new method:

```
public void Insert(Person person)
{
  try
  {
    SqlCommand command = new SqlCommand("INSERT
    INTO Addresses (Phone, Firstname,
      Lastname, Address, Zipcode, EMail, Title)
      VALUES (@Phone, @Firstname,
      @Lastname, @Address, @Zipcode, @EMail, @Title)", connection);
    command.Parameters.Add(CreateParam("@Phone", person.Phone,
      SqlDbType.NVarChar));
    command.Parameters.Add(CreateParam("@Firstname", person.Firstname,
      SqlDbType.NVarChar));
    command.Parameters.Add(CreateParam("@Lastname", person.Lastname,
      SqlDbType.NVarChar));
    command.Parameters.Add(CreateParam("@Address", person.Address,
      SqlDbType.NVarChar));
    command.Parameters.Add(CreateParam("@Zipcode", person.Zipcode.Code,
      SqlDbType.NChar));
    command.Parameters.Add(CreateParam("@Email", person.Mail,
      SqlDbType.NVarChar));
    command.Parameters.Add(CreateParam("@Title", person.Title,
      SqlDbType.NVarChar));
    connection.Open();
    if (command.ExecuteNonQuery() == 1) return;
  }
  catch
  {
  }
  finally
  {
    if (connection.State == ConnectionState.Open) connection.Close();
  }
  throw new Exception("Person not be inserted in the database");
}
```

The SQL statement is again a parameterized statement and this time there are seven parameters, one for each column where to insert a value. It means that it is necessary to add seven *SqlParameter* objects to the command before it can be performed. Note that the command is performed with *ExecuteNonQuery()* which is the method to perform an INSERT, a UPDATE as well as a DELETE command. The method returns the number of rows which are modified and for an INSERT command it should be one.

You must also note that the method does not test where the foreign key *Zipcode* exists in the table *Zipcodes*. If not the operation fails and the method raises an exception.

The method is used in the following test method:

```
static void Test2()
{
  try
  {
    DB db = new DB();
    db.Insert(new Person { Phone = "12345678", Firstname = "Ragnar",
      Lastname = "Lodbrog", Address = "Voldgraven 2",
      Zipcode = new Zipcode { Code = "4320", City = "Lejre" },
      Mail = "", Title = "Viking" });
    Console.WriteLine(db.GetPerson("12345678"));
  }
  catch (Exception ex)
  {
    Console.WriteLine(ex.Message);
  }
}
```

The method try to create a new person in the database, and if it is possible the method read the person again and print the person on the screen.

3.3 SQL UPDATE

Executing a SQL UPDATE is the same as for a SQL INSERT, and the only difference is that the SQL expression is different:

```
public void Update(Person person)
{
  try
  {
    SqlCommand command = new SqlCommand("UPDATE Addresses
      SET Firstname = @Firstname, Lastname = @
      Lastname, Address = @Address,
      Zipcode = @Zipcode, Email = @Email, Title = @Title
      WHERE Phone = @Phone", connection);
    command.Parameters.Add(CreateParam("@Phone", person.Phone,
      SqlDbType.NVarChar));
    command.Parameters.Add(CreateParam("@Firstname", person.Firstname,
      SqlDbType.NVarChar));
    command.Parameters.Add(CreateParam("@Lastname", person.Lastname,
      SqlDbType.NVarChar));
    command.Parameters.Add(CreateParam("@Address", person.Address,
      SqlDbType.NVarChar));
    command.Parameters.Add(CreateParam("@Zipcode", person.Zipcode.Code,
      SqlDbType.NChar));
    command.Parameters.Add(CreateParam("@Email", person.Mail,
      SqlDbType.NVarChar));
    command.Parameters.Add(CreateParam("@Title", person.Title,
      SqlDbType.NVarChar));
    connection.Open();
    if (command.ExecuteNonQuery() == 1) return;
  }
  catch
  {
  }
  finally
  {
    if (connection.State == ConnectionState.Open) connection.Close();
  }
  throw new Exception("Person not be updated");
}
```

The following test method updates the row with key *12345678* and print the person after the database is updated:

```
static void Test3()
{
  try
  {
    DB db = new DB();
    db.Update(new Person { Phone = "12345678", Firstname = "Ragnar",
      Lastname = "Lodbrog", Address = "Ormegården 1",
      Zipcode = new Zipcode { Code = "4320", City = "Lejre" },
      Mail = "ragnar.lodbrog@mail.dk", Title = "Viking" });
    Console.WriteLine(db.GetPerson("12345678"));
  }
  catch (Exception ex)
  {
    Console.WriteLine(ex.Message);
  }
}
```

3.4 SQL DELETE

To delete a row in a database table the class *DB* is expanded with the following method:

```
public void Delete(string phone)
{
  try
  {
    SqlCommand command = new SqlCommand(
      "DELETE FROM Addresses WHERE Phone = @Phone", connection);
    command.Parameters.Add(CreateParam("@
    Phone", phone, SqlDbType.NVarChar));
    connection.Open();
    if (command.ExecuteNonQuery() == 1) return;
  }
  catch
  {
  }
  finally
  {
    if (connection.State == ConnectionState.Open) connection.Close();
  }
  throw new Exception("Person not be updated");
}
```

and the method can be tested as:

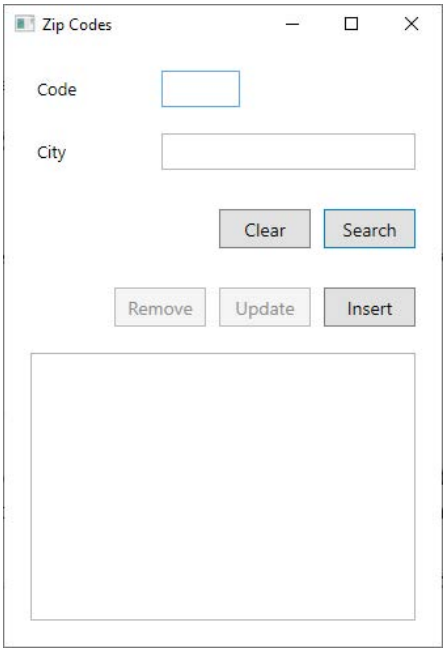
```
static void Test4()
{
  try
  {
    DB db = new DB();
    db.Delete("12345678");
    Console.WriteLine(db.GetPerson("12345678"));
  }
  catch (Exception ex)
  {
    Console.WriteLine(ex.Message);
  }
}
```

You must note that this time the method results in an exception as the person with phone number *12345678* is just deleted.

EXERCISE 3: ZIPCODES

Create a copy of the project *Zipcodes* shown in the start of this chapter. The class *DB* has one method used to search the database table. You should change the method such it uses a parameterized query.

Next you should expand the user interface with three new buttons:



1. If the user enter a code (4 digits) and a city name (must not be blank) and click *Insert* the program should insert a new row in the table *Zipcodes*.
2. If the user double click on a line in the list box the fields for code and city must be initialized with value for clicked zip code and the two buttons *Remove* and *Update* should be enabled and *Insert* disabled. It should then be possible to update the name of the city or delete the zip code.

3.5 PROVIDER INDEPENDENCE

The next example still uses the *Contacts* database, it is still a simple console application, and the program does nothing but execute a SQL SELECT statement, that is basically the same as in the first example, but the example should show how to make the code provider independent and is thus a program that can be connected to any contact database, whether the database is a SQL Server, MySQL or something else.

```
using System;
using System.Data;
using System.Data.Common;
using System.Configuration;

namespace ProviderProgram
{
  public class Program
  {
    public static void Main(string[] args)
    {
      DbConnection connection = null;
      DbCommand command;
      try
      {
        GetDbConnection(out connection, out command, "post");
        command.CommandText = "SELECT * FROM
        Zipcodes WHERE City LIKE 'Sk%'";
        connection.Open();
        DbDataReader reader = command.ExecuteReader();
        while (reader.Read()) Console.WriteLine("{0}
        {1}", reader[0], reader[1]);
        reader.Close();
      }
      catch
      {
      }
      finally
      {
        if (connection != null) connection.Close();
      }
    }

    private static void GetDbConnection(
      out DbConnection connection, out DbCommand command, string name)
    {
      ConnectionStringSettings settings =
        ConfigurationManager.ConnectionStrings[name];
      DbProviderFactory factory =
        DbProviderFactories.GetFactory(settings.ProviderName);
      connection = factory.CreateConnection();
```

```
      connection.ConnectionString = settings.ConnectionString;
      command = factory.CreateCommand();
      command.Connection = connection;
    }
  }
}
```

First, note that this time the namespace *System.Data.Common* was used and not one of the specific the database types are similar. The database types are similar to the following types

- *DbConnection*
- *DbCommand*
- *DbDataReader*

that are independent of the specific database. These classes are abstract base classes for the specific provider classes, and somewhere they must of course refer to specific objects, and this is done in the method *GetDbConnection()*. First, note that the program has a *config* file that contains a connection string to the *Contacts* database. The method *GetDbConnection()* must create a connection object and a command object for the specific database, and there are two parameters that are both *out* parameters. In addition, there is a third parameter that is the *name* of the connection string in the *config* file. The first thing that happens is that the definitions in the *config* file are read as an object of type *ConnectionStringSettings*. From here, in the form of a property *provider*, the name is used as parameter for a static method

```
DbProviderFactories.GetFactory(settings.ProviderName)
```

which returns a *DbProviderFactory* object determined from the provider name. This factory object can then create specific database objects for the current database. First the method creates a connection and note that the type of the variable is *DbConnection*, even though it is actually a *SqlConnection* object. For this connection object, its *ConnectionString* property is then set based on the string in the *config* file. Next, a *DbCommand* object is created for this connection.

In the *Main()* method there is not much new except that connection and command objects are created by the method *GetDbConnection()*, and all types are provider independent types. Otherwise, everything is a complete copy of the past. However, in connection with the Reader I have used numerical indices. It is really just to show that it is possible.

4 WPF AND DATABASES

Database applications resemble each other to some degree. Queries are performed on the basis of specified search criteria and the found data is displayed in a *DataGrid*. To maintain data, the program opens dialog boxes where data can be edited or they are entered directly in the *DataGrid*. In order to be able to develop such programs appropriately, some additions are required regarding WPF, and this chapter addresses three important concepts that have nothing to do with databases directly but are used to such an extent in database applications that there is a need to look at the theory a bit:

1. Data binding, which is the ability to associate a data object directly with an element in the user interface
2. *DataGrid*, which is a component like other components, but a very complex control
3. *Model-view-view-model*, which is a variant of the model-view-controller pattern that is adapted to the development of WPF applications

4.1 DATA BINDING

Data binding is a technique that associates an object (data) with an element in the user interface. A good example is where a component such as a *ListBox* is bound to a collection of objects. Hereby it can be obtained that if the data source (the content of the collection object) changes, then the change will immediately be reflected in the user interface component, and similarly if a bound object in the user interface is modified, then the data source is automatically updated. In reality, it is all a matter of inserting objects between components in the user interface and the data source, and these binding objects will then take over much of the work that would otherwise be necessary to write code yourselves.

Data binding consists of creating a dependency between the value of a property on a component in the user interface (this property is called the target property) and the value of another property (called the source property), which is merely a property of some object. The result is that the target property automatically gets the value of the source property, but such that a change of the source is automatically displayed in the user interface. The binding can be one-way, where the target property only shows the value of the source and a change of target does not update the data source, but the binding can also be two-way, so a change of the target automatically updates the source. This distinction is necessary as one is not always interested in automatically updating the data source.

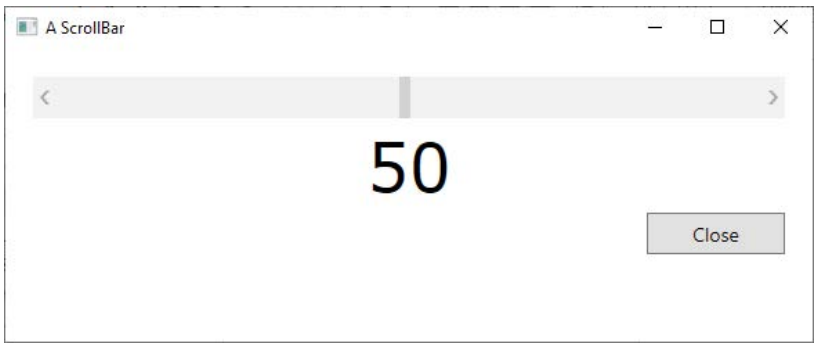
What that holds it all together is the class *Binding*, which represents the connection between target and source properties. The main features of the class are:

- *ElementName*, read-write property for the source object - the object to bind to (with a single exception)
- *FallbackValue*, read-write property, which specifies a value to use if the binding fails for some reason
- *Mode*, read-write property that defines the binding type
- *NotifyOnSourceUpdated*, read-write property that specifies whether to raise a *SourceUpdated* event when passing a value from target to source
- *NotifyOnTargetUpdated*, read-write property that specifies whether to raise a *TargetUpdated* event when transferring a value from source to target
- *Path*, read-write property to source property at the binding source object
- *Source*, read-write property for the binding source object (used only if the object is not a WPF element)
- *TargetNullValue*, read-write property for the value used if the source is *null*

4.2 BINDING TWO WPF CONTROLS

The simplest form for data binding is to bind a property on one WPF component to a property on another WPF component. It may not be the most common form of data binding, but it also has its uses.

The program is called *AScrollBar* and if you run the program, you get the window



which contains a *ScrollBar*, a *Label* and a *Button*. The scroll bar scrolls over the range from 1 to 100, and if you move the scroll bar, the label component is updated with the value of the scroll bar. The XML is quite simple:

```
<StackPanel Margin="20">
  <ScrollBar Name="sb" Orientation="Horizontal" Height="30" Minimum="1"
    Maximum="100" Value="50" SmallChange="1" LargeChange="10"/>
  <Label Name="label" FontFamily="Verdana" FontSize="48"
    HorizontalContentAlignment="Center"></Label>
  <Button Width="100" Content="Close" HorizontalAlignment="Right"
FontSize="14" Height="30" Click="Button_Click"/>
</StackPanel>
```

and the only thing to note is the names of the components. The code behind is:

```
public partial class MainWindow : Window
{
  public MainWindow()
  {
    InitializeComponent();
    BindComponents();
  }

  private void BindComponents ()
  {
    Binding bind = new Binding();
    bind.Converter = new ScrollConverter();
    bind.Source = sb;
    bind.Path = new PropertyPath("Value");
    label.SetBinding(Label.ContentProperty, bind);
  }

  private void Button_Click(object sender, RoutedEventArgs e)
  {
    Close();
  }
}
```

The method *BindComponents()* bind the two components using a *Binding* object. The object is assigned a data source which is the scrollbar and a target assigned to property *Path* as a *PropertyPath* object that tells which property on the source object (here the scrollbar) the target should bind to. As the last the binding is performed by the method *SetBinding()* where the first argument is target property and the other is the *Binding* object. There is one more statement where the *Binding* object is assigned a converter. The value of a scrollbar is a *double* and I want the program to show the value as an *int*. To do that the *bind* object get at converter which is an object of the type:

```
class ScrollConverter : IValueConverter
{
  public object Convert(object value, Type
  targetType, object parameter,
    CultureInfo culture)
  {
    return (int)(double)value;
  }

  public object ConvertBack(object value, Type
  targetType, object parameter,
    CultureInfo culture)
  {
    throw new NotImplementedException();
  }
}
```

When the last method is not implemented it is because then binding is one way as a *Label* has no user interaction.

The project *AScrollBar1* is a program that works in exactly the same way, but the binding is defined in XML:

```
<Window.Resources>
  <local:ScrollConverter x:Key="convert"/>
</Window.Resources>
<StackPanel Margin="20">
  <ScrollBar Name="sb" Orientation="Horizontal" Height="30" Minimum="1"
    Maximum="100" Value="50" SmallChange="1" LargeChange="10"/>
  <Label Name="label" FontFamily="Verdana" FontSize="48"
    HorizontalContentAlignment="Center"
    Content="{Binding ElementName=sb, Path=Value,
    Converter={StaticResource convert}}"/>
  <Button Width="100" Content="Close"
  HorizontalAlignment="Right" FontSize="14"
    Height="30" Click="Button_Click"/>
</StackPanel>
```

Note that the converter is defined as a resource which means that *convert* is an object of the type *ScrollConverter*. When you look at the definition of the *Label* element there is defined three attributes for binding of the components *Content* property: The source as *ElementName*, the *Path* as the property on the source and the converter. Then the binding logic in C# can be removed, but the class *ScrollConverter* still needs to be there.

If the data source had been anything other than a WPF component, the source is referenced differently, but this is shown in the next example.

EXERCISE 4: BIND THE BACKGROUND

You must write a program called *BackgroundProgram* which opens the following window:

The window has a check box, and when the user selects the check box, the windows background must change to red, and if the check box again is deselected, the background must again be changed to white. The problem must be solved using data binding where the check box's *IsChecked* property should be bound the *Window* elements *Background* property using a converter. You can do that in the following way:

```
<Window.Resources>
  <local:CheckConverter x:Key="convert"/>
</Window.Resources>
<Window.Background>
  <Binding ElementName="chkColor" Path="IsChecked"
    Converter="{StaticResource convert}">
  </Binding>
</Window.Background>
```

4.3 BINDING TO AN OBJECT

If you bind to an object that is not a WPF component, you use the *Source* property of the component to specify the source object. The object can be any object, and the binding can basically be done in two ways:

- a property on the WPF component (the target object) is bound to a property by a static object

```
<Button Background="{Binding Source={x:Static
SystemColors.WindowColor}}" ...
```

- a property on the WPF component (the target object) is bound to a property by a logical resource

```
<Window.Resources>
  <local:aClass x:Key="theObject" />
</Window.Resources>
<Grid>
  <Button
    Content="{Binding Source={StaticResource
    theObject},Path=myProperty}" />
```

You can also bind to data sources other than the *Source* property, and you can use a *DataContext* property in the component's visual tree instead. If a binding object's *Source*, *RelativeSource*, or *Element* property is not set, the system will search upward in the visual tree until it arrives at an element where the property *DataContext* is not *null*. It will then be a source object for all bindings in the visual tree:

```
<Window.Resources>
  <local:DataObject x:Key="dataObject" />
</Window.Resources>
<Grid DataContext="{StaticResource dataObject}">
  <Label Content="{Binding Path=Title}" />
```

The property *Binding.Mode* specifies how the component should behave with respect to changes in either the source or the target value. There are the following options:

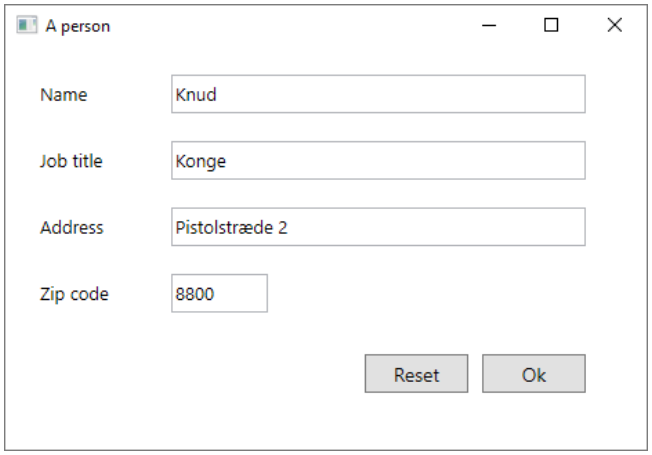
1. *Default*, which means that the binding must use default mode on the target property
2. *OneTime*, which means that the binding must update the target when the application starts, but the target does not need to be updated if the source changes value

3. *OneWay*, which means that the binding must update the target when the source changes value, but changes in the target do not update the source
4. *TwoWay*, which means that changes in the source must update the target, and changes in the target must update the source

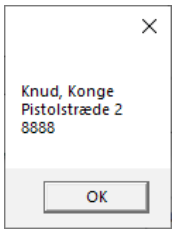
Here, *OneWay* and *TwoWay* are most commonly used, where the first is used if the program only has to display data, while the second is used if it is necessary to edit data.

Sometimes you bind an element to a data value that can be *null*. You can then assign a default value to *TargetNullValue*.

As an example the program *BindToObject* opens the following window:



The window shows information about a person, and if you click on the *OK* button you get a message box which shows the information entered in text fields:



The fields for *Name* and *Job title* are bound to properties on an object of the type *Person* while the fields *Address* and *Zip code* are bound to properties on an object of the type *Address*. The message box shows the state of these two objects and if you change the values in the *TextBox* fields you should note that the objects states are automatic updated. If the user clicks the *Reset* button all fields are changed as shown above and happens when the objects state is changed and then is a two-way binding.

The class *Person* represents a person as a name and a job title:

```
public class Person : INotifyPropertyChanged
{
  public event PropertyChangedEventHandler PropertyChanged;
  private string name;
  private string job;

  public Person(string name, string job)
  {
    Name = name;
    Job = job;
  }

  public string Name
  {
    get { return name; }
    set
    {
      name = value;
      OnPropertyChanged("Name");
    }
  }

  public string Job
  {
    get { return job; }
    set
    {
      job = value;
      OnPropertyChanged("Job");
    }
  }

  protected void OnPropertyChanged(string name)
  {
    if (PropertyChanged != null)
      PropertyChanged(this, new PropertyChangedEventArgs(name));
  }

  public override string ToString()
  {
    return name + ", " + job;
  }
}
```

Since it must be possible to bind the class to a WPF component, it must implement the *INotifyPropertyChanged* interface, which defines a method to be called when a property's state is changed. This way, the user interface is notified that the value has changed. The class has an event where a component bound to an object of the type *Person* can indicate that it wants to receive notifications about changes in the *Person* object's state. It is the method *OnPropertyChanged()* that sends notifications.

Otherwise, there is not much to explain, but note that the class's set properties send notifications of changes by calling *OnPropertyChanged()*.

The class *Address* is basically completely identical to the class *Person* and has two properties:

```
public class Address : INotifyPropertyChanged
{
  public event PropertyChangedEventHandler PropertyChanged;
  private string line;
  private string post;
```

You should thus note that in order to bind an object to a component of the user interface, the object's class must implement the interface *INotifyPropertyChanged* and the *Set* methods for all properties where it should be able to bind to must call the method *OnPropertyChanged()*. That's all that's needed on the data source.

Then there is the user interface:

```
<Window.Resources>
  <local:Address x:Key="adr" Line="Pistolstræde 2" Post="8888"/>
</Window.Resources>
<Grid Margin="20">
  <Grid.RowDefinitions>
    <RowDefinition Height="48" />
    <RowDefinition Height="48" />
    <RowDefinition Height="48" />
    <RowDefinition Height="48" />
    <RowDefinition Height="48" />
    <RowDefinition />
  </Grid.RowDefinitions>
```

```
  <Grid.ColumnDefinitions>
    <ColumnDefinition Width="100"/>
    <ColumnDefinition Width="300"/>
    <ColumnDefinition/>
  </Grid.ColumnDefinitions>
  <Label Content="Name" FontSize="13" Margin="0,0,0,20" />
  <Label Grid.Row="1" Content="Job title"
  FontSize="13" Margin="0,0,0,20" />
  <Label Grid.Row="2" Content="Address"
  FontSize="13" Margin="0,0,0,20" />
  <Label Grid.Row="3" Content="Zip code"
  FontSize="13" Margin="0,0,0,20" />
  <TextBox Grid.Row="0" Grid.Column="1" Margin="0,0,0,20" x:Name="txtName"
    FontSize="13" VerticalContentAlignment="Center" />
  <TextBox Grid.Row="1" Grid.Column="1"
  Margin="0,0,0,20" x:Name="txtJob"
    FontSize="13" VerticalContentAlignment="Center" />
  <TextBox Grid.Row="2" Grid.Column="1"
  Margin="0,0,0,20" x:Name="txtAddr"
    Text="{Binding Source={StaticResource
    adr}, Path=Line}" FontSize="13"
    VerticalContentAlignment="Center" />
  <TextBox Grid.Row="3" Grid.Column="1"
  Margin="0,0,0,20" x:Name="txtPost"
    Text="{Binding Source={StaticResource adr}, Path=Post}" FontSize="13"
    VerticalContentAlignment="Center" Width="70"
    HorizontalAlignment="Left" />
  <StackPanel Grid.Row="4" Grid.ColumnSpan="2"
  HorizontalAlignment="Right"
    Orientation="Horizontal">
    <Button Content="Reset" Margin="0,10,0,10"
    x:Name="cmdReset" Width="75"
      FontSize="14" Click="cmdReset_Click" />
    <Button Content="Ok" Margin="10,10,0,10" x:Name="cmdOk" Width="75"
      FontSize="14" Click="cmdOk_Click" />
  </StackPanel>
</Grid>
```

First, a local resource is defined for an *Address* object. In particular, note how the object is initialized and how it is identified by the name *adr*. For the window's components it is just the last two *TextBox* components that you should note. They bind to a property at the local resource, and you should especially notice the syntax where the property *Text* is the target property and is bound to a property on the local resource.

As the next is the code behind for the window:

```
public partial class MainWindow : Window
{
  private Person pers = new Person("Knud", "Konge");

  public MainWindow()
  {
    InitializeComponent();
    Databind(txtName, "Name");
    Databind(txtJob, "Job");
  }

  private void Databind(TextBox ctrl, string name)
  {
    Binding bind = new Binding(name);
    bind.Source = pers;
    ctrl.SetBinding(TextBox.TextProperty, bind);
  }

  private void cmdOk_Click(object sender, RoutedEventArgs e)
  {
    MessageBox.Show(pers.ToString() + "\n" + Resources["adr"]);
  }

  private void cmdReset_Click(object sender, RoutedEventArgs e)
  {
    pers.Name = "Knud";
    pers.Job = "Konge";
    ((Address)Resources["adr"]).Line = "Pistolstræde 2";
    ((Address)Resources["adr"]).Post = "8800";
  }
}
```

First, an object of type *Person* is created. The properties of this object are bound in the constructor using the method *Databind()* to the two upper *TextBox* components in the window. Here, too, note the syntax where the bound object's *Source* property is set to the object to be bound and how to specify which target property to bind.

Then there is the event handlers for the buttons and here you should especially notice how in the code you refer to the local resource. The goal of the *Reset* button is to show when the two objects (*pers* and the local resource) are updated then the values in the input fields are also updated and then to show that the UI automatically displays changes to the data source.

4.4 A LITTLE MORE ABOUT CONVERSION

In the first example of data binding, I showed how to use a converter, which is a class that implements the interface *IValueConverter*. Data conversion generally deals with how data from a data source is displayed in the presentation layer. The interface *IValueConverter* defines two methods where one - *Convert()* - converts source data before they are displayed (linked to target), while the other - *ConvertBack()* - converts the other way, thus converting data in the target object before updating the source object. If it alone is an *OneWay* binding, it is unnecessary to implement *ConvertBack()*.

In practice, you write a converter class that implements *IValueConverter* you has to (should) decorate the class with a *ValueConversion* attribute:

```
[ValueConversion(typeof(double), typeof(string))]
class DoubleConverter : IValueConverter
{
```

Here the attribute tells that the value is converted from a *double* to a *string*. You can also use converters to return objects. For example, an image stored in a database or XML document could be stored as a reference to the file name, and the converter could then load the image and return it as a *BitmapImage* object. Converters can thus be used for many things other than simple conversion between two types. In this context, conversions related to the current culture can also be mentioned, and the two conversion methods also have a reference to a *Culture* object. For example you may need to determine which character is used for the decimal point. An example of a converter could be the following which converts between *string* and *double*:

```
[ValueConversion(typeof(double), typeof(string))]
class DoubleConverter : IValueConverter
{
  public object Convert(object value, Type
  targetType, object parameter,
    System.Globalization.CultureInfo culture)
  {
    if (value == null || ("" + value).Length == 0) return "";
    double number = (double)value;
    return Math.Abs(number) < 0.001 ? "" : string.Format("{0:F2}", number);
  }
```

```
  public object ConvertBack(object value, Type
  targetType, object parameter,
    System.Globalization.CultureInfo culture)
  {
    try
    {
      return double.Parse(value.ToString());
    }
    catch
    {
      return 0;
    }
  }
}
```

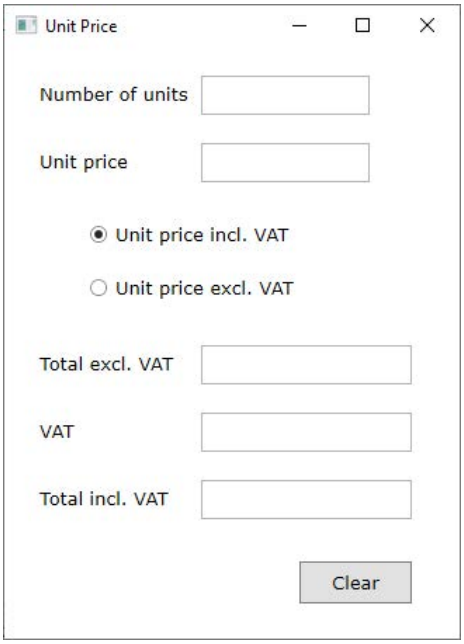
There is an alternative to *IValueConverter*, called *IMultiValueConverter*, which similarly defines two conversion methods. The difference is that this converter can be bound to multiple target objects, thus allowing the conversion to depend on multiple values. In order to bind a property to a value from a multi value converter, you have to add a local resource for a converter object, just like a conventional converter, but the binding is done with a class called *MultiBinding*, which is really a collection of bindings.

EXERCISE 5: CALCULATE A PRICE

You must write a program which opens the following window, a program you have seen before (in the book C# 2) where the user should enter the number of units for an item as well as the unit price. The unit price can be entered with and without VAT, and the program must then calculate the total price without VAT, the VAT and the total price inclusive VAT.

This time the calculation must happens automatic when the user has entered a number of units and a unit price when all *TextBox* fields and the two *RadioButtons* must be bound to properties in a model class. For the three bottom fields the binding should be one way as fields should be read-only *TextBox* components. For a *RadioButton* you must bind the property *IsChecked*.

When the user enter text it is necessary to associate converters to all *TextBox* fields and you can use the converter shown above, but you must write your own converter for the number of units.



As a model you can use the following class:

```
public class Item : INotifyPropertyChanged
{
  public event PropertyChangedEventHandler PropertyChanged;
  private static readonly double VAT = 0.25;
  private int number;
  private double price;
  private bool isIncl = true;

  public int Number
  {
    get { return number; }
    set
    {
      number = value;
      OnPropertyChanged("Number");
      Notify();
    }
  }

  public double Price { ... }
  public bool IsExcl { ... }
  public bool IsIncl { ... }

  public double Excl
  {
    get { return GetPrice() * number; }
  }
```

```
  public double Vat { ... }
  public double Incl { ... }

  protected void OnPropertyChanged(string name)
  {
    if (PropertyChanged != null)
      PropertyChanged(this, new PropertyChangedEventArgs(name));
  }

  public void Notify()
  {
    OnPropertyChanged("Excl");
    OnPropertyChanged("Vat");
    OnPropertyChanged("Incl");
  }

  public void Clear()
  {
    Number = 0;
    Price = 0;
    Notify();
  }

  private double GetPrice()
  {
    return isIncl ? price / (1 + VAT) : price;
  }
}
```

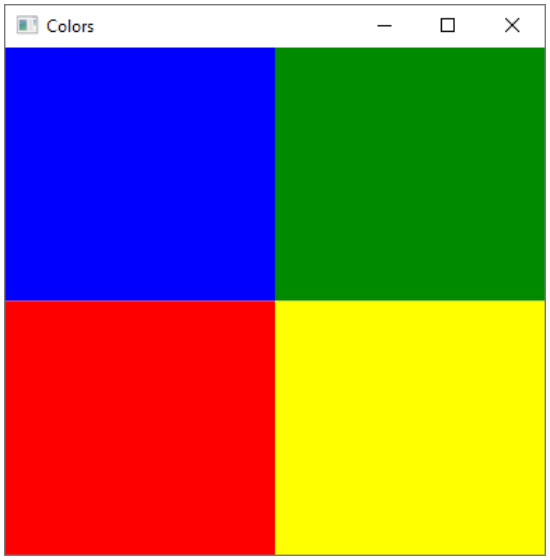
It is a requirement that all data bindings must be done in XML. Note that this means that you have to define resources for an *Item* object and the necessary converters.

EXERCISE 6: CALCULATE A PRICE AGAIN

Make a copy of the project from exercise 5. You must then change the program so all bindings are performed in code behind. This means that you must delete all resources defined in XML as well as all binding attributes. The bindings must then be performed from the constructor. Other than that, the program should work the same way as before.

EXERCISE 7: BIND COLORS

You must write a program that opens the following window:



The layout should be a simple *UniformGrid* with four *Rectangle* components. The program should as resources define four *SolidColorBrush* objects, and these objects must be bound to a *Rectangle*'s *Fill* property.

The purpose of the exercise is to show that all properties can be bound.

4.5 BINDING TO A COLLECTION

This book is about database programming and regarding data binding, the most important thing is to bind a data source consisting of a collection of objects to a component in the user interface. The above examples show data binding where a target property is bound to a source property and it certainly has many uses, and it is worth mentioning that as the last exercise shows, virtually all properties of a component can be bound to a data source. However, data binding is particularly interesting if you bind a component to a collection, and here are two cases

1. a single element in a collection is bound to a component with the ability to navigate the elements
2. the entire contents of that collection are bound to an item control such as a *ListBox*

The simplest form of data binding to an item control is a *ListBox*, which displays the contents of a list. Below is a type representing a king with a name and the reign:

```
public class King
{
  public int From { get; private set; }
  public int To { get; private set; }
  public string Name { get; private set; }

  public King(string name, int from, int to)
  {
    Name = name;
    From = from;
    To = to > 0 ? to : 9999;
  }

  public override string ToString()
  {
    return string.Format("{0, -40} {1, 4} - {2, 4}",
        Name, From > 0 ? "" + From : "", To < 9999 ? "" + To : "");
  }
}
```

You should mainly note that the three properties are readonly. Below is a list of the Danish kings where the data is hard coded in the class:

```
public class Kings : IEnumerable<King>
{
  private List<King> list;

  public Kings()
  {
    CreateList();
  }

  public int Count
  {
    get { return list.Count; }
  }

  public King this[int n]
  {
    get { return list[n]; }
  }
```

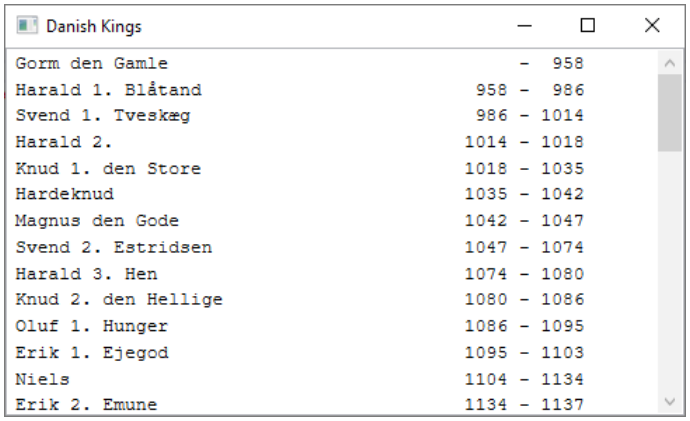
```
  public IEnumerator<King> GetEnumerator()
  {
    return list.GetEnumerator();
  }

  System.Collections.IEnumerator System.
  Collections.IEnumerable.GetEnumerator()
  {
    return GetEnumerator();
  }

  private void CreateList() { ... }
}
```

The program consists of a single *ListBox*:

```
<Grid>
  <ListBox Name="lstKings" FontFamily="Courier new" FontSuze=13 />
</Grid>
```

and the code binds this *ListBox* to an object of the type *Kings*:

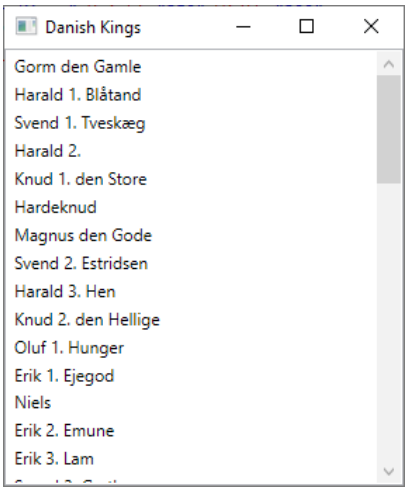
```
public partial class MainWindow : Window
{
  public MainWindow()
  {
    InitializeComponent();
    lstKings.ItemsSource = new Kings();
  }
}
```



In particular, note that data binding to an items control is done with *ItemsSource* as the target property. The source must be a collection that implements the interface *IEnumerable*, what all .NET collections and including an array do. Thus, it is simple to bind an item control to for example a *List* or an array. The class *Kings* is not a list, but a class that encapsulates a list, and as the class implements the interface *IEnumerable*, it can immediately be bound to a *ListBox*.

The list box displays a line for each *King* object in the class *Kings*, and what appears is the result of the objects *ToString()*. To make the result look reasonable, I have chosen to give the list box a none-proportional font. You should note that it is extremely simple to view the contents of a list in a *ListBox*.

The project for the above program is called *TheKings*. There is also a project called *TheKings1* which opens the following window:



This example is essentially the same as the above, and the class *Kings* is completely unchanged but the XML code has been slightly modified, so that this time the list box should only show the name:

```
<Grid Name="mainGrid">
  <ListBox ItemsSource="{Binding}" DisplayMemberPath="Name" />
</Grid>
```

Here you should note that the *ItemsSource* of the list box is bound, but no object is mentioned. Therefore, the visual tree is searched upwards and the data source is this time bound to the *Grid* component:

```
public partial class MainWindow : Window
{
  public MainWindow()
  {
    InitializeComponent();
    mainGrid.DataContext = new Kings();
  }
}
```

A *ListBox* is a very frequently used item control, and it is among other things suitable for data binding to display the contents of a collection, but there are other options including a *ListView*. A *ListBox* displays objects with one object on each line, but a list box cannot immediately display multiple columns, which is exactly what a *ListView* can do. It should be added that a better solution than a *ListView* is often a *DataGrid*, but I will look at that later. The project *TheKings2* is in princip the same project as *TheKings* but shows the objects in a *ListView*:

The program displays the same data source as before, but the data are arranged nicely in resizable columns. The component that displays data is a *ListView*. The user interface is written as:

```
<Window.Resources>
  <local:YearConverter x:Key="converter" />
</Window.Resources>
<Grid>
  <ListView Name="lstKings">
    <ListView.View>
      <GridView>
        <GridViewColumn Width="200" Header="Name"
          DisplayMemberBinding="{Binding Name}" />
        <GridViewColumn Width="60" Header="From" DisplayMemberBinding=
          "{Binding From, Converter={StaticResource converter}}"/>
        <GridViewColumn Width="60" Header="To" DisplayMemberBinding=
          "{Binding To, Converter={StaticResource converter}}" />
      </GridView>
    </ListView.View>
  </ListView>
</Grid>
```

The view has a name so it can be linked to a data source in the code. A *ListView* has a *View* that can view and scroll an item. In this case, it is a *GridView*, which is a component that can display elements organized in rows and columns. In this case, three columns are defined and each column is bound to a property for a *King* object. In particular, despite the name, a *GridView* has nothing to do with a *Grid*. Also note that the last two columns have a converter defined as a local resource, and the class is written in the program's code behind.

4.6 BINDING A COLLECTION TO A SINGLE PROPERTY

It is possible to bind a single property in a collection such as a list, and the current target property will then display the first element of the data source. However, you can navigate through the list, showing the individual elements one at a time. When a collection is encapsulated in an *ICollectionView* object before it is bound, several properties and methods are defined (by the interface *ICollectionView*) to navigate the object that the component displays:

- *CurrentItem*, which refers to the current item - the item being displayed
- *CurrentPosition*, which is the index of the current item
- *IsCurrentAfterLast*, which is a property that indicates that the current element refers to an object after the last element
- *IsCurrentBeforeFirst*, a property that indicates that the current element refers to an object before the first element
- *MoveCurrentTo()*, which is a method that sets the current element to a specific object
- *ModeCurrentToFirst()*, which sets the current element to the first element
- *MoveCurrentToLast()*, which sets the current element to the last element
- *MoveCurrentToNext()*, which sets the current element to the next element
- *MoveCurrentToPosition()*, which sets the current element to the object with a specific index
- *MoveCurrentToPrevious()*, which sets the current element to the previous element

The next example is also a program called *TheKings3* which binds to the collection *Kings*, but this time a single *King* is bound to the window:

Using the buttons you can navigate through the kings and show one at a time. The XML for *MainWindow* is:

```
<Window.Resources>
  <local:YearConverter x:Key="converter" />
</Window.Resources>
<Grid Name="mainGrid" Margin="20">
  <Grid.RowDefinitions>
    <RowDefinition Height="50"/>
    <RowDefinition Height="50"/>
    <RowDefinition Height="40"/>
    <RowDefinition />
  </Grid.RowDefinitions>
  <Label HorizontalAlignment="Left" Margin="0,0,0,20"
Content="Danish kings" FontWeight="Bold" FontSize="14" />
  <Grid Grid.Row="1" Margin="0,0,0,20">
    <Grid.ColumnDefinitions>
      <ColumnDefinition Width="80"/>
      <ColumnDefinition Width="80"/>
      <ColumnDefinition />
    </Grid.ColumnDefinitions>
    <Label Content="{Binding Path=From,
    Converter={StaticResource converter}}"
      FontWeight="Bold" />
    <Label Grid.Column="1" FontWeight="Bold"
      Content="{Binding Path=To, Converter={StaticResource converter}}" />
    <Label Grid.Column="2" FontWeight="Bold" Foreground="Blue"
      Content="{Binding Path=Name}" HorizontalAlignment="Left" />
  </Grid>
  <UniformGrid Grid.Row="2" Columns="4">
    <Button Margin="0,10,0,0" Content="First"
    FontSize="14" Name="cmdFirst"
      Click="cmdFirst_Click" />
    <Button Margin="10,10,10,0" Content="Previous"
    FontSize="14" Name="cmdPrev"
      Click="cmdPrev_Click" />
    <Button Margin="0,10,10,0" Content="Next" FontSize="14" Name="cmdNext"
      Click="cmdNext_Click" />
    <Button Margin="0,10,0,0" Content="Last" FontSize="14" Name="cmdLast"
      Click="cmdLast_Click" />
  </UniformGrid>
</Grid>
```

You should note how the three *Label* components are bound and especially here, that only the property of the data source to be bound is defined. However, the data source itself is not defined and it is bound in the code to the *Grid* container, which is in the visual tree for the three components. Also note that for the first two components, a converter is defined in the same way as in the previous example.

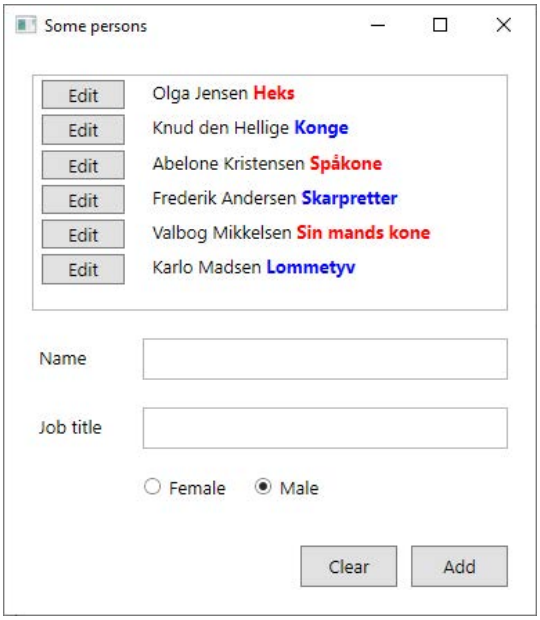
The window also has buttons to navigate the data source and thus what data to display in the window. The code is as follows:

```
public partial class MainWindow : Window
{
  private ICollectionView view = CollectionViewSource.
  GetDefaultView(new Kings());

  public MainWindow()
  {
    InitializeComponent();
    mainGrid.DataContext = view;
  }

  private void cmdFirst_Click(object sender, RoutedEventArgs e)
  {
    view.MoveCurrentToFirst();
  }

  private void cmdPrev_Click(object sender, RoutedEventArgs e)
  {
    view.MoveCurrentToPrevious();
  }

  private void cmdNext_Click(object sender, RoutedEventArgs e)
  {
    view.MoveCurrentToNext();
  }

  private void cmdLast_Click(object sender, RoutedEventArgs e)
  {
    view.MoveCurrentToLast();
  }
}
```

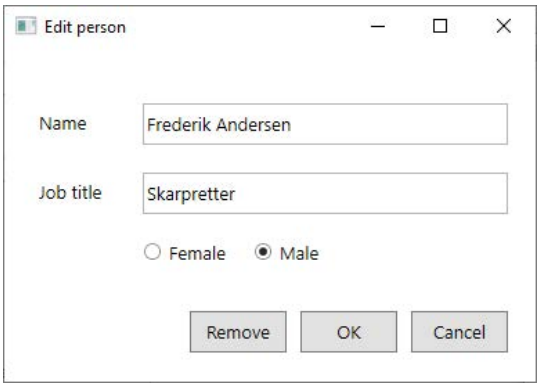
Here you should especially notice how the data source in the form of a *Kings* object is encapsulated in an *ICollectionView* and how the event handlers calls methods defined by this view.

4.7 DATA TEMPLATES

Above I have shown how to bind data to a component so that the component displays data as text, but with a data template you can define a more advanced presentation. A data template is just some XML that describes how data should be displayed. In this example I will show how to use a data template, a subject that in reality can be said much more about. The project is called *SomePeople* and opens the following window:



At the bottom are two *TextBox* components to name and job title for a person as well as two radio buttons to select the gender. The upper component is a *ListBox* which shows the content of a list with *Person* objects, but the *ListBox* has a data template which define how each person should be shown: As a button, a text and a bold text that is blue (a male) or red (a female). It means that using a data template you can control how a content of a control should be shown. If you click on a button in the *ListBox* the program opens a window, where you can edit the person:



A person is defined as:

```
using System.ComponentModel;

namespace SomePeople
{
  public class Person : INotifyPropertyChanged
  {
    public event PropertyChangedEventHandler PropertyChanged;
    private string name;
    private string job;
    private bool male;

    public Person(string name, string job, bool male)
    {
      Name = name;
      Job = job;
      IsMale = male;
    }

    public Person Current
    {
      get { return this; }
    }

    public bool IsMale
    {
      get { return male; }
      set
      {
        male = value;
        OnPropertyChanged("IsMale");
      }
    }

    public string Name { ... }
    public string Job { ... }

    protected void OnPropertyChanged(string name)
    {
      if (PropertyChanged != null) PropertyChanged(this,
        new PropertyChangedEventArgs(name));
    }

    public override bool Equals(object obj) { ... }
  }
}
```

There is not much new, but you should note that the class is defined so that its properties can be bound to the UI and that the class overrides *Equals()*. In particular, you should note that there is a read-only property *Current* that returns an instance of the current object.

The program also has a class *PersonList* that inherits *List<Person>*, and the sole purpose is to initialize the list with a few persons when the program starts.

The most important happens in XML:

```
<Window.Resources>
  <local:ColorConverter x:Key="converter" />
</Window.Resources>
<Grid Margin="20">
  <Grid.Resources>
    <DataTemplate x:Key="PersonTemplate">
      <Grid>
        <Grid.ColumnDefinitions>
          <ColumnDefinition Width="80"/>
          <ColumnDefinition/>
        </Grid.ColumnDefinitions>
        <Button Content="Edit" Tag="{Binding
        Path=Current}" Margin="0,0,20,0"
          Click="cmdEdit_Click" />
        <WrapPanel Grid.Column="1">
          <TextBlock Text="{Binding Path=Name}"/>
          <TextBlock Text=" "/>
          <TextBlock FontWeight="Bold" Text="{Binding Path=Job}"
  Foreground="{Binding Path=IsMale,
  Converter={StaticResource converter}}"/>
        </WrapPanel>
      </Grid>
    </DataTemplate>
  </Grid.Resources>
  <Grid.RowDefinitions>
    <RowDefinition/>
    <RowDefinition Height="50"/>
    <RowDefinition Height="50"/>
    <RowDefinition Height="50"/>
    <RowDefinition Height="50"/>
  </Grid.RowDefinitions>
  <Grid.ColumnDefinitions>
    <ColumnDefinition Width="80"/>
    <ColumnDefinition/>
  </Grid.ColumnDefinitions>
```

```
  <ListBox Grid.ColumnSpan="2" Name="lstNames" ItemsSource="{Binding}"
    ItemTemplate="{StaticResource PersonTemplate}"/>
  <Label Grid.Row="1" Content="Name" Margin="0,20,0,0" />
  <Label Grid.Row="2" Content="Job title" Margin="0,20,0,0" />
  <TextBox Grid.Row="1" Grid.Column="1" Margin="0,20,0,0" Name="txtName"
    VerticalContentAlignment="Center"/>
  <TextBox Grid.Row="2" Grid.Column="1" Margin="0,20,0,0" Name="txtTitle"
    VerticalContentAlignment="Center" />
  <StackPanel Grid.Row="3" Grid.Column="1" Orientation="Horizontal">
    <RadioButton Content="Female" Margin="0,20,10,0" Name="cmdFemale" />
    <RadioButton Content="Male" Margin="10,20,0,0" Name="cmdMale"
      IsChecked="True" />
  </StackPanel>
  <StackPanel Grid.Row="4" Grid.ColumnSpan="2" Orientation="Horizontal"
    HorizontalAlignment="Right">
    <Button Content="Clear" Margin="0,20,10,0" Name="cmdClose" Width="70"
      Click="cmdClear_Click" />
    <Button Content="Add" Margin="0,20,0,0" Name="cmdAdd" Width="70"
      Click="cmdAdd_Click" />
  </StackPanel>
</Grid>
```

First is defined a converter as a window resource, and the class *ColorConverter* is defined in code behind and it is a converter that converts a *bool* to a *SolidColorBrush*. The rest of the XML is a *Grid* that starts with defining a data template as *Grid.Resource*. The data template is itself a *Grid* with two columns. The first column is used for a *Button* while the other column contains a *WrapPanel* with three *TextBlock* components, the first for the name and the last for the job title. The three components button and the first and the last *TextBlock* are bound to a property for a *Person* object. The buttons *Tag* property is bound to the property *Current* in the *Person* object and the to the *Person* object shown in this line in the *ListBox*. Also note that all buttons in the *ListBox* are assigned the same event handler. The first *TextBlock* component is bound to the *name* property while the last is bound to the *Job* property. Here you must note the last that use the converter. The source property has the type *bool* and this value converts the converter to a *SolidColorBrush* bound to the *Foreground* property.

For the rest of the XML there is nothing new, but you must note the *ListBox* element, and here that the *ItemsSource* property is bound, not to concrete data source and the runtime system will then search upward in the visual tree to find a binding.

As the next the code behind:

```
public partial class MainWindow : Window
{
  private ObservableCollection<Person> list =
    new ObservableCollection<Person>(new PersonList());

  public MainWindow()
  {
    InitializeComponent();
    lstNames.ItemsSource = list;
  }

  private void cmdAdd_Click(object sender, RoutedEventArgs e)
  {
    string name = txtName.Text.Trim();
    string title = txtTitle.Text.Trim();
    if (name.Length > 0 && title.Length > 0)
    {
      list.Add(new Person(name, title, cmdMale.IsChecked == true));
      txtName.Clear();
      txtTitle.Clear();
    }
  }

  private void cmdClear_Click(object sender, RoutedEventArgs e)
  {
    list.Clear();
  }

  private void cmdEdit_Click(object sender, RoutedEventArgs e)
  {
    EditWindow dlg = new EditWindow((Person)(((Button)sender).Tag));
    dlg.RemovePerson += RemoveHandler;
    dlg.ShowDialog();
  }

  public void RemoveHandler(object sender, PersonEventArgs e)
  {
    list.Remove(e.Person);
  }
}
```

The most important is the encapsulation of the list *PersonList()* in an *ObservableCollection* and in the constructor this collection is bound to the *ListBox*. It is a wrapper for a collection, in this case a *List<Person>*, and is a collection that raises events when the content is changed, events that the target component in the user interface is notified. The result is that the *ListBox* automatic is updated when the collection *List* is modified. You should note that this means that the event handlers all are simple.

The last event handler is for the buttons in the *ListBox* and opens a dialog box to edit the selected person. You must note how the *Tag* property reference the *Person* object which is send to the constructor in the class *EditWindow*. I will not show the code for this class here as it does not contain anything new, but you must observe that *ListBox* due to the data binding is automatically updated. Also note that the dialog box fires an event, when the user click on the *Remove* button and the event handler *RemoveHandler()* then remove the object from the list, and when it is a *ObservableCollection* the user interface is updated.

4.8 DATAGRID

A *DataGrid* is a component which shows data in rows and columns. It is a complex component and perhaps even the most complex of all WPF components, and it is a component that is directly designed to use data binding and in combination with the use of data templates it is an extremely flexible component. The component has many uses and can generally be used to present data arranged in rows and columns and thus in particular the result of database queries.

I will start as simple as possible and I have created a project called *ShowPersons*. To this project I have copied the classes

- *Zipcode*
- *Person*
- *DB*

from the project *SqlCommands*. The two first classes are unchanged while the last is expanded with a method

```
public List<Person> GetPersons()
{
  List<Person> list = new List<Person>();
  ...
  return list;
}
```

which returns all persons in the *Contacts* database. I want to show this persons in a *DataGrid* and the user interface is defined as:

```
<Grid>
  <DataGrid x:Name="grid" />
</Grid>
```

and in the code behind I bind the *DataGrid* to a database query:

```
public partial class MainWindow : Window
{
  private DB db = new DB();

  public MainWindow()
  {
    InitializeComponent();
    grid.ItemsSource = db.GetPersons();
  }
}
```

and if you run the program the result is:

Phone	Firstname	Lastname	Address	Zipcode	Mail	Title
10007891	Børge	Troelsen	Galgebakken 11	3912 Maniitsoq	jgbhv.host6434.dk	Skarpretter
10014966	Petrea	Knudsen	Hulvejen 56	8963 Auning	fsuva.host1743.dk	Sin mands kone
10021596	Magnus	Kristensen	Hulvejen 18	3500 Værløse	rprbv.host9918.dk	Præst
10027837	Ejner	Troelsen	Knivhjørnet 85	1875 Frederiksberg C	jzmuh.host9805.dk	Degn
10036541	Petrea	Bendtsen	Keddelmosen 1	1760 København V	ptzrw.host1970.dk	Feltmadras
10039269	Kristoffer	Andersen	Tørvegraven 66	1606 København V	bnwsx.host4461.dk	Præst
10045584	Viktor	Lassen	Jerntorvet 70	1163 København K	dsxnj.host9365.dk	Mestermand
10046517	Viktor	Jensen	Tørvegraven 64	1610 København V	zdopu.host2600.dk	Tater
10056387	Laurids	Rasmussen	Pistolstræde 59	1011 København K	vshpj.host2119.dk	Skarpretter
10061440	Alfred	Knudsen	Tingstedet 40	8950 Ørsted	ipthx.host5402.dk	Tater
10064174	Volmer	Kristensen	Køkkenmøddingen 73	1817 Frederiksberg C	yedzy.host7961.dk	Viking
10068968	Petrea	Albertsen	Knivhjørnet 44	1001 København K	deghv.host8835.dk	Sangerinde
10074233	Ingeborg	Lassen	Knivhjørnet 41	6622 Bække	kybld.host5880.dk	Feltmadras
10075420	Ingeborg	Kristensen	Hulvejen 0	5690 Tommerup	rfoac.host7854.dk	Klog kone
10081680	Volmer	Lassen	Voldgraven 82	4243 Rude	jwotc.host1426.dk	Præst

It can't be simpler, and it can be determined that showing the result of a query in a *DataGrid* is completely trivial. You should note that columns headers are the name of the properties in the class *Person* and what is shown in the *DataGrid* are the values of properties on the objects in the data source. You should also note that you can resize the columns, and if you click on a column header the rows are sorted ascending or descending after the values in this column.

The next example is called *ShowPersons1* and is a copy of the above project

Name	Address	Zip Code	Job title
Børge Troelsen	Galgebakken 11	3912	Skarpretter
Petrea Knudsen	Hulvejen 56	8963	Sin mands kone
Magnus Kristensen	Hulvejen 18	3500	Præst
Ejner Troelsen	Knivhjørnet 85	1875	Degn
Petrea Bendtsen	Keddelmosen 1	1760	Feltmadras
Kristoffer Andersen	Tørvegraven 66	1606	Præst
Viktor Lassen	Jerntorvet 70	1163	Mestermand
Viktor Jensen	Tørvegraven 64	1610	Tater
Laurids Rasmussen	Pistolstræde 59	1011	Skarpretter
Alfred Knudsen	Tingstedet 40	8950	Tater

and the example should show some of the options available to adjust the result. The C# code is completely unchanged, and the only place where anything has changed is in the XML section, but here, on the other hand, there are also many changes. I will not review the XML code here, but I will mention what to note.

1. The *DataGrid* does not have a column for all properties in the data source, and the columns are defined in XML.
2. The *DataGrid* is defined read-only.
3. The row heights are defined to make table more readable.
4. There is a column definition for each column.
5. The columns names (headers) are defined in XML.
6. The first column is a template column which is bound to both first name and last name.
7. Three of the columns has a left margin while the fourth has a right margin and is right aligned.

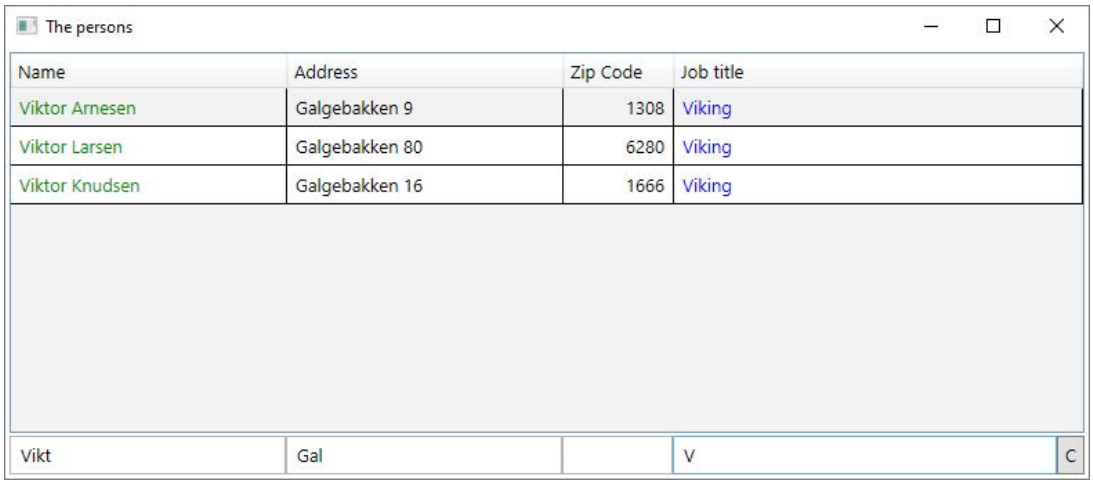
When you read the XML you should study how these adjustments are implemented.

EXERCISE 8: THE ZIP CODES AGAIN

In exercise 3 you write a program to maintain the *Zipcodes* table in the *Contacts* database. The program uses a *ListBox* to show the result of a search. In this exercise you should change the program such it instead uses a *DataGrid*. Try to create the *DataGrid* where you define the columns in XML.

4.9 FILTERING A DATAGRID

If a *DataGrid* has many rows you can advantageously use a filter for the table, and in the above example the *DataGrid* has 10000 rows and you need to scroll much to find a certain range. As an example I will use the same program as above (the project *ShowPersons2*), but this times is added a filter for each column in the *DataGrid*:



The user interface is expanded with four *TextBox* components, and the grid will shows all persons where

1. the combination of first name and last name contains the text in the first *TextBox*
2. the address contains the text in the second *TextBox*
3. the zip code starts with the text in the third *TextBox*
4. the job title contains the text in the last *TextBox*

In this case only 3 of the 10000 rows fulfill that. The last button is used to remove the filter, and the program again shows all rows.

To implement this functionality the classes *Zipcode* and *Person* are expanded and implements the interface *INotifyPropertyChanged*. It means that all properties raises a *ChangedEvent* when the value of a property is changed.

The change to the XML code consists primarily of adding the 4 *TextBox* components and the button. The filter needs to be updated every time a key is pressed (each time the contents of one of the four *TextBox* components are changed). Therefore, their *Text* property is bound to a property in code behind, and for the component for the address it is:

```
<TextBox x:Name="txtAddr" Grid.Row="1" Grid.Column="1" FontSize="13"
  Margin="0,2,0,0" VerticalContentAlignment="Center" Padding="5,0,0,0"
  Text="{Binding ElementName=main, Path=Addr,
    UpdateSourceTrigger=PropertyChanged}" />
```

Here you should note the value of *ElementName* which is an attribute defined in the *Window* element as a reference to the current object:

```
x:Name="main"
```

The *TextBox* is then bound to a property *Addr* in *MainWindow* and the last attribute in the binding means that the properties *set* method is performed every time the content of the field is changed.

The code behind is where it's all pasted together:

```
public partial class MainWindow : Window
{
  private DB db = new DB();
  private string name = "";
  private string addr = "";
  private string code = "";
  private string titl = "";
  private CollectionViewSource viewSource;

  public MainWindow()
  {
    InitializeComponent();
    viewSource = new CollectionViewSource()
      { Source = new ObservableCollection<Person>(db.GetPersons()) };
    viewSource.Filter += new FilterEventHandler(personFilter);
    grid.ItemsSource = viewSource.View;
  }
```

```
  private void personFilter(object sender, FilterEventArgs e)
  {
    if (e.Item != null)
    {
      Person person = (Person)e.Item;
      e.Accepted = Filter(person);
    }
  }

  private bool Filter(Person pers)
  {
    return (pers.Firstname + " " + pers.Lastname).Contains(name) &&
      pers.Address.Contains(addr) && pers.Zipcode.Code.StartsWith(code) &&
      pers.Title.Contains(titl);
  }

  public string Pers
  {
    get { return name; }
    set
    {
      name = value;
      CollectionViewSource.GetDefaultView(grid.ItemsSource).Refresh();
    }
  }

  public string Addr { ... }
  public string Code { ... }
  public string Titl { ... }

  private void cmdClear_Click(object sender, RoutedEventArgs e)
  {
    ...
  }
}
```

The class has four variables used to contain the value of the filter. The class also has a variable of the type *CollectionViewSource*, and it is a type that represents a filter as well as other functions to group the rows and sorting (not used in this example). A *CollectionViewSource* is defined for a data source, and in this case it is the collection with all *Person* objects selected from the database, but the collection wrapped in an *ObservableCollection<Person>*. To this *viewSource* is assigned a filter, which is an event handler performed for all *Person* objects in the source, and the handler defines for each *Person* where the object should be shown in the *DataGrid* or not. Where it should be the case is determined by the method *Filter*. A *CollectionViewSource* has a view, and it is this view which is bound to the grid as data source.

If the user enter in a *TextBox* the set methods for the four properties id performed, and they update the instance variable for the filter and performs refresh of the view. It means that the view is updated and the filter performed and only the rows that matches the filter are shown in the user interface.

4.10 EDITING A DATAGRID

It is also possible to enter directly in a *DataGrid* and as the last example to show the persons the project *ShowPersons3* shows how you can edit the content of the *DataGrid*, but also add new rows and delete existing rows. All changes you make in the *DataGrid* are stored back in the database. Immediately it sounds like a simple method to maintain a database table, whatever it is in many contexts, but it is not always the best solution, among which it can be difficult to keep an overview, and keep track of what changes have been made in the database.

The program has the same classes as above, but the class *DB* is extended with one new method, which returns a list with *Zipcode* objects with an object for each zip code in the database. The program should not be able to maintain the *Zipcodes* table, but the list of zip codes must be used to test that an entered zip code is legal and exists in the database.

Also the class *Person* is extended as it now implements the interface *IEditableObject*. The class must the implements three methods:

1. *BeginEdit()*
2. *CancelEdit()*
3. *EndEdit()*

The first method is called when the user starts editing of field, and the method store a copy of the content (the value of the property bound to a *TextBox*). The next is performed if the user cancel the editing (enter Esc) and the method restore the value of the property from the copy. The last method is called when the user end the editing.

The user interface is changed:

Phone	First name	Last name	Address	Zip Code	Email	Job title
10007891	Børge	Troelsen	Galgebakken 11	3912	jgbhv.host6434.dk	Skarpretti
10014966	Petrea	Knudsen	Hulvejen 56	8963	fsuva.host1743.dk	Sin manc
10021596	Magnus	Kristensen	Hulvejen 18	3500	rprbv.host9918.dk	Præst
10027837	Ejner	Troelsen	Knivhjørnet 85	1875	jzmuh.host9805.dk	Degn
10036541	Petrea	Bendtsen	Keddelmosen 1	1760	ptzrw.host1970.dk	Feltmadr
10039269	Kristoffer	Andersen	Tørvegraven 66	1606	bnwsx.host4461.dk	Præst
10045584	Viktor	Lassen	Jerntorvet 70	1163	dsxnj.host9365.dk	Mesterm
10046517	Viktor	Jensen	Tørvegraven 64	1610	zdopu.host2600.dk	Tater
10056387	Laurids	Rasmussen	Pistolstræde 59	1011	vshpj.host2119.dk	Skarpretti

so the *DataGrid* has a column for each column in the database table, and there is a *TextBox* to enter a filter for each column. The most important is the definition of the *DataGrid* in XML:

```
<DataGrid x:Name="grid" Grid.ColumnSpan="8"
AutoGenerateColumns="False"
  FontSize="13" RowHeight="28" IsReadOnly="False" RowHeaderWidth="20"
  CanUserAddRows="True" CanUserDeleteRows="True"
  AddingNewItem="grid_AddingNewItem"
  RowEditEnding="grid_RowEditEnding"
  CellEditEnding="grid_CellEditEnding"
  PreviewKeyDown="grid_PreviewKeyDown"
  Margin="0,0,0,2" SelectionMode="Single"
  BeginningEdit="grid_BeginningEdit" >
```

The first attributes are as before, but note that the *DataGrid* no longer is *read-only*. Also note that the grid now has a row-header. It is not absolutely necessary, but is included to have something to click on to select a row and also to show how to define row headers. The next two attributes defines that the user can add new rows and can delete rows and next is defined 5 event handlers:

1. *AddingNewItem*, performed when the user starts to enter an new row
2. *RowEditEnding*, performed when the user end editing a row
3. *CellEditEnding*, performed when the user end the editing of a cell
4. *PreviewKeyDown*, performed when the grid has focus (not a cell) and the user enter a key
5. *BeginningEdit*, performed when the user opens a row for editing

An important work in writing the code behind is to write these event handlers. A large part of the code relates to the filter and works in the same way as in the previous example except that there are now several fields for entering the filter. Since this is basically the same procedure as above, I will not show the code and I will just mention the 5 event handlers.

The last event handler store the object for the selected row in an instance object *Person*, if it not is the last blank row (ready to add a new row), where a new *Person* object is created:

```
private void grid_BeginningEdit(object sender,
SelectionChangedEventArgs e)
{
  object obj = ((DataGrid)sender).SelectedItem;
  if (obj is Person) person = (Person)obj; else person = new Person();
}
```

The first event handler is trivial and initialize a *bool* variable to mark where an edit operation is for a new person or an existing person:

```
private void grid_AddingNewItem(object
sender, AddingNewItemEventArgs e)
{
  newrow = true;
}
```

The third event handler is used to update a property for the current *Person* object:

```
private void grid_CellEditEnding(object sender,
DataGridCellEditEndingEventArgs e)
{
  try
  {
    UpdatePerson(e.Row, e.Column);
  }
  catch
  {
  }
}
```

The event arguments has values for the row and column and is in the method *UpdatePerson()*, used to update a property for current person. The second event handler updates the *DataGrid* and also the database:

```
private void grid_RowEditEnding(object sender,
DataGridRowEditEndingEventArgs e)
{
  if (newrow)
  {
    try
    {
      if (MessageBox.Show(...) == MessageBoxResult.Yes)
      {
        try
        {
          db.Insert(person);
        }
        catch
        {
          MessageBox.Show(...);
          persons.Remove(person);
        }
      }
      else persons.Remove(person);
    }
    catch
    {
    }
    finally
    {
      newrow = false;
    }
  }
  else
  {
    try
    {
      db.Update(person);
    }
    catch
    {
      MessageBox.Show(...);
      CollectionViewSource.GetDefaultView(grid.ItemsSource).Refresh();
    }
  }
  grid.UnselectAll();
}
```

The event handler is quite simple but you should note that the method updates the database. Finally, there is the last event handler that tests whether the user has enter the *Delete* key, and if so, the current row must be deleted in the database:

```
private void grid_PreviewKeyDown(object sender, KeyEventArgs e)
{
  DataGrid grid = (DataGrid)sender;
  if (Key.Delete == e.Key)
  {
    if (MessageBox.Show(...) == MessageBoxResult.No) e.Handled = true;
    else
    {
      try
      {
        object obj = grid.SelectedItem;
        if (obj != null)
        {
          Person pers = (Person)obj;
          db.Delete(pers.Phone);
          persons.Remove(pers);
        }
      }
      catch
      {
        MessageBox.Show(...);
        CollectionViewSource.GetDefaultView(grid.ItemsSource).Refresh();
      }
    }
  }
  grid.UnselectAll();
}
```

PROBLEM 1: DENMARK

In the previous book problem 3 you write a program *Denmark* where you could seek in a file containing information about Danish municipalities and Danish zip codes. In this exercise you must create a database with the same information and write a program to maintain the database. The source for this book contains three text files (not exactly the same as in the previous book):

1. *regions* with information about the 5 Danish regions where each line contains a region number and the region name separated by comma.
2. *zipcodes* consisting of lines with Danish zip codes, wherein each line consists of the code and the city name separated with a semicolon.
3. *municipalities*, which contains lines with information about Danish municipalities. Each line contains at least 6 fields separated by semicolon: Municipality number, municipality name, region number for this municipality, area in square kilometers, number of inhabitants and the year for number of inhabitants. The 6 fields can be followed of a number of (semicolon separated) zip codes, which denotes the zip codes used by this municipality.

You must note that each municipality belongs precisely to one region, and a zip code can be used of one or more municipalities.

1. As the first you must write a SQL script to create a database for this information, when the script must create four tables:

   1. *Region*, a table for the 5 regions
   2. *Zipcode*, a table for zip codes
   3. *Municipality*, a table for municipalities
   4. *Post*, a relation table for *Zipcode* and *Municipality*

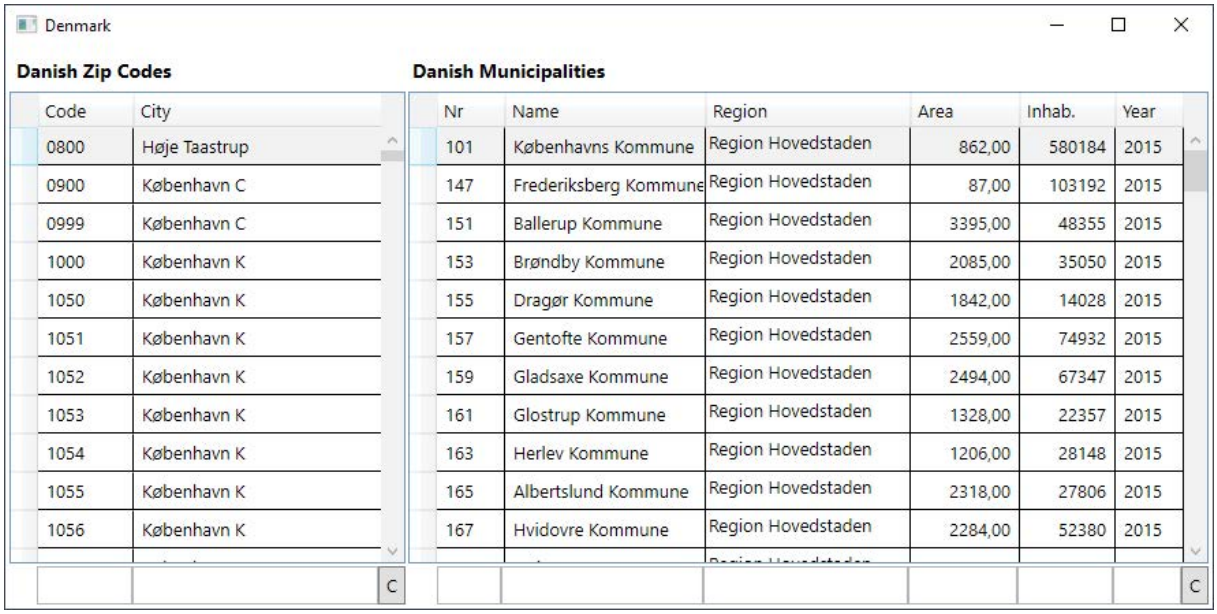
   The script must also contain SQL statements to insert the 5 regions in the table.

   Create the database by executing the script in *Microsoft SQL Server Management Studio*.

2. When you have created the database you must create a console application project as you can call *CreateDanmark*. The program should using the text files *zipcodes* and *municipalities* updates the three tables *Zipcode*, *Municipality* and *Post*.

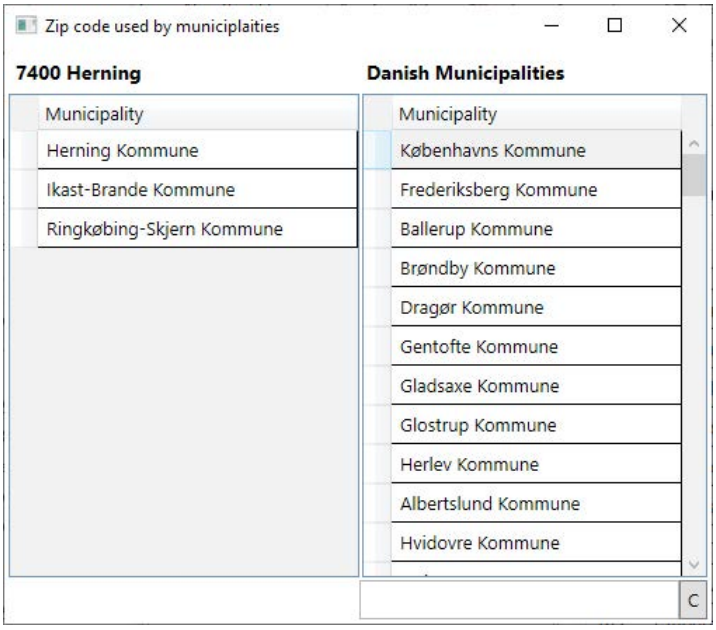
   After you have written and executed the program the result should be that you have a database with data for Danish municipalities and zip codes.

3. You must the write a program called *Denmark* that can be used to maintain the database. It must be a GUI program, and when the program is performed if must open the following window:



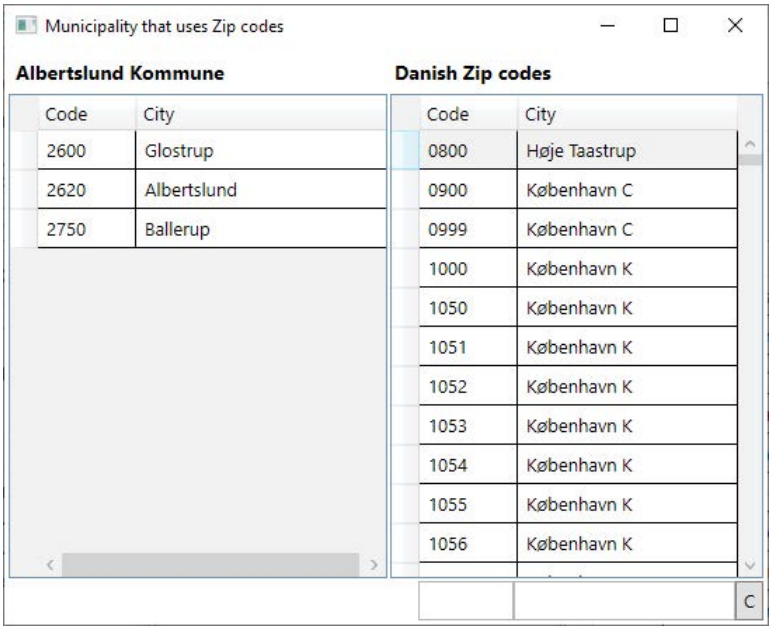
The window shows two *DataGrid* components, one for each of the two tables *Zipcode* and *Municipality*. Under each column there is a *TextBox* used as a filter, and it must be possible to edit the contents and save the changes in the database. It must also be possible to add new rows and delete existing rows. That is, that the program should work somewhat in the same way as the example above.

When you double click on a row header, for example the table for zip codes, the program must open a window that shows which municipalities that uses this zip code:



If you here double click on a row header in the left table, the program must remove the clicked municipality from this zip code, and if you double click on a row header in the right table (which contains all municipalities and has a filter), the program must add the clicked municipality to this zip code.

If you in the main window double click on a row header for a municipality the program must in the same way opens a window which shows the zip codes used by this municipality:

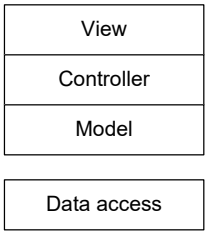


4.11 MVVM

If you write a slightly larger program with a graphical user interface, it will consist of many windows, where each window belongs to a specific function. The windows are activated by as example clicking a button, selecting a function in a menu or toolbar or otherwise, and seen from the programmer, windows are created by adding a new Window class to the project in Visual Studio. The program's code - business logic and data manipulation - is linked to event handlers placed in code behind, at least if you do nothing. All of the book's examples up to this point have been made that way, and that's fine too, as these are small examples where the purpose alone has been to show a specific WPF component or concept, but for larger programs it provides a flat architecture with very complex classes. The result is a program that is difficult to understand and thus difficult to maintain.

Therefore, one has tried to develop a template for a program's architecture, a template that is usually called a design pattern. In fact, it is not easy to develop a pattern that suits every situation, and there is also a tendency for the pattern to become so complex that it itself is

a problem. Therefore, you should use patterns correctly as a guide, but do not necessarily try to push any program into the pattern if for some reason it does not fit the task. Use that of the pattern that fits the task and benefit the result, but on the other hand, there should be no doubt that with a slightly larger program it is worthwhile to start with a pattern. The classic pattern for writing a program with a graphical user interface is MVC, which stands for *Model View Controller*. Basically, it's a matter of writing the program from a layered architecture:

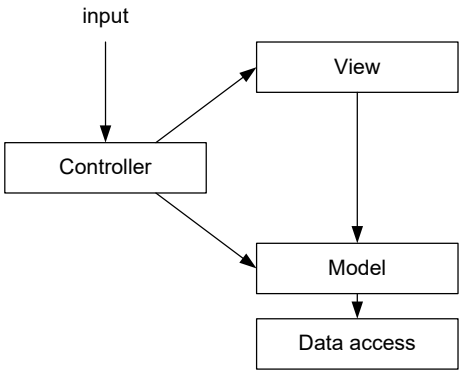


where

1. *View* is the part of the program that represents the user interface and thus classes for windows and other UI elements.
2. *Model* represents the program's data and provides an object-oriented wrapping of the data that the program should process. Also a large part of the program's business logic is found in the model layer in the form of methods in the model's classes.
3. It all has to be tied together, which happens in the *Controller* layer. User interaction in the form of entries and so on in the program view is passed on to the controller, which validates data before passing them on to the model, and it is also the controller's job to make the model's data available to the user interface.

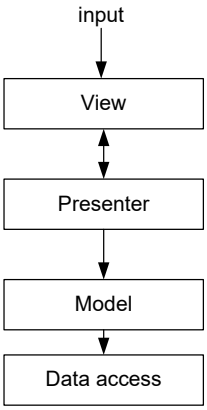
The lower layer is not really part of MVC, but often you want to place the code that must directly manipulate the physical data source into a special layer. That way, you can just replace the layer if the program has to use a data source of a completely different nature.

There is nothing magical about the architecture above, except it expresses an experience for a sensible split of the program code. There may well be several (and possibly fewer) layers, and that is solely determined by the specific task, but a starting point with a three-layer model has proved to be a sensible division of a program. In its simplest version, MVC is nothing more than folders containing classes with a well-defined responsibility, but a program's classes are linked to each other and are not isolated parts, and this is where the challenge lies and how to define an interaction between classes, so one, on the one hand, ensures the necessary coupling between the classes while ensuring the separation of classes so that they do not all know each other, and so that parts of the code can be replaced without interfering with the rest of the program. Often, you choose to draw MVC pattern as follows:



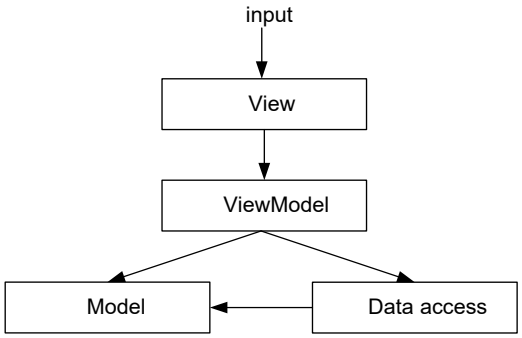
The main principle is to separate presentation and logic, and the central component is the controller, which receives the input and sends it to the model. It is the controller that creates both view and model, and while the view component knows the model, the model does not know the program's view or, for that matter, other parts of the program - except for the data access layer, which can be perceived as an interchangeable lower layer in the model. In most cases, MVC will also implement observer patterns, where for example the model sends a notification if its state changes.

The MVC pattern does not quite match the way Visual Studio builds a program where the starting point is a window and thus the program's view. Here you apply a modification of MVC, which you call MVP for *Model View Presenter*:



The main difference is that it is the program's view that receives the input. The controller layer is replaced with a presenter layer, which is closely linked to the view layer and has many of the same functions as the controller in MVC. It means that it is the program's view that drives the program and sends data to presenters that again pushes data back to the view typically in the form of a data binding. Another difference is that the view does not know the model. Here, too, the model can implement the observer pattern and send notifications when its state changes.

WPF has brought yet another change to the pattern, which is called MVVM for *Model View ViewModel*. First, a large part of the user interface is written as XML, and a large part of the controller (or presenter's) work is done automatically in the XML code with data binding. Similarly, the user interaction is different and is largely based on *ICommand* objects. The presenter's task is less and more WPF specific and consists primarily of preparing the model's data for data binding by for example to encapsulate a collection in an *ObservableCollection*, define converters and so on. The MVVM can be outlined as follows:



and compared to MVP, the main difference is that *ViewModel* does not know *View*. Both one-way and two-way communication between view and the view model are still possible, but this is done automatically with data binding, where elements of the view bind their *DataContext* property to an object in view model.

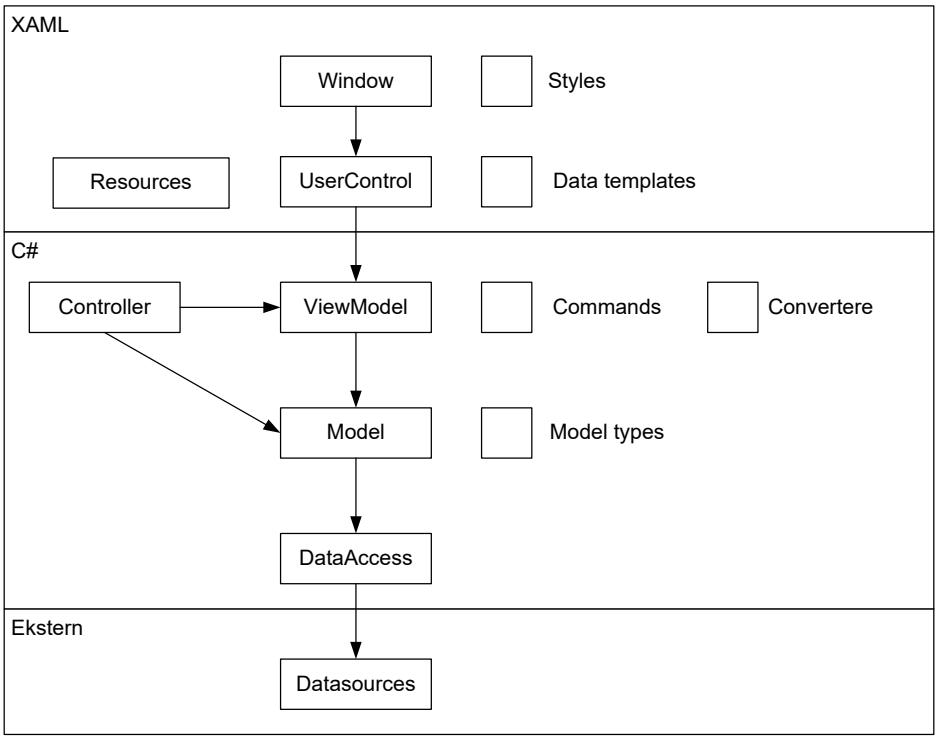
In general, event handlers in code behind are considered less fortunate, since the program logic is thus linked to the user interface and thus is difficult to test independently of the user interface components. It also becomes more difficult to change the UI, as each element has different event handlers. Therefore, all presentation logic must be moved from the view to view model in the form of *Commands*, which can then be bound to UI components.

When applying the pattern, you should follow some simple guidelines:

1. The data model classes provide an object-oriented approach to the program's data, and the classes will often contain the logic needed to manipulate the data. The model will often consist of two parts or layers, one layer being a data access layer.
2. *ViewModel* is a model for the view, but it must not inherit WPF classes or assume other properties of the components to which it binds. It will typically encapsulate the model types and thus prepare them for data binding.
3. All user interaction is handled by *Command* objects in *ViewModel* rather than event handlers in code behind.

4. The view contains all the components to which to bind, and it is recommended to bind the view's *DataContext* attribute to an object in *ViewModel*, as this binding is inherited downwards in the visual tree.
5. One should aim for a view implemented exclusively in XML and have as little code in code behind as possible.

In response, you can often think of a WPF program in the following way, where at the top you have the program's *View* written exclusively in XML. Next you have the program's *ViewModel*, but often combined with some *Controller* that has responsible for instantiating objects, and below you have the model:



The MVVM pattern sounds really simple, but it is actually not that easy to use, and the pattern tends to be complex and difficult to figure out - and that is obviously not the goal. Therefore, it is recommended:

1. Use the pattern when the task suits it. That is when the task has some size and a modular design with low coupling is needed.
2. Use that of the pattern that promotes the solution and increases the quality of the given task, but avoid pushing a solution into the pattern for the pattern's own sake.
3. Step forward and learn from the experience and above all from what others have done. Incorporate more principles as experience increases.

Of course, the best thing is to look at the pattern in practice, and I will use the pattern many times in the following.

4.12 HISTORY PEOPLE

To introduce MVVM I will show the development of a small program using the pattern. It should be said right away that it takes some practice to apply the pattern, and as long as it is a small program it can be hard to get the benefits, but for larger programs the benefits become clearer simply because the pattern sets a pattern for the program's architecture that can make maintenance easier.

The example is a program to maintain a simple database with data for historical persons. Specifically, the program must display a list (an overview) of people in the database. You must also be able to create new persons, edit existing ones and delete persons.

In the following, I will explain how the program is written specifically with emphasis on using the MVVM pattern, and the development can be used guidelines for using MVVM for small programs where I will develop the program in the following steps:

1. The database
2. The project
3. Data
4. Model
5. Data access layer
6. The MainWindow
7. ViewModel for MainWindow
8. Binding the MainWindow
9. The EditWindow

The database

The program's data and thus the data about people must be stored somewhere and for this purpose it is decided to use a database with a single table:

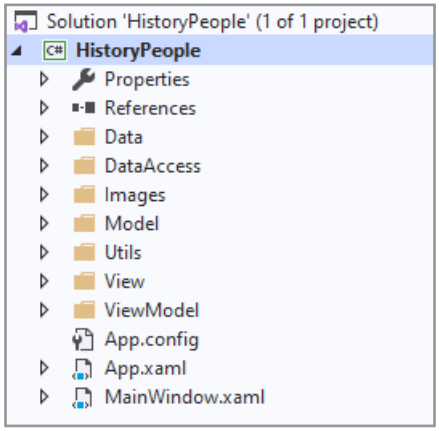
```
use master
go
drop database if exists History;
create database History;
go
use History
go

create table People
(
  Hist_Id INT NOT NULL IDENTITY(1,1) PRIMARY KEY,
  Hist_Name NVARCHAR(50) NOT NULL,
  Hist_From INT,
  Hist_To INT,
  Hist_Gender NCHAR,
  Hist_Title NVARCHAR(100),
  Hist_Country NCHAR(2),
  Hist_Text TEXT
);
```

The script is called *CreateHistory.sql* and after the script is performed in *Microsoft SQL Server Management Studio* the database is created and ready for use.

You should note that a script like the above can not only be used to create the database, but also terns as a documentation for the database and including what the individual columns are called and their types.

The project

As the next step I create a project in Visual Studio and at the same time I have determined the overall project architecture:



You should note that it is an ordinary WPF project. These directories may not always be used, but the purpose is as follows:

- *Data*, to the program's data files, if any such files.
- *DataAccess*, which is access layer created to the application's data source.
- *Images* that contain graphics in the form of images and icons.
- *Model* containing the model classes of the program.
- *View*, which contains classes for the program's windows.
- *ViewModel*, which contains the program's *ViewModel* classes and is often the most comprehensive folder.
- *Utils*, to other classes

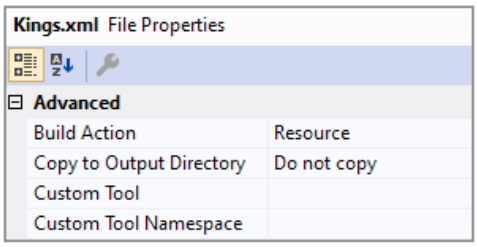
Typically, I will start from this structure, but as mentioned, there may be more directories, and for larger programs, the directories will be subdivided.

Data

In this case this directory contains a single file, which is an XML document with some person data used to initialize the database:

```
<?xml version="1.0" encoding="utf-8" ?>
<kings>
  <person name="Regnar Lodbrog" title="Viking" gender="M" />
  ...
</kings>
```

There are 54 persons. You should note that the file is defined as a resource:



and then is compiled into the program's assembly.

One can think of the file as the preparation of resources that the program should use. In this case, it is intended that it should be possible to reset the database corresponding to the contents of the file, but also to be able to quickly get data into the database in connection with testing.

The model

Then I will write the model and in this example there is only one class (there is only one table in the database) which should represents a person with 8 properties, and then one property for each column in the database table:

```
namespace HistoryPeople.Model
{
  public class Person : IDataErrorInfo
  {
    private static readonly string[] properties =
      { "Id", "Name", "From", "To", "Gender", "Title", "Country", "Text" };

    public int Id { get; set; }
    public string Name { get; set; }
    public int? From { get; set; }
    public int? To { get; set; }
    public char? Gender { get; set; }
    public string Title { get; set; }
    public string Country { get; set; }
    public string Text { get; set; }

    public override bool Equals(object obj) { ... }

    public override int GetHashCode() { ... }

    public override string ToString() { ... }
    {
      return Name;
    }

    public bool IsValid
    {
      get
      {
        foreach (string property in properties)
          if (Validate(property) != null) return false;
        return true;
      }
    }

    string IDataErrorInfo.Error
    {
      get { return IsValid ? null : "Illegal Person object"; }
    }

    string IDataErrorInfo.this[string propertyName]
    {
      get { return Validate(propertyName); }
    }
```

The class is quite simple and is a typical model class which in this case models a row in a database table. The class implements an interface *IDataErrorInfo*, which is a contract for when an object is legal - its properties have legal values. The interface defines a method that returns a string that is interpreted as an error code or error message. If the object is legal, the method returns *null*, and otherwise it returns a text describing the error (an error message). Finally, the interface defines an index override for *string* that is interpreted as the name of a property to be validated.

The class also has a property *IsValid* that specifies whether the properties to be validated have legal values. The properties in question are defined in a static array at the beginning of the class. You can discuss whether this validation of properties makes sense or not, but the control must be done somewhere and the interface *IDataErrorInfo* defines a standard way to implement this control. You should note that the way in principle has nothing to do with WPF or GUI programs.

DataAccess layer

This layer contains also only one class called *DB*. It is a comprehensive class which provides the necessary database operations:

```
namespace HistoryPeople.DataAccess
{
  public class DB
  {
    private SqlConnection connection = new SqlConnection(...);

    public event EventHandler<PaEventArgs<Person>> PersonAdded;
    public event EventHandler<PaEventArgs<Person>> PersonModified;
    public event EventHandler<PaEventArgs<Person>> PersonRemoved;
    public event EventHandler<PaEventArgs<int>> PersonsLoaded;

    public DB()
    {
    }

    public void Add(Person person) { ... }
    public void Update(Person person) { ... }
    public void Delete(Person person) { ... }
    public void Reload() { ... }
```

```
    public bool Contains(Person pers)
    {
      return persons.Contains(pers);
    }

    public IEnumerator<Person> GetEnumerator()
    {
      return persons.GetEnumerator();
    }

    public List<Person> GetPersons() { ... }
  }
}
```

The class is similar to classes defined in the above examples in this book and starts by defining a connection to database.

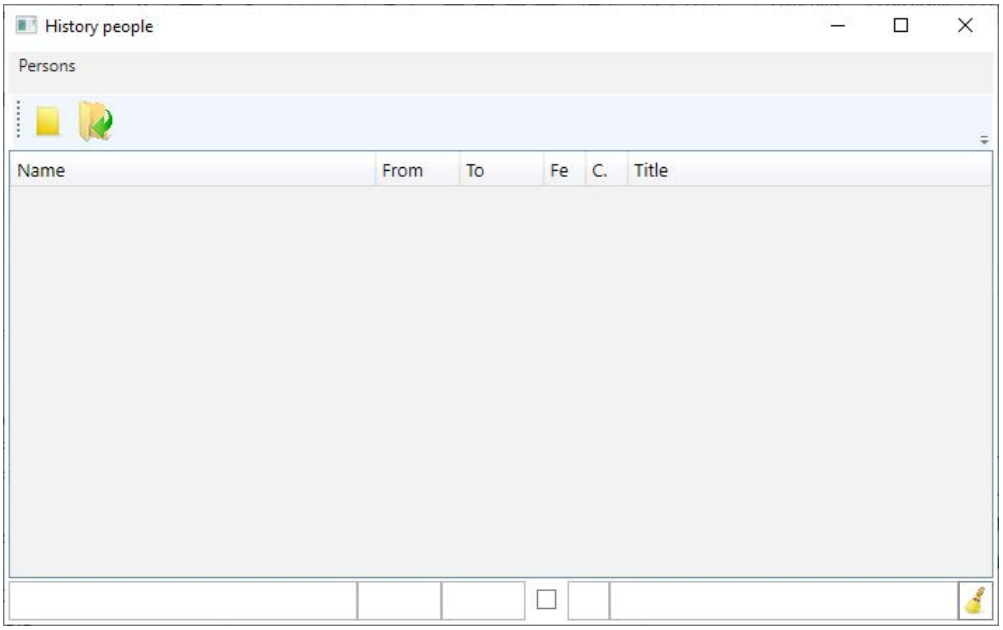
The method *Reload()* delete the database table and initialize the database from the XML file. When you read the code you will see that I uses a LINQ expression and for the moment you should just accept the code as it is. I returns to LINQ in the book C# 8.

The other methods does not contain anything new, but the methods *Add()*, *Update()*, *Remove()* and *Reload()* all raises an event if they are performed with correct result. Any user of the class can thus register as an observer for these events and in this way be notified that the database state has changed. You should note that the *EventArgs* for these events is a generic type *PaEventArgs<T>*. This type is in the folder *Utils*. Since the table *People* has several columns that accept null values, this is something to consider, and when you read the code, you should pay special attention to how it is resolved.

You should note that neither the model layer nor the data access layer knows anything about the other layers, and the only communication that can occur from the model and data access layers to the other layers is that the class *DB* raises an event, but the class doesn't know anything about, whether there are other classes receiving these events.

The MainWindow

As the next step I will write the XML for the *MainWindow*. It is not the finished code and the window does not show data (the *DataGrid* is empty) as well as the buttons and menu items must be assigned to commands, but it illustrates which items must be bound to properties in the *ViewModel* layer:



In this case the window has a menu (only four menu items) and a toolbar with two buttons:

1. Create a new Person
2. Modify the selected person in the *DataGrid*

These two commands is also found as menu items, and the menu has an item to reload the database as an item to close the program.

Below are *TextBox* components used for filters and a *CheckBox* when the column for gender should be a check box column.

ViewModel

It is the most important layer that contains most classes and thus also most of the program code. The layer is the link between the program's UI classes and the program's model classes, and the idea is that classes in the program's view contain only XML, while the C# code with the program's logic resides in the *ViewModel* layer. It is the *View* classes that must create the classes in *ViewModel*, and it is the job of the *ViewModel* classes to encapsulate the model classes so that the *View* classes can bind to them.

I will start with a class *ViewModelBase*, which is the base class for all *ViewModel* classes:

```
namespace HistoryPeople.ViewModel
{
  public abstract class ViewModelBase : INotifyPropertyChanged, IDisposable
  {
    public event PropertyChangedEventHandler PropertyChanged;

    protected virtual void OnPropertyChanged(string propertyName)
    {
      if (PropertyChanged != null)
        PropertyChanged(this, new PropertyChan
        gedEventArgs(propertyName));
    }

    public void Dispose()
    {
      this.OnDispose();
    }

    protected virtual void OnDispose()
    {
    }
  }
}
```

The class does not contain much, and one can discuss whether it is worth the work, but for the sake of data binding, the *ViewModel* classes must implement the interface *INotifyPropertyChanged*, and so it may be an idea (and most do) to move this functionality to a base class. In addition, the class implements *IDisposable* in order to override *OnDispose()* in the derived classes in case if some cleanup is needed. It is not used in this example, but it may be excellent to prepare the *ViewModel* classes for this functionality.

The base *ViewModel* class in this example is *PersonModel*, which can be considered as a wrapper class for the model class *Person* in order to prepare the model for data binding. The class represents a single *Person*, but also has a reference to the model's data access layer:

```
namespace HistoryPeople.ViewModel
{
  public class PersonViewModel : ViewModelBase, IDataErrorInfo
  {
    private Person person;
    protected DB db;

    public PersonViewModel(Person person, DB db)
    {
      this.person = person;
      this.db = db;
    }

    public Person Seletced
    {
      get { return person; }
    }

    public int Id
    {
      get { return person.Id; }
    }

    public string Name
    {
      get { return person.Name; }
      set
      {
        person.Name = value;
        OnPropertyChanged("Name");
      }
    }

    public int? From { ... }
    public int? To { ... }
    public char? Gender { ... }
    public string Title { ... }
    public string Country { ... }
    public string Text { ... }

    public void Clear() { ... }
```

```
    public bool Update()
    {
      if (person.IsValid && Id > 0)
      {
        db.Update(person);
        return true;
      }
      return false;
    }

    public bool Add() { ... }
    public bool Remove() { ... }

    string IDataErrorInfo.Error { ... }
    string IDataErrorInfo.this[string propertyName] { ... }
  }
}
```

One of the class's tasks is to encapsulate the class *Person* properties so that they can be bound to the UI, and you should notice how they call the base class's *OnPropertyChanged()* method. In addition to that, the class has methods for modifying a person as well as adding a new *Person* to the database. Note how these methods uses the *Person* class's *IsValid* property.

The class also implements the interface *IDataErrorInfo*, which primarily encapsulates the same functionality as in the *Person* class. You should especially note the last statement, which ensures that the *CommandManager* object raises a *RequerySuggested* event.

The program has another window than *MainWindow*, which is a window to create a new *Person* and modify an existing *Person*. Typically, the *ViewModel* layer will contain a class for each window, and this is the case here as well. These classes implements the program's *Command* objects and thus take care of the user's interaction with the program. To ease it a bit, a class *RelayCommand* has been defined, which has the sole purpose of making it easier to define commands. The class is simple (and one can discuss its eligibility), but the code is:

```
namespace HistoryPeople.ViewModel
{
  public class RelayCommand : ICommand
  {
    readonly Action<object> execute;
    readonly Predicate<object> canExecute;

    public RelayCommand(Action<object> execute) : this(execute, null)
    {
    }

    public RelayCommand(Action<object> execute,
    Predicate<object> canExecute)
    {
      this.execute = execute;
      this.canExecute = canExecute;
    }

    public bool CanExecute(object parameter)
    {
      return canExecute == null ? true : canExecute(parameter);
    }

    public event EventHandler CanExecuteChanged
    {
      add { CommandManager.RequerySuggested += value; }
      remove { CommandManager.RequerySuggested -= value; }
    }

    public void Execute(object parameter)
    {
      execute(parameter);
    }
  }
}
```

The class has two methods which are a method *Execute()* that has an *object* as parameter and a method *CanExecute()* that also has an *object* as parameter. The first is the method which performs what the command should do, and the other test whether the command can be executed. *Action* is a generic delegate representing a *void* method with one argument, and in the class *RelayCommand* it is used to reference the *Execute()* method. The class also has a *Predicate* to reference the *CanExecute()* method. The class has two constructors to initialize these two variables, when the variable *canExecute* can be *null* which means that the *Execute()* method always can be performed. The most important reason for the class is that a user (an object) can register as observer the command's execution.

The most comprehensive of the *ViewModel* layer classes is *MainViewModel*, which is the model for the main window. The class is not very complex, but it is comprehensive as it has to implement commands for respectively the menu and the toolbar. Below is shown a part of the code, and you should especially note how to implements the commands:

```
public class MainModel : ViewModelBase
{
  private DB db = new DB();
  private ObservableCollection<PersonModel> persons =
    new ObservableCollection<PersonModel>();
  private CollectionViewSource viewSource;
  private RelayCommand reloadCommand;
  private RelayCommand addCommand;
  private RelayCommand modCommand;
  private RelayCommand exitCommand;
  private RelayCommand clearCommand;
  private string name = "";
  private string from = "";
  private string to = "";
  private string country = "";
  private string title = "";
  private bool? female = null;
  private PersonModel personModel;

  public event EventHandler<EventArgs> ModelChanged;
  public event EventHandler<EventArgs> ModelUnselected;
  public MainModel()
  {
    List<Person> list = db.GetPersons();
    foreach (Person pers in list) persons.Add(new PersonModel(pers, db));
    viewSource = new CollectionViewSource() { Source = persons };
    viewSource.Filter += new FilterEventHandler(personFilter);
    db.PersonAdded += PersonAdded;
    db.PersonModified += PersonUpdated;
    db.PersonRemoved += PersonRemoved;
  }

  public void PersonAdded(object sender, PaEventArgs<Person> e) { ... }
  public void PersonRemoved(object sender, PaEventArgs<Person> e) { ... }
  public void PersonUpdated(object sender, PaEventArgs<Person> e) { ... }

  private void personFilter(object sender, FilterEventArgs e) { ... }
```

```
public string FName
{
  get { return name; }
  set
  {
    name = value;
    OnPropertyChanged("FName");
    if (ModelChanged != null) ModelChanged(this, EventArgs.Empty);
  }
}

public string FFrom { ... }
public string FTo { ... }
public string FCountry { ... }
public string FTitle { ... }
public bool? FFemale { ... }

public ICollectionView Persons
{
  get { return viewSource.View; }
}

public RelayCommand ClearCommand
{
  get
  {
    if (clearCommand == null)
      clearCommand = new RelayCommand(param => ClearFilter());
    return clearCommand;
  }
}

public RelayCommand ReloadCommand
{
  get { ... }
}

public RelayCommand AddCommand
{
  get { ... }
}
```

```
  public RelayCommand ModCommand
  {
    get
    {
      if (modCommand == null) modCommand =
        new RelayCommand(p => Modify(p), p => CanModify(p));
      return modCommand;
    }
  }

  public RelayCommand ExitCommand
  {
    get { ... }
  }

  ...
}
```

In particular, note the following:

- The class creates a *DB* object.
- The class defines an *ObservableCollection* for the persons read in the database, but all encapsulated in *PersonModel* object.
- The *ObservableCollection* is wrapped in a *CollectionViewSource* to implement the filter.
- The class handles events from the database when the state of the database is changed.
- The class defines events to notify the user interface which fires when it is necessary to update the program's *DataGrid*.
- Commands are defined, and most of the code applies to these commands. The goal is that the commands can be bound to UI components. Instead, consider placing these commands in their own class, which (especially with many commands) can provide a nicer code.

Binding the MainWindow

With the model for the *MainWindow* in place you can define the data binding. In XML you must define necessary converters as windows resources, and in this case the classes for converters are defines in a file *Converters* in the *ViewModel* layer. The binding for the *MainWindow* is defined in code behind where the property *DataContext* if bound to a *ViewModel* object:

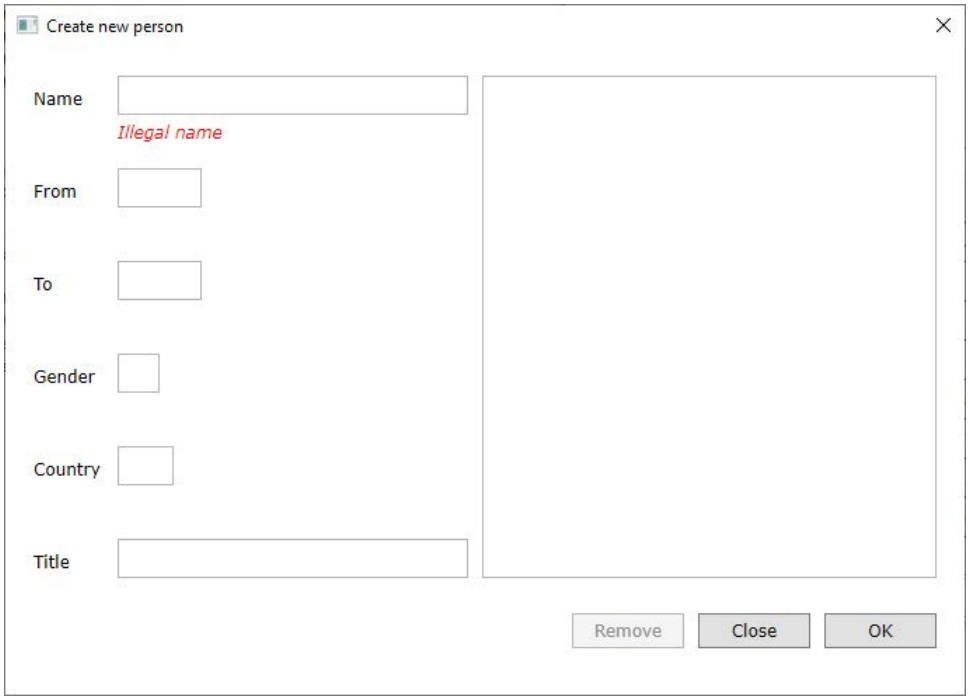
```
public partial class MainWindow : Window
{
  private MainModel viewModel;
  public MainWindow()
  {
    InitializeComponent();
    MainModel mvvm = new MainModel();
    mvvm.ModelChanged += PersonsChanged;
    mvvm.ModelUnselected += PersonsUnselected;
    DataContext = viewModel = mvvm;
    grid.Sorting += new DataGridSortingEventHandler(SortHandler);
  }

  public void PersonsChanged(object sender, EventArgs e)
  {
    CollectionViewSource.GetDefaultView(grid.ItemsSource).Refresh();
  }

  public void PersonsUnselected(object sender, EventArgs e)
  {
    grid.UnselectAll();
  }

  void SortHandler(object sender, DataGridSortingEventArgs e) { ... }
}
```

A *DataGrid* support sorting of columns, but sometimes it is necessary to define your own sorting. This is done with *IComparer* objects, which are defined in the file *Converters*.

With the *DataContext* bound you can define all binding attributes in XML, and you should especially notice how the buttons and menu items are bound to the *Command* objects defined in *MainViewModel*.

The EditWindow

This is a dialog box used to create and edit information about a person:



I will not show the code here, but in the *View* layer is define a single class for this dialog. Note that the window uses data validation as defined in the model class and the class *PersonModel*:

- A *DataTemplate* for a *TextBlock* is defined as a resource. Its *Text* property is bound to an *ErrorContent* property and the template is used to display an error message if a field contains an illegal value. This is where it is important that the model class implements the interface *IDataErrorInfo*.
- The input fields that is *TextBox* controls are bound to properties in the model. In connection with the binding, it is defined that the content must be validated and that the validation must occur every time the content is changed. A field is associated with a *ContentPresenter* control which is used to display the error message.

In the *ViewModel* layer is defined a *ViewModel* for this dialog called *EditViewModel*. The buttons are bound to a *Command* in the model. Thus, no usual event logic has been used, primarily to delegate the processing of the user interaction to logic in the *ViewModel layer*, thus making the UI as thin as possible.

A last remark

MVVM comes up when writing a WPF program, and in principle you could be consistent and apply the pattern every time. However, the program must be of some size in order for it to make sense. The pattern is complex and difficult to overview, but large WPF programs are complex, and for larger programs, the pattern is worth the work as it provides a modular program and thus enhances the maintenance of the program. My approach is the following:

1. I'll start by creating the directory structure as shown in this example.
2. Next, I grab the model layer and write all the model classes. In this example, there is only one, but typically there will be more or many.
3. Then I prepare the program's data and other resources that the program should use and thus the contents of the folder *Data*. Often there will be none, and the program may also use external data sources such as databases, which means that the program's *App.config* file needs to be changed.
4. Next, I write the necessary repository, and that is the data access layer. Often there will be more or many tables and the work can be quite extensive. When set aside, the model is ready and testable, and sometimes I write a test program, so I can be sure that it will work properly before addressing the next part of the work. Below I will mentioned the *Entity Framework*, and the purpose of this framework is exactly to automatic perform the step and also step 2.
5. After the model is in place, I grab the user interface. I can't finish the UI as there is no view model to bind to, but I can create all windows and custom components including styles and resource dictionaries. In practice, I will sometimes switch in order and start with the user interface, as this will quickly reach something that you can show to the program's future users.
6. With the user interface and model ready, the classes in the *ViewModel* layer must be written. It can be a big job as a great deal of business logic is placed here, but there are rarely the very big uncertainties as most decisions are made.
7. Finally, it all needs to be linked, which includes the UI to instantiate model objects to which it can bind.

PROBLEM 2: THE WORLD

You must write a program to maintain a database with information about countries defined by the following script:

```
use master
go
drop database if exists World;
create database World;
go
use World
go

create table Continent (
  Cont_Id INT NOT NULL IDENTITY(1,1) PRIMARY KEY,
  Cont_Name NVARCHAR(20) not null
);

create table Country (
  Coun_Code2 NCHAR(2) PRIMARY KEY,
  Coun_Code3 NCHAR(3),
  Coun_Name NVARCHAR(50) not null,
  Coun_Area DECIMAL(10, 2),
  Coun_Popu INT,
  Coun_Curr NCHAR(3),
  Coun_City NVARCHAR(50),
  Coun_Inha INT,
  Coun_Text TEXT,
  Coun_Cont INT,
  FOREIGN KEY (Coun_Cont) REFERENCES Continent
);

insert into Continent (Cont_Name) values('Asia');
insert into continent (Cont_Name) values('Africa');
insert into continent (Cont_Name) values('North America');
insert into continent (Cont_Name) values('South America');
insert into continent (Cont_Name) values('Antarctica');
insert into continent (Cont_Name) values('Europe');
insert into continent (Cont_Name) values('Oceania');
```

There are two tables where the first table contains the names of the seven continents, and the second contains information about countries:

1. 2-character country code used as primary key
2. 3-character country code
3. the country name
4. the country area in square kilometers

5. the number for the country population
6. the currency code
7. the country capital city
8. the number of inhabitants in the capital
9. a description
10. a foreign key to country continent

You should ignore that some countries belong to more continents, and in this task all countries belongs to one primary continent.

Create this database using Microsoft SQL Server Management Studio. You can call the database for *World*.

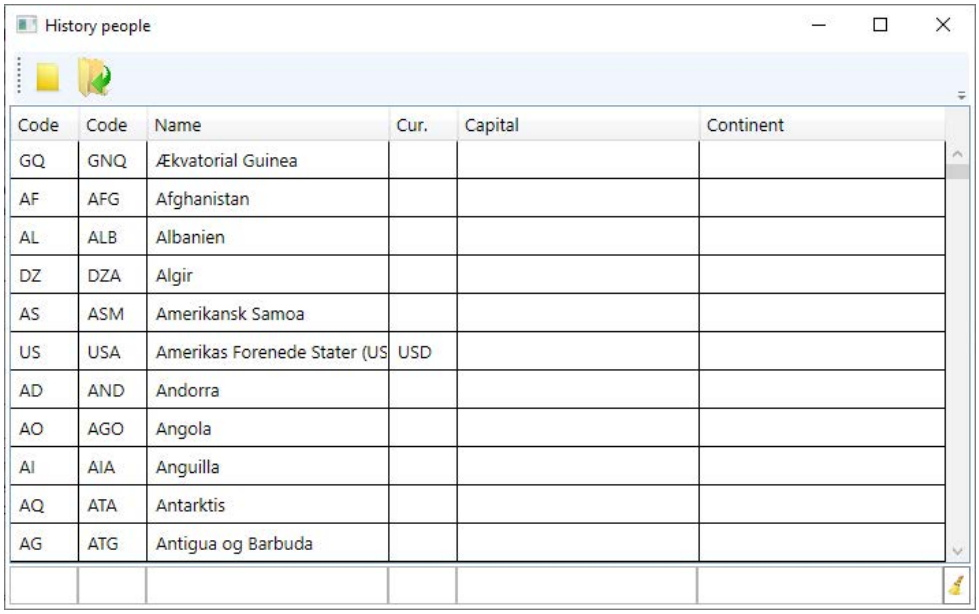
The source directory for this book contains a text file called *countries.txt* which consists of line with country codes:

1. the 2-character code
2. the 3-character code
3. the country name (in Danish)

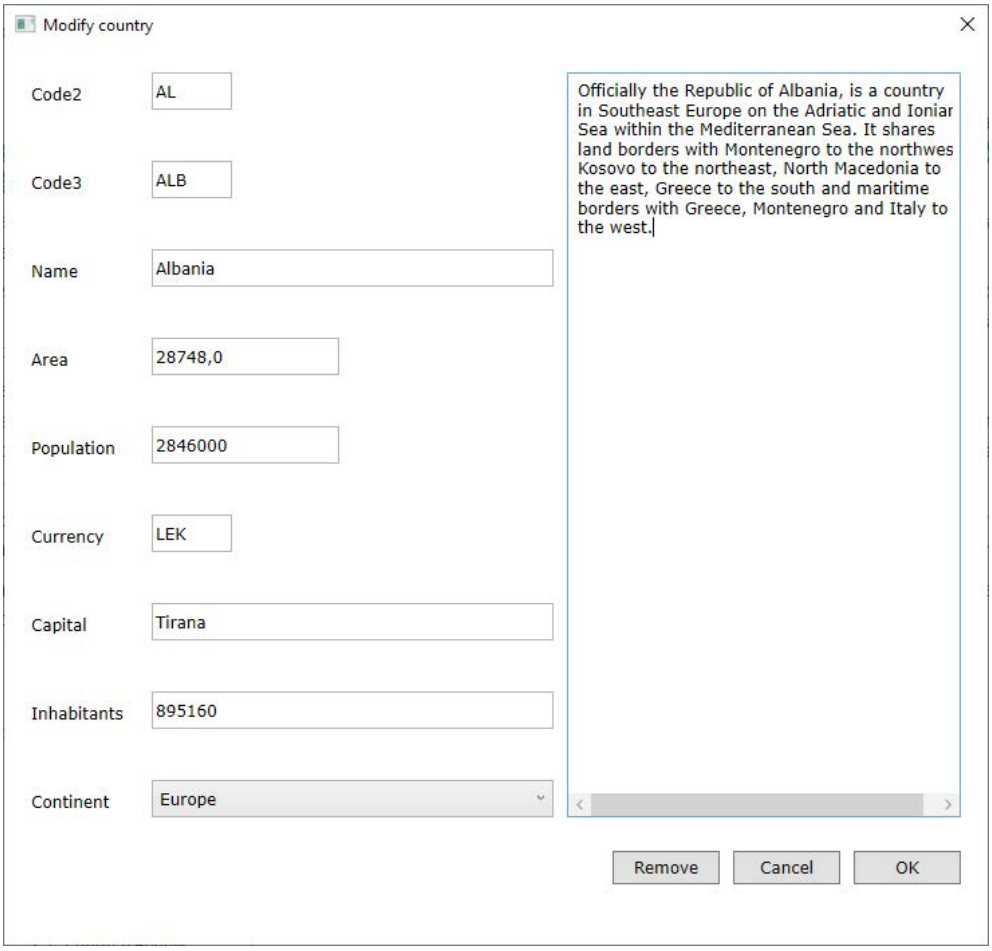
where the three fields are separated by semicolon. For a few countries there is also information about currency. You should next write a console application that using the file *countries.txt* initializes the database with data and create a row in the *Country* table for each line in the file.

As the next you must write a program with two windows to maintain the table *Country* when the program must be written in the same way as the program *HistoryPeople* using the MVVM design pattern. You should solve the task by following the same procedure outlined with the *HistoryPeople* example and remember that the goal of the task is to show a simple example of using *Model-View-View-Model*.

The program should open the following window:



which shows an overview over the countries and with a filter below each column. The toolbar has two buttons (there is no menu) and is used to create a new country and edit the country for the selected row. When you click one of the two buttons the program should open the following dialog box, here shown how to edit information about Albania:



5 DISCONNECTED ADO

The above examples illustrate all the classic approach to databases as opening a connection, performing a database operation and closing the connection again. This strategy is not always preferred, and especially not if there are many and frequent database operations and the database server is on another machine where the communication is done over a network. The classic approach means that as a programmer you have to write a lot and the program code that is closely linked to the relational database model and partly that it takes time to open and close connections. In this chapter I will show another approach called disconnected ADO.

The idea behind disconnected database access is slightly different, where an adapter object is inserted between the database and the application. The task of the adapter is to extract data from the database and copy it to memory objects where they can be manipulated. When data is then manipulated, for example by adding new rows or changing existing rows, the adapter can reconnect to the database and write all changes back to the database. The method is called disconnected, since during the period in which data is manipulated, there is no connection to the database.

One of the key benefits of using a data adapter is that you get an object-oriented model of the database consisting of objects that you can work with in the same way as any other object without the distinction of how the objects are represented in the database. The price is of course a number of new types, all of which are defined in the namespace *System.Data*. The most important are:

- *DataColumn* representing a column in a table
- *DataRow* representing a row in a table
- *DataTable* representing a table
- *DataSet* representing a database - that is an extract of a database in the form of a number of tables

Also mentioned is the type *DataRelation*, which represents a relationship between two tables (a foreign key).

Next, there is the adapter where an abstract class is defined in *System.Data.Common*, which is called *DbDataAdapter*. Of course, there are correspondingly specific classes for the individual providers.

In the following, I will show a few examples of how disconnected ADO works, but I will only go into the basic principles, as there is a further encapsulation, which is the topic of the next chapter. I will use a console application called *DisconnectedADO*.

5.1 A SQL SELECT

The following method connects to the database *Contacts* and selects some rows from the *Zipcodes* table and prints the result on the console:

```
static void Test1()
{
  SqlConnection connection = new SqlConnection("...");
  SqlDataAdapter adapter = new SqlDataAdapter(
    "SELECT * FROM Zipcodes WHERE City LIKE 'Sk%'", connection);
  DataTable table = new DataTable();
  adapter.Fill(table);
  foreach (DataRow row in table.Rows)
    Console.WriteLine(row[0] + " " + row["City"]);
}
```

The method creates in the same way as above a connection to the database, and next a *SqlDataAdapter* for a SQL SELECT is defined and a *DataTable*. So far, no connection has been made to the database, but this is done in the following statement where the *DataTable* object is filled. It means that the adapter opens a connection to the database, executes the SQL SELECT statement and loads the rows to the *DataTable* object, and then the connection is automatically closed.

You now have a memory copy of the data retrieved from the database, and they can then be processed in memory. In this case, data is in a *DataTable*, and such an object has a collection called *Rows*, which represents the rows read. Here, a *foreach* loop is used to iterate through the table and print each row on the screen.

In this example, there is no benefit to using an adapter rather than a reader - perhaps even the opposite, as the only thing that happens is that a SQL SELECT is performed and the result is printed on the screen, but if data had to be modified and the database is updated, the adapter is an advantage.

5.2 THE ADAPTER

A *DataAdapter* has four commands - or may have four commands - and correspondingly it has four properties:

- *SelectCommand*
- *InsertCommand*
- *UpdateCommand*
- *DeleteCommand*

the first being often created on the basis of a string which is passed to the adapter's constructor. The first command is executed when the *Fill()* method of the adapter is executed and used to specify which data to retrieve from the database. The other three represent respectively a SQL INSERT, UPDATE, and DELETE command and are executed if the adapter's *Update()* method is performed. They ensure that the changes made to the content of a *DataSet* are then written back to the physical database. As mentioned, a connection to the database is opened automatically when the *Fill()* method is executed and it is automatically closed again. The same happens with the method *Update()*, which automatically re-opens the connection from the *Fill()* method and executes the adapter's update methods. In principle (and partly also in practice) it is the programmer's job to write the code for these update methods, which can be difficult and requires one to know its SQL, but for many applications there is good support to get from Visual Studio.

I want to show an example that performs some updates to the *Zipcodes* table:

```
static void Test2()
{
  SqlConnection connection = new SqlConnection("...");
  SqlDataAdapter adapter = new SqlDataAdapter(
    "SELECT * FROM Zipcodes WHERE Code LIKE '78%'", connection);
  DataSet ds = new DataSet();
  adapter.Fill(ds);
  DataTable table = ds.Tables[0];
  Show(table);
  table.PrimaryKey = new DataColumn[] { table.Columns["Code"] };
  Insert(table);
  Show(table);
  Update(table);
  Show(table);
  Delete(table);
  Show(table);
  adapter.InsertCommand = CreateInsertCommand(connection);
  adapter.UpdateCommand = CreateUpdateCommand(connection);
  adapter.DeleteCommand = CreateDeleteCommand(connection);
  adapter.Update(ds);
  Show(connection);
  Remove(connection);
}
```

First, a connection to the database is established. Next, an adapter is defined that extracts all zip codes starting with 78, but this time the data is filled in a *DataSet* and not a *DataTable*. It is only to show the use of a *DataSet* which is a class representing more tables. In this case there is only one. When the *DataSet* is filled and I have defined a reference to the table the method *Show()* is called that prints the rows selected form the database:

```
static void Show(DataTable table)
{
  Console.WriteLine("----------------------------------------");
  foreach (DataRow row in table.Rows) Show(row);
  Console.WriteLine("========================================");
}

static void Show(DataRow row)
{
  Console.WriteLine(row["Code"] + " " + row["City"]);
}
```

The method *Fill()* retrieves only data and not database constraints, and therefore what is the primary key must be manually defined. Then some *Insert()*, *Update()*, and *Delete()* methods are executed, all of which work on the *DataTable* object and perform some database operations. These methods are basically simple and there is not much to notice.

```
static void Insert(DataTable table)
{
  if (!table.Rows.Contains("7891")) Insert(table, "7891", "A small town");
  if (!table.Rows.Contains("7892")) Insert(table, "7892", "The big city");
  if (!table.Rows.Contains("7893")) Insert(table,
  "7893", "The last station");
  if (!table.Rows.Contains("7891")) Insert(table, "7891", "A small town");
}

static void Insert(DataTable table, string code, string city)
{
  DataRow row = table.NewRow();
  row[0] = code;
  row[1] = city;
  table.Rows.Add(row);
}

private static void Update(DataTable table)
{
  Update(table, "7891", "A very small town");
  Update(table, "7894", "Another city");
}
```

```
private static void Update(DataTable table, string code, string city)
{
  DataRow row = table.Rows.Find(code);
  if (row != null) row[1] = city;
}

private static void Delete(DataTable table)
{
  DataRow row = table.Rows.Find("7892");
  if (row != null) row.Delete();
}
```

Note, however, how *Insert()* tests whether the table already contains a row with that key:

```
table.Rows.Contains("7891")
```

The parameter is is object that is interpreted as the primary key, so it is necessary to tell which column contains the primary key. Also note how to test in the methods *Update()* and *Delete()* if there is a row with a specific primary key:

```
table.Rows.Find(code)
```

After performing one of the three methods, the contents of the table are printed on the screen so that the changes can be noted. Note that all changes have occurred in memory, that is, only in the object of the type *DataTable* which in turn is part of the *DataSet* object - there have been no changes in the physical database or no SQL statements have been executed. That's the next step.

Three methods in the program create *SqlCommand* objects respectively a SQL INSERT, SQL UPDATE and SQL DELETE operation:

```
private static SqlCommand CreateInsertCom
mand(SqlConnection connection)
{
  SqlCommand cmd = new SqlCommand(
    "INSERT INTO Zipcodes (Code, City) VALUES (@
    Code, @City)", connection);
  cmd.Parameters.Add("@Code", SqlDbType.VarChar, 4, "Code");
  cmd.Parameters.Add("@City", SqlDbType.VarChar, 30, "City");
  return cmd;
}

private static SqlCommand CreateUpdateCom
mand(SqlConnection connection)
{
  SqlCommand cmd = new SqlCommand(
    "UPDATE Zipcodes SET City = @City WHERE
    Code = @Code", connection);
  cmd.Parameters.Add("@City", SqlDbType.VarChar, 30, "City");
  cmd.Parameters.Add("@Code", SqlDbType.VarChar, 4, "Code");
  return cmd;
}

private static SqlCommand CreateDeleteCom
mand(SqlConnection connection)
{
  SqlCommand cmd = new SqlCommand(
    "DELETE FROM Zipcodes WHERE Code = @Code", connection);
  cmd.Parameters.Add("@Code", SqlDbType.VarChar, 4, "Code");
  return cmd;
}
```

These are parameterized commands that are necessary in this case, but otherwise the methods are in principle simple. For example, if considers the definition of a single parameter:

```
cmd.Parameters.Add ("@Code", SqlDbType.VarChar, 4, "Code");
```

there is nothing mysterious in it, and the only thing to be aware of is the last parameter that is the column name in the database.

The three methods are in the test method used to associate update commands with the adapter. When the adapter executes the method *Update()*, these update commands are executed and the changes made in memory are transferred to the physical database. You should note that by writing update commands you, as a programmer, have full control over what happens when the database is updated.

The last method *Show()* called from the test method basically has nothing to do with the program, but the method prints all zip codes (starting with 78) on the screen using a *DataReader*:

```
private static void Show(SqlConnection connection)
{
  Console.WriteLine("****************************************");
  SqlCommand cmd = new SqlCommand(
    "SELECT * FROM Zipcodes WHERE Code LIKE '78%'", connection);
  connection.Open();
  SqlDataReader reader = cmd.ExecuteReader();
  while (reader.Read()) Console.WriteLine(reader["Code"]
  + " " + reader["City"]);
  connection.Close();
  Console.WriteLine("****************************************");
}
```

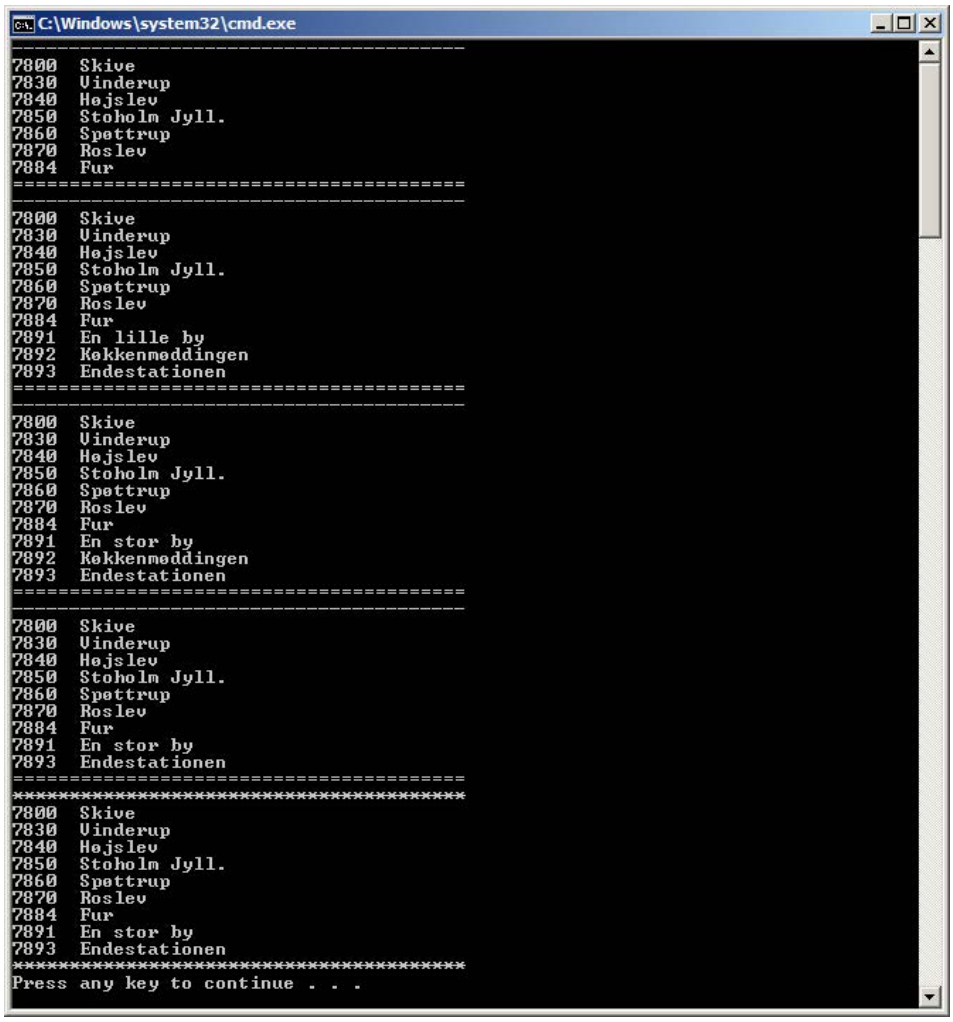
The goal is to show that changes are actually written back to the database, which you can then also convince by opening the database in SQL Server Management Studio (if you do not perform the last statement).

Immediately, the use of a data adapter is smart in situations where a program needs to modify the contents of a database, but there are two things to be aware of:

1. Writing update commands can be quite comprehensive - it was simple in this case, where there is also only one table and two columns. However, there is help from Visual Studio, and in many cases the development tool can generate the commands itself.
2. When an adapter opens the connection, extracts the data, closes the connection, later reopens the connection, writes data to the database and closes the connection again, a concurrency problem is created where data may in the meantime be modified by another program.

The last method is called *Remove()*. In principle, it has nothing to do with the example and does not use disconnected access types, but it removes the changes that the program has made to the physical database, just because the program can run multiple times.

Below is an example of a program run:



The project has another test method which performs exactly the same as the above, but after the adapter executes a call of the method *Fill()* the method *FillSchema()* is called, which retrieves constraints from the database and especially the primary key. If this method is performed, it is not necessary to explicitly define the primary key as in the previous example. Finally, there are the two statements:

```
SqlCommandBuilder builder = new SqlCommandBuileder();
builder.DataAdapter = adapter;
```

The result of these statements is that Visual Studio will automatically generate update commands for the adapter. So you are free to write methods yourself that can create these objects (one of the problems I mentioned in the previous example). In general, this function cannot be done, but in many situations it can.

6 THE ENTITY FRAMEWORK

The basic API using databases is the ADO as described above, which explain respectively connected access and disconnected ADO. There is nothing wrong with these APIs and they are by no means outdated, but there is an alternative that Microsoft calls *Entity Framework* or short EF.

EF can best be described as a layer on top of (disconnected) ADO, and the purpose is to hide the physical database details for the programmer. If you look at ADO, this API through a *DataSet* object provides a nice object-oriented approach to the database, but yet the link between the code and the physical database is strong, for example column names (or their order) can be found in the code in the form of SQL. It is therefore necessary that the programmer has knowledge of the structure of the physical database. EF, through a higher abstraction, tries to hide the basic database schema, including SQL, and the agent is

- LINQ
- a pile of auto generated code

Technically, one talks about an EDM (Entity Data Model), which is a conceptual model of the physical database, and to say a little differently, a number of classes that encapsulate the database. ADO uses provider dependent classes, and in exactly the same way, EF requires an update of these provider dependent classes. For SQL Server, these classes are in the assembly *System.Data.Entity.dll*, but there are also updated provider classes for Oracle and MySQL.

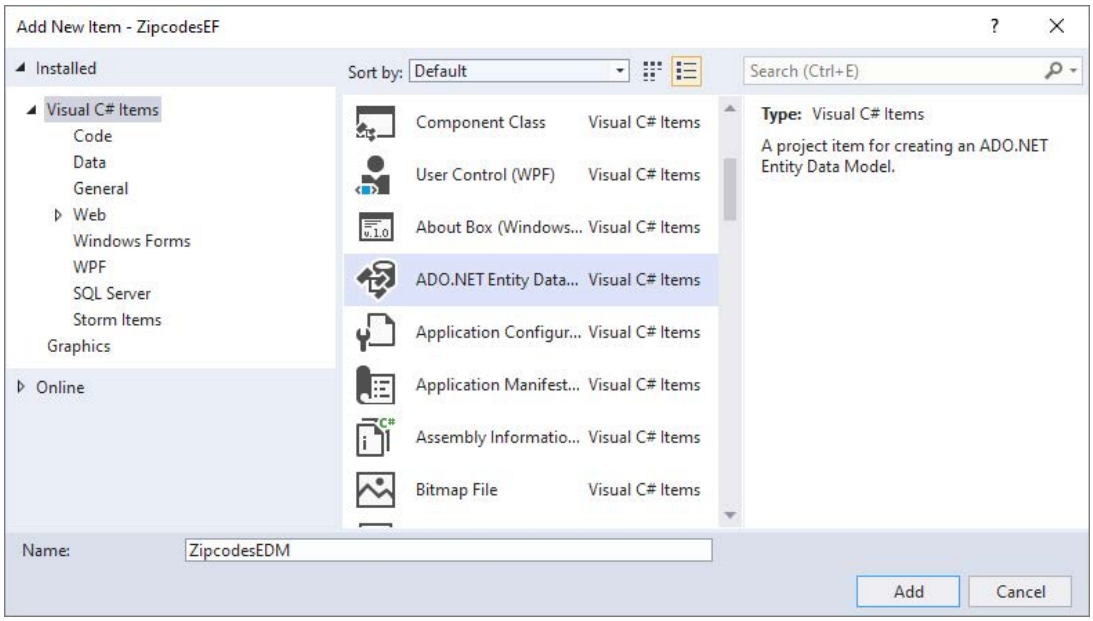
Once you become familiar with EF, the benefits are great, and you can actually write database programs without knowing anything about SQL. You do not have to write SQL statements and you do not see SQL anywhere, but you have to be able to understand LINQ (and for that the book C# 8). As always, there are drawbacks too. As point one, you must have created an *Entity Data Model* and this is done in Visual Studio using a wizard which automatically creates a pile of code using some new classes. A large part of that code is not seen at all, as it happens at compile time. All of which means that it is hard to see what Visual Studio is doing - and that's not the point.

6.1 USING EF FOR ZIP CODES

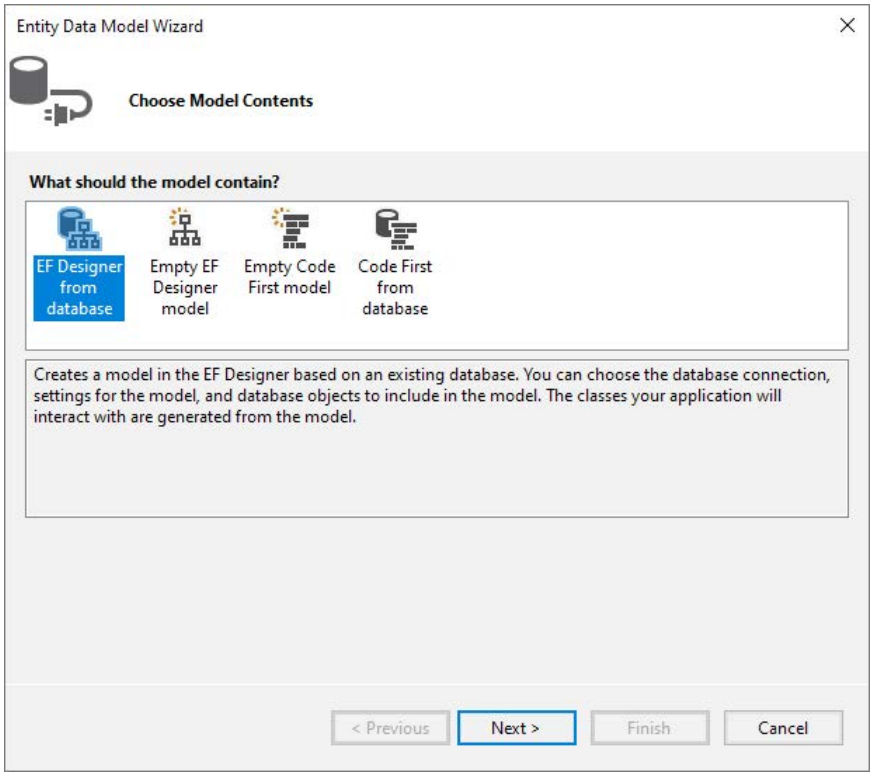
I want to write a program that performs some simple database operations on the table *Zipcodes* in the database *Contacts*. It's a console application called *ZipcodesEF*, and I want to focus primarily on what to do, but also a little bit on what it is that is auto generated and how it works. Specifically, the program should

1. Search the database and print the result on the screen
2. Add new rows to the table
3. Modify rows
4. Delete rows

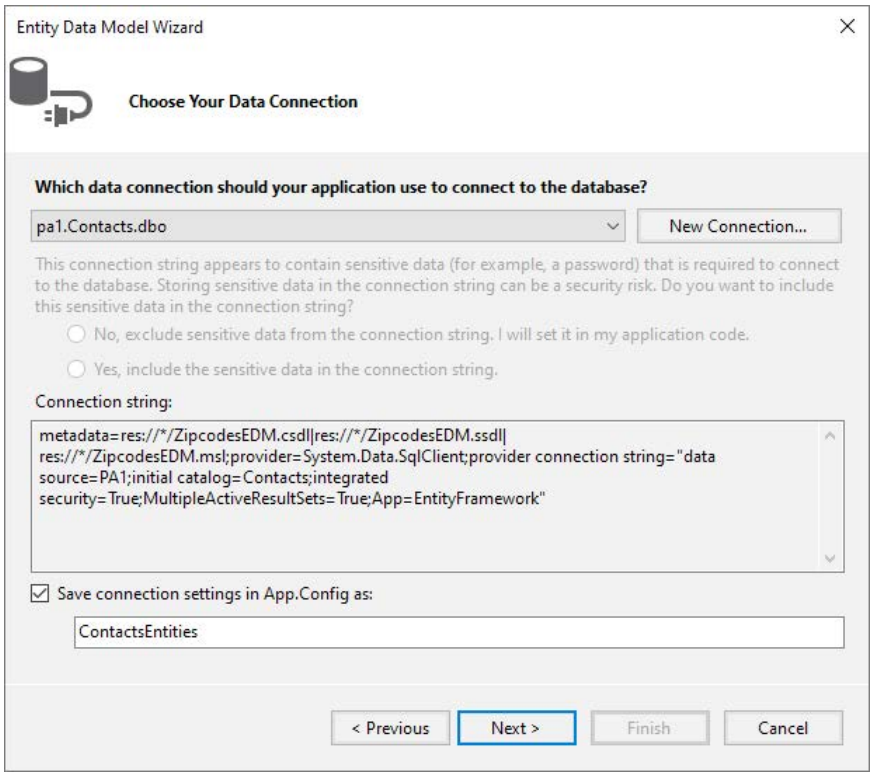
and the program thus shows how to perform the four basic database operations. I will start by creating a new Console Application project called *ZipcodesEF*, and next a connection must be defined between the program and the database, which in the form of classes that can map program objects to the database. For this is used an *edmx* file, which is an XML document that defines an *Entity Data Model*. I start by choosing *Add New Item*, and here I choose *ADO.NET Entity Data Model*, and finally I call the file *ZipcodesEDM*:



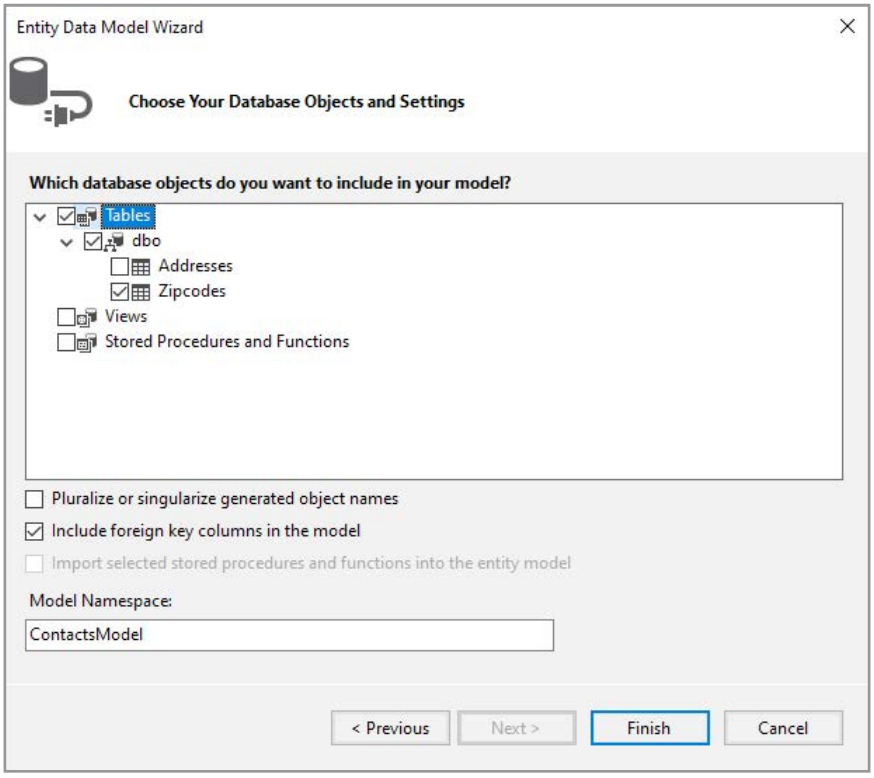
After clicking on *Add*, I get a window where you can choose how to create the model, and in this case I select (which is the default) *Generate from database*:



When I clicks *Next* I get a window to select a connection to the database. Maybe it's already there, but otherwise you click on *New Connection* and you can create a connection. Note that the connection string looks different than usual:



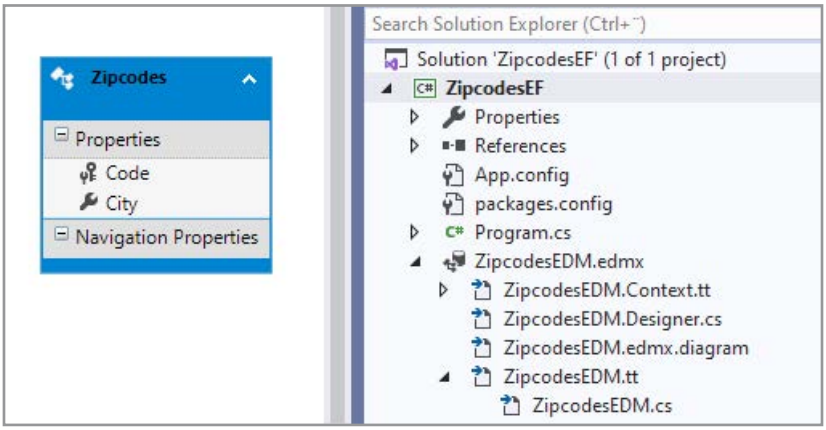
Also note the name *ContactsEntities*, which will represent the database in the program. The next step is to choose which table to use:



and finally you must click *Finish*. Visual Study has now done several things:

1. More references to multiple dll files have been set
2. A file called *ZipcodesEDM.edmx* and several other files has been added

The file *ZipcodesEDM.edmx* opens automatically in a design window:



and it represents in the form of a class a mapping of the current database table. In this, the class is called *Zipcodes* (the same name as the table) and the two properties have the same names as the corresponding columns. However, changing these names is allowed.

If you in Solution Explorer right-click *ZipcodesEDM.edmx* and select *Open With | XML (Text) Editor*, you get the XML that maps the program to the database.

If you open the class *Zipcodes* under *ZipcodesEDM.edmx*:

```
namespace ZipcodesEF
{
  using System;
  using System.Collections.Generic;

  public partial class Zipcodes
  {
    public string Code { get; set; }
    public string City { get; set; }
  }
}
```

you get a very simple class that simply defines the two properties. You should note that the names are the column names from the database. The class is defined partial, and if you want to extend the class, you should write the extensions in another file. In this case, I want to expand the class *Zipcodes* with a *ToString()* method, so I add a C# file, which I call *ZipcodesExt.cs*:

```
namespace ZipcodesEF
{
  partial class Zipcodes
  {
    public override string ToString()
    {
      return Code + " " + City;
    }
  }
}
```

The class *Zipcodes* represents a data object and as such is a model class. As mentioned, another model class is created called *ContactsEntities*, which represents the database:

```
namespace ZipcodesEF
{
  using System;
  using System.Data.Entity;
  using System.Data.Entity.Infrastructure;

  public partial class ContactsEntities : DbContext
  {
    public ContactsEntities() : base("name=ContactsEntities")
    {
    }

    protected override void OnModelCreating(DbModelBuilder
    modelBuilder)
    {
      throw new UnintentionalCodeFirstException();
    }

    public virtual DbSet<Zipcodes> Zipcodes { get; set; }
  }
}
```

and here you should especially notice that the class has a collection of *Zipcodes* objects called *Zipcodes*.

So far, I've done nothing but click the mouse to add an *Entity Data Model*, and then I've changed the *Zipcodes* model class and adding a *ToString()*.

Back there is the program itself, and it consists solely of the *Program* class with 5 static methods.

The first method searches the database for all objects where the zip code starts with a specific value:

```
static void Print(string code)
{
  using (ContactsEntities context = new ContactsEntities())
  {
    var q = context.Zipcodes.Where(z => z.Code.StartsWith(code));
    foreach (Zipcodes z in q) Console.WriteLine(z);
  }
}
```

First, an object representing the database is instantiated. This is equivalent to retrieving the contents of the database (here only the table *Zipcodes*) into an object of type *ContactsEntities*. Next, a LINQ expression is used to extract all objects that match the search criterion - in this case, the objects where the zip code that starts with the value being searched. The result is printed on the screen.

You should note that nowhere is direct reference to the database in the form of SQL and there is no reference to the table name, column names or the like. The method *Where()* is a method from LINQ, and although LINQ is not treated in this place, it is easy enough to understand the meaning.

The following method adds a new object to the table:

```
static void Add(string code, string name)
{
  using (ContactsEntities context = new ContactsEntities())
  {
    try
    {
      context.Zipcodes.Add(new Zipcodes { Code = code, City = name });
      Console.WriteLine(context.SaveChanges());
    }
    catch (Exception ex)
    {
      Console.WriteLine(ex.Message);
    }
  }
}
```

The method is easy to understand. It starts in the same way as in the method *Print()* and creates a database object - a *context*. Then an object is added to the *context* object's *Zipcodes* collection, and finally the changes are written back to the database with *SaveChanges()*. Here you can clearly see that there is an adapter under an *Entity Data Model* that takes care of updating the database.

The following is a similar method that updates a row in the database:

```
static void Update(string code, string name)
{
  using (ContactsEntities context = new ContactsEntities())
  {
    try
    {
      context.Zipcodes.Where(z => z.Code.
      Equals(code)).First().City = name;
      Console.WriteLine(context.SaveChanges());
    }
    catch (Exception ex)
    {
      Console.WriteLine(ex.Message);
    }
  }
}
```

There is not much to notice here either, and the procedure is the same as in the above methods. Note, however, how the object to be updated is found with a LINQ expression, after which the name is changed. Also note that the content is written back to the database.

Finally, there is a *Remove()* method, which deletes a row in the table:

```
static void Remove(string code)
{
  using (ContactsEntities context = new ContactsEntities())
  {
    try
    {
      Zipcodes zipcode = context.Zipcodes.Where(z
      => z.Code.Equals(code)).First();
      context.Zipcodes.Remove(zipcode);
      Console.WriteLine(context.SaveChanges());
    }
    catch (Exception ex)
    {
      Console.WriteLine(ex.Message);
    }
  }
}
```

Here you should note that you first find the object with a LINQ expression, and then the context object's *Zipcodes* collection has a *Remove()* method.

The *Main()* method is:

```
static void Main(string[] args)
{
  Print("78");
  Add("7892", "Voldsted");
  Add("7895", "Ulvkæret");
  Update("7892", "Voldgravene");
  Print("78");
  Remove("7892");
  Remove("7895");
  Print("78");
}
```

The program performs the classic database operations, but everywhere you work with model objects and without the programmer having to know the structure of the database or SQL. EF thus provides a completely object-oriented approach to the database, and that is the whole idea behind the *Entity Framework*.

The idea behind EM is often referred to in the literature as ORM (Object-Relational Mapping).

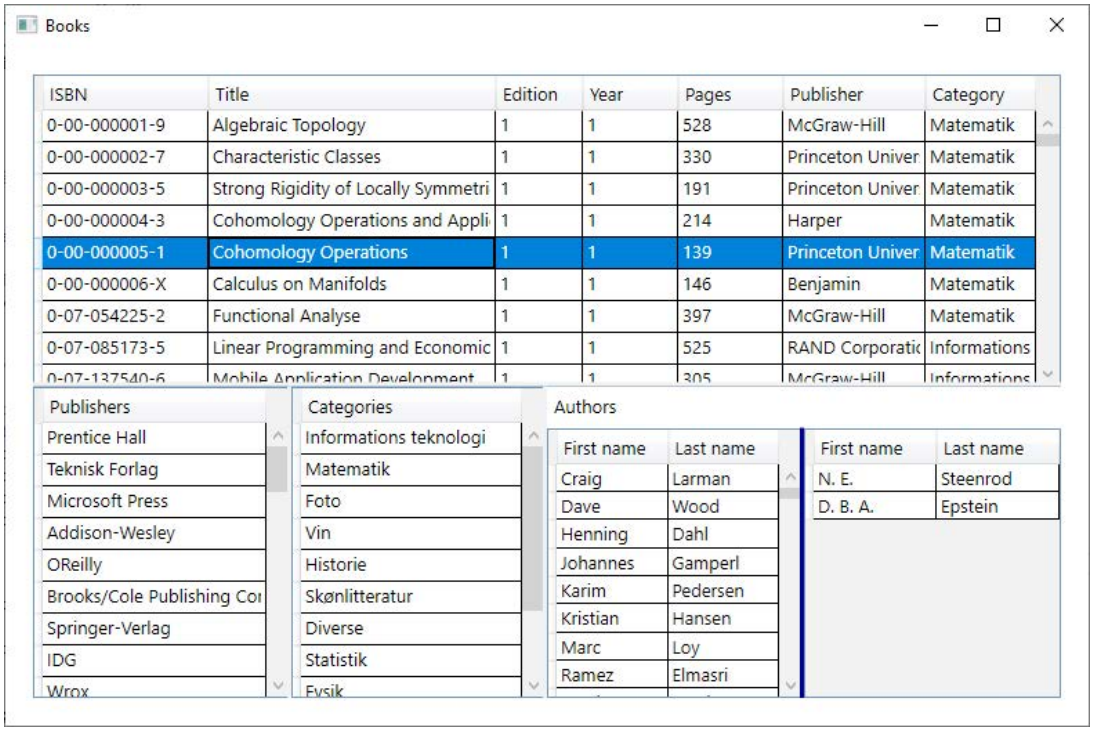
6.2 THE BOOKS DATABASE

In this section I will show how to use an Entity Data Model for the *Books* database. I have written a program which opens the window below, that is a window with 5 *DataGrid* component where

1. The upper grid shows an overview over all books.
2. The left lower grid all publishers.
3. The next all categories.
4. The next all authors.
5. The last the authors for the selected book.

All the *DataGrid* components are separated with *GridSplitter* components.

I do not want to display the XML code for the window which adds nothing new and is based on data binding. Instead, I want to focus on how the program is made and how Visual Studio has generated most of it.



The program is based on an entity data model for the *Books* database. This time, the model requires little explanation. The database has 5 table

- *Book* to books
- *Publisher* to publishers
- *Category* to categories
- *Author* to authors
- *Written* as a many-to-many relation between *Book* and *Author*

As the same way as in the above example I have added an *ADO Entety Data Model* to the project called *BooksEDM*. It requires to make a connection to the database (or more precisely let Visual Studio do it) and I have to choose which tables to use, and it should be all 5.

If you look at the entity model, then only four model classes are created, and thus one model for each of the first four of the above tables, but there is no model class for the relationship table *Written*. It has to do with how the entity model implements foreign keys.

In the database, the *Book* table has a foreign key to the *Category* table, corresponding to the book belonging to a specific category. The class *Category* has properties for the two columns in the *Category* table as well as a collection of *Book* objects

```
public partial class Category
{
  public Category()
  {
    this.Book = new HashSet<Book>();
  }

  public int Category_id { get; set; }
  public string Category_name { get; set; }

  public virtual ICollection<Book> Book { get; set; }
}
```

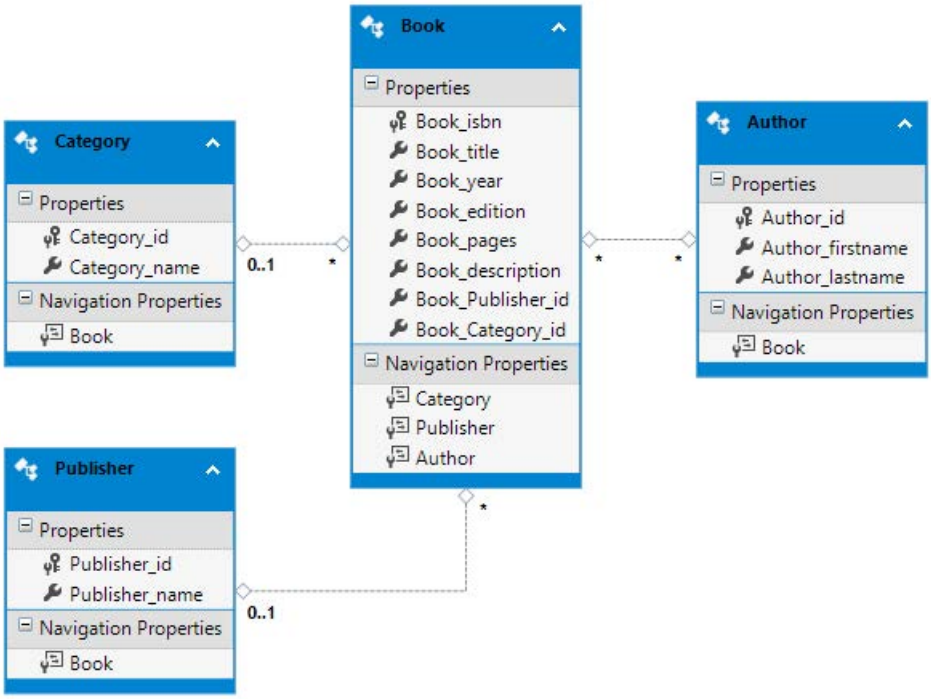
As then the class has a reference to all books that has a foreign key to this category. If you look at the class *Publisher* you will observe the same and a *Publisher* object has a reference to all *Book* objects published by that publisher.

If you then look at the *Book* class, it has properties for all columns in the database table, but for the foreign keys for the tables *Publisher* and *Category* it is a reference to corresponding objects. The class also has a collection with *Author* object and the references to the books authors, and this collection represents one of the columns in the table *Written*:

```
public partial class Book
{
  public Book()
  {
    this.Author = new HashSet<Author>();
  }

  public string Book_isbn { get; set; }
  public string Book_title { get; set; }
  public Nullable<int> Book_year { get; set; }
  public Nullable<int> Book_edition { get; set; }
  public Nullable<int> Book_pages { get; set; }
  public string Book_description { get; set; }
  public Nullable<int> Book_Publisher_id { get; set; }
  public Nullable<int> Book_Category_id { get; set; }

  public virtual Category Category { get; set; }
  public virtual Publisher Publisher { get; set; }
  public virtual ICollection<Author> Author { get; set; }
}
```

The other side of the table *Written* is represented in the class *Author*:

```
public partial class Author
{
  public Author()
  {
    this.Book = new HashSet<Book>();
  }

  public int Author_id { get; set; }
  public string Author_firstname { get; set; }
  public string Author_lastname { get; set; }

  public virtual ICollection<Book> Book { get; set; }
}
```



With these classes it is simple to write the code behind:

```
public partial class MainWindow : Window
{
  private BooksEntities entities = new BooksEntities();
  private Book book;

  public MainWindow()
  {
    InitializeComponent();
    DataContext = this;
    isbGrid.ItemsSource = new ObservableCol
    lection<Book>(entities.Book);
    pubGrid.ItemsSource = new ObservableCollect
    ion<Publisher>(entities.Publisher);
    catGrid.ItemsSource = new ObservableCollect
    ion<Category>(entities.Category);
    autGrid.ItemsSource = new ObservableColle
    ction<Author>(entities.Author);
  }

  public Book SelectedBook
  {
    get { return book; }
    set
    {
      book = value;
      wriGrid.ItemsSource = book.Author;
    }
  }
}
```

The variable *entities* refers to the database model (the entity model). The variable *book* refer to the book that is selected. In the constructor, the model's collections are linked to their own *DataGrid* in the user interface.

Back there is a property *SelectedBook*, which in XML is bound to *SelectedItem* for the first *DataGrid*. The result is that when a book is selected in the user interface, the book object is assigned a reference to that book, after which the last grid in the user interface is bound to a collection of the book's authors.

6.3 MAINTAINING CONTACTS A LAST TIME

I have made a copy of the project *ShowPersons3* and called the copy *ShowPersons4*. The program must work in exactly the same way and there should be only one difference where the class *DB* is removed and where the program uses an Entity Data Model instead.

The changes include:

1. An entity data model called *ContactsEDM* is added, when the model is created for both tables.
2. The type of the property *Zipcode* in the class *Person* is changed to a *string*.
3. Two methods are added to the class *Person* which respectively returns a new *Addresses* object initialized by the values of the *Person* object and initialize an existing *Addresses* object with the values of the *Person* object.
4. In the code behind for *MainWindow* a *ContactsEntities* object is created and is the object representing the database, and all *Addresses* objects in the *ObservableCollection* are wrapped in a *Person* object.
5. The event handles which updates the database are changed slightly so that they apply instead the entity data model.
6. The classes *DB* and *Zipcode* are remove.

Then it all after a few adjustments works again. Actually, you could completely do without the class *Person* and directly bind the UI to *Addresses* objects, but due to the *CollectionViewSource* object and the filter there are problems getting the UI updated, so the class is maintained.

7 FINAL EXAMPLE: BOOKS

As a final example, I will write a program to maintain the books database and then a program that is a typical database application. The purpose of the program is to show the development of a program as a MVVM program and using an Entity Data Model. I want to focus primarily on the process and at least give an example of what you can do, but there are definitely other ways, and the current solution is not the one where you have to write the least code yourself.

The development will include the following iterations which means that the program will be developed through a series of small steps:

1. Task formulation
2. The database
3. The Visual Studio project and developing a prototype
4. The Entity Data Model
5. The ViewModel layer
6. Developing the MainWindow
7. Maintenance of publishers, categories and authors
8. Maintenance of books
9. Search
10. The last thing

7.1 THE TASK-FORMULATION

The database contains information about a book collection and consists of information about books, publishers, categories and authors. The program must be used to search / find information about these entities and to maintain this information. As so the program's functions are

1. a main window which shows an overview and searches for books
2. a window for maintaining books
3. a window for maintaining publishers
4. a window for maintaining categories
5. a window for maintaining authors

The task is very precisely formulated and therefore no actual analysis is required. The only uncertainty is related to the user interface which is clarified in connection with the prototype.

7.2 THE DATABASE

The database is described at the beginning of this book, but due to several shortcomings and inconveniences, it has been decided to redesign the database similar to the following script, as shown here in its full length as it simultaneously serves as documentation of the database:

```
create table Publisher
(
  Pub_Id INT NOT NULL IDENTITY(1,1) PRIMARY KEY,
  Pub_Name NVARCHAR(100) NOT NULL,
  Pub_Text TEXT
);

create table Category
(
  Cat_Id INT NOT NULL IDENTITY(1,1) PRIMARY KEY,
  Cat_Name NVARCHAR(100) NOT NULL,
  Cat_Text TEXT
);

create table Author
(
  Aut_Id INT NOT NULL IDENTITY(1,1) PRIMARY KEY,
  Aut_Fname NVARCHAR(100),
  Aut_Lname NVARCHAR(50) NOT NULL,
  Aut_Text TEXT
);

create table Book
(
  Book_Id INT NOT NULL IDENTITY(1,1) PRIMARY KEY,
  Book_Isbn10 NVARCHAR(13),
  Book_Isbn13 NVARCHAR(17),
  Book_Title NVARCHAR(255) NOT NULL,
  Book_Year int,
  Book_Edit int,
  Book_Pages int,
  Book_Count int default 1,
  Book_Text TEXT,
  Book_Pub int,
  Book_Cat int,
  foreign key (Book_Pub) references Publisher ON DELETE SET NULL,
  foreign key (Book_Cat) references Category ON DELETE SET NULL
);
```

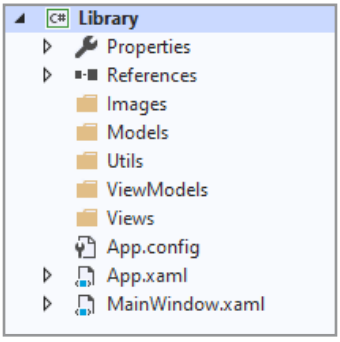
```
create table Written
(
  Written_Book int not null,
  Written_Aut int not null,
  Primary Key (Written_Book, Written_Aut),
  foreign key (Written_Book) references Book ON DELETE CASCADE,
  foreign key (Written_Aut) references Author ON DELETE CASCADE
);
```

The script is called *CreateBook.sql*. When the script is executed in Microsoft SQL Server Management Studio the result is an empty database, and to help the development of the program is written a program called *InitBooks*, which copy the old database to the new. I will not show this program here, but the program is not quite simple as the database architecture has changed.

You should note that in the context of a development project, it is quite common to write that kind of utilities that are not part of the actual task, but instead are programs that need to test something or as here to convert data or maybe something else entirely.

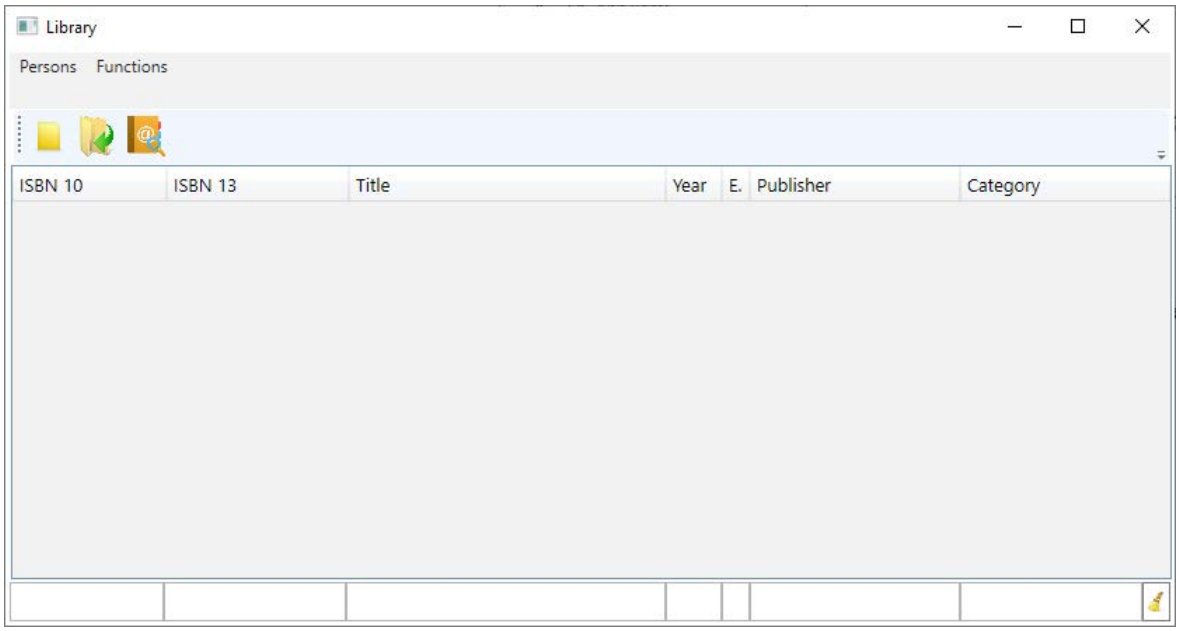
7.3 A PROTOTYPE

The project is a WPF application called *Library* and is created for a MVVM architecture:



It's a classic architecture. For the time being I do not know what the folder *Utils* should be used for, but during the development of a program there will often be classes that do not naturally belong in other project folders, and for example, there may also be general classes that can be used in other contexts.

When the program is executed it should open the following window which must show an overview over the book in the database:

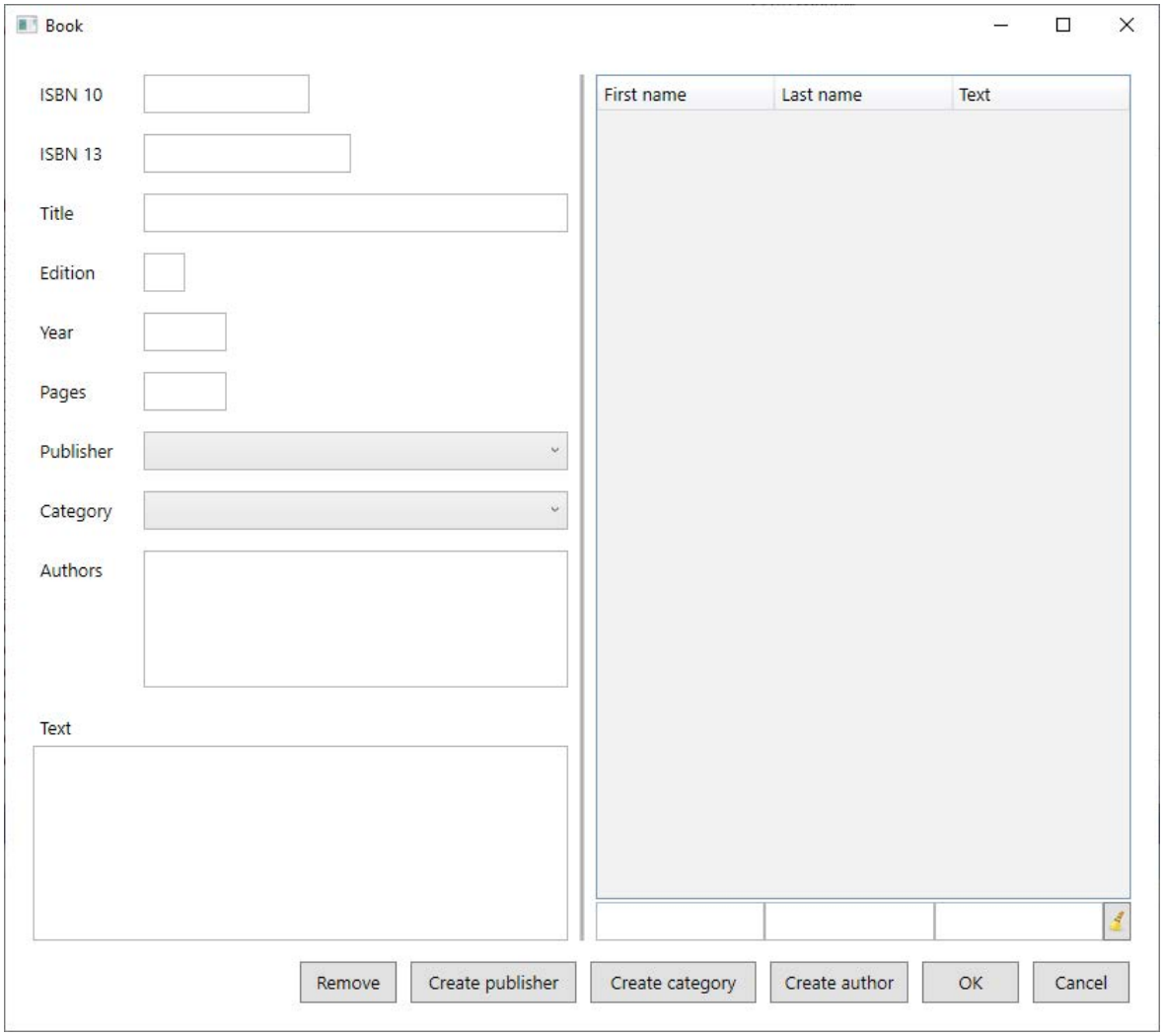


The window has a *DataGrid* with columns for book properties and the window has filters for these properties. The three buttons in the toolbar should be used to

1. open a dialog box to create a new book
2. open the same dialog box to edit the selected book
3. open a dialog box to enter search criteria

These functions are also found in the menu and here are also functions which opens dialog boxes to maintain publishers, categories and authors.

The dialog box to maintain books (if you in the prototype clicks on the first button) opens the following window:

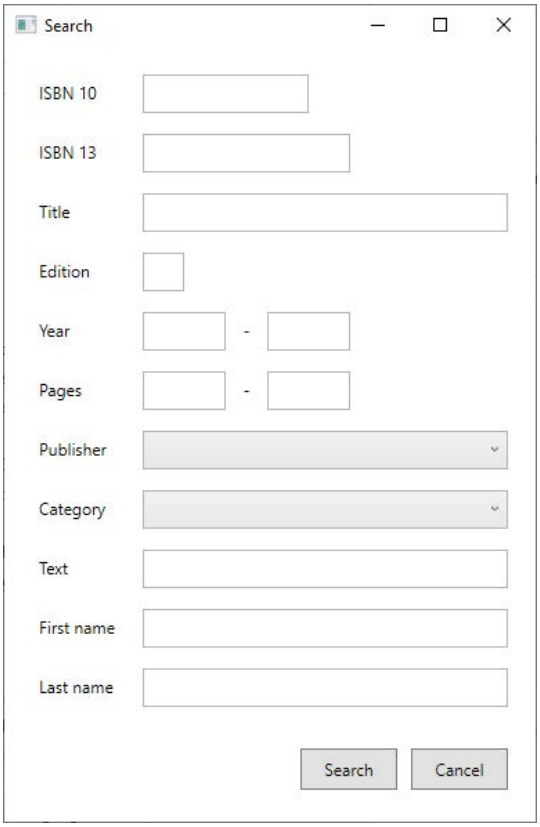


It is a complex window where the left side has fields to enter / select values for a book, while the right side has a *DataGrid* with a filter, which shows all authors. If the user double click on an author the author should be added to the list box for book's authors indicating that this author is an author for the book. If the user double click on an author in the list box, this author is removed from the books authors.

The window has buttons to create a publisher, a category and an author. This means that you can create one of these entities when you are creating a new book, and it is especially important to be able to create a new author. If you creates a new entity of one of these three objects the object should be selected for the book.

If you click on the search button in the toolbar the program opens the window below where to enter search criteria, and when you click *Search* the main window should be updated with the books which matches the criteria. The criteria for isbn, title, edition, publisher and category works the same way as the filters, while you for year and pages can enter a from

and to value. You can also search in the description (all books which contains the search text in the book's description matches), and you can search for author where all books where whose authors' first name and last name contains the search values match the search criteria.

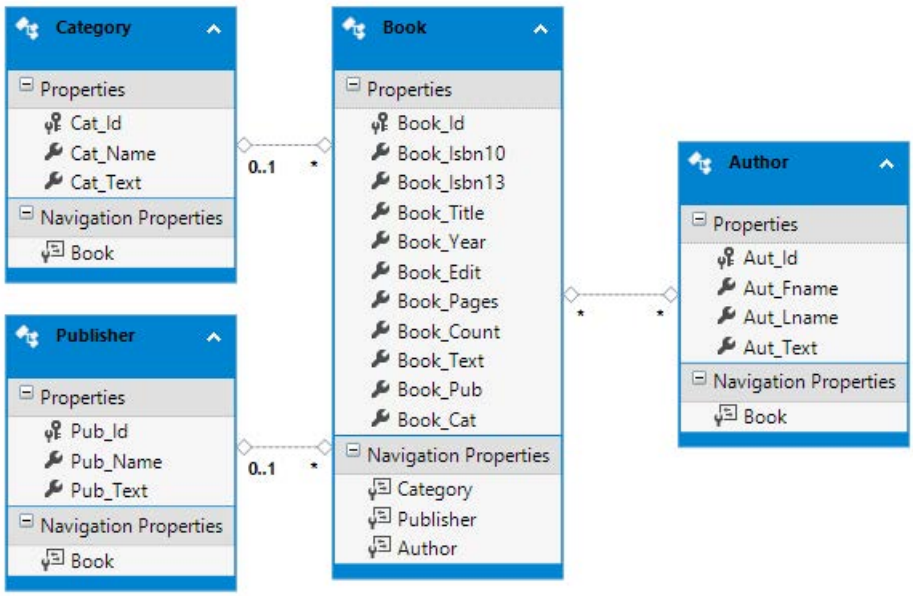


If you selects books from the search criteria it is equivalent to having selected a subset of all books.

7.4 THE ENTITY DATA MODEL

This is a short iteration as Visual Studio does it all. As start of a new iterations I creates a copy of the project from the previous iteration, and that way you can always go back and continue the development from the last stable version. The copy is called *Library1*, and it is the result of the program after the completion of iteration 1 and thus after the prototype is completed.

The only thing to do in this iteration is to add an entity data model for the database *Library*. The model is called *LibraryEDM* and is added to the *Models* folder as the classes in the entity model is the projects model layer:



and the model also contains the class *LibraryEntities* which represents the database.

When the project is compiled this iteration is completed, and you can still run the prototype.

7.5 THE VIEWMODEL LAYER

Also, this iteration starts with a copy of the project called *Library2*.

In this iteration I will start programming the *ViewModel* layer. The layer will usually contain of two classes for each window, one that is a wrapper for model layer for the purpose of adapting the model class for data binding, while the other establishes the data binding and contains commands for the user interface. In this iteration, I will alone look at the wrapper classes. In fact, this classes can be completely avoided, as there are auxiliary tools that automatic wrapper the model, but I want to keep the class partly because it makes it easier to see what is happening and partly because the class is quite simple even if some code has to be written, but it is the same thing that must happen every time.

The model (created be the Entity Framework) has four model classes, one for each of the four entity tables in the database. For each of these classes I must write a wrapper. A wrapper class has the form:

```
public class PublisherModel : ViewModelBase, IDataErrorInfo
```

where *ViewModelBase* is the base class for a *ViewModel* class from the project *HistoryPeople*. The purpose of the wrapper is then to send a notification to the user interface when the value of a property is changed, but also to implements validation of a model object. The first is simple and and happens in the same way for all classes but the validation can be more comprehensive. It is the case for wrapper for the class *Book* when the class must validate where an ISBN is correct. The folder *ViewModels* is then expanded with five classes:

1. *ViewModelBase* from the project *HistoryPeople*
2. *PublisherModel* that is a wrapper for the model class *Publisher*
3. *CategoryModel* that is a wrapper for the model class *Category*
4. *AuthorModel* that is a wrapper for the model class *Author*
5. *BookModel* that is a wrapper for the model class *Book*

There are also added two types to folder *Utils* where the first is a simple enumeration to indicate which operation is performed on an entity:

```
public enum ModelChanged { UPDATE, ADD, REMOVE }
```

while the other is a generic *EventArgs* type from the project *HistoryPeople*. The wrapper classes defines a wrapper for each property and the class also defines methods for add a new object, update an existing object and remove an object. These three methods also updates the database (the entity method *SaveChanges()*) and raises an event telling observers which operation is performed.

7.6 THE MAINWINDOW

As the next step I will implement the main window, and the copy of the project from the previous iteration is called *Library3*. The design is ready from the prototype and to implement the window I must:

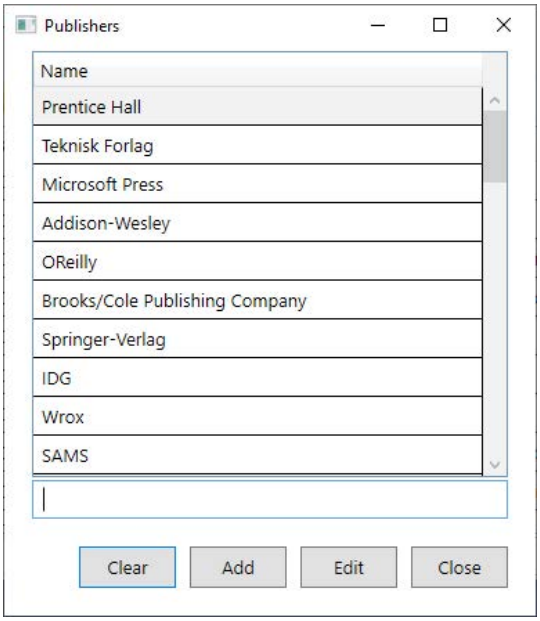
1. Copy the class *RelayCommand* from the project *HistoryPeople* to the folder *ViewModels*.
2. Add a class called *ViewModelMain* to the folder *ViewModels*. The class must inherits the class *ViewModelBase*.

3. Create an instance of *LibraryEntities* in *ViewModelMain*.
4. Add an *ObservableCollection<BookModel>* to *ViewModelMain* and initialize this collection with book entities and the wrap the collection in a *CollectionViewSource* (to create the filter). Add a property for the view source.
5. Add properties for the filter and implements the filter.
6. Add a command to clear the filter.
7. Add two commands to open the two dialog boxes from the prototype and the remove the two event handlers in code behind.
8. In code behind for *MainWindow* create an instance of *ViewModelMain* and bind this instance to the window's *DataContext*.
9. Define all the binding in XML and remove the *Click* events in the definition of the two buttons in the toolbar.

Then the *MainWindow* should be initialized so it shows all the books and the filter should be ready.

7.7 PUBLISHERS, CATEGORIES AND AUTHORS

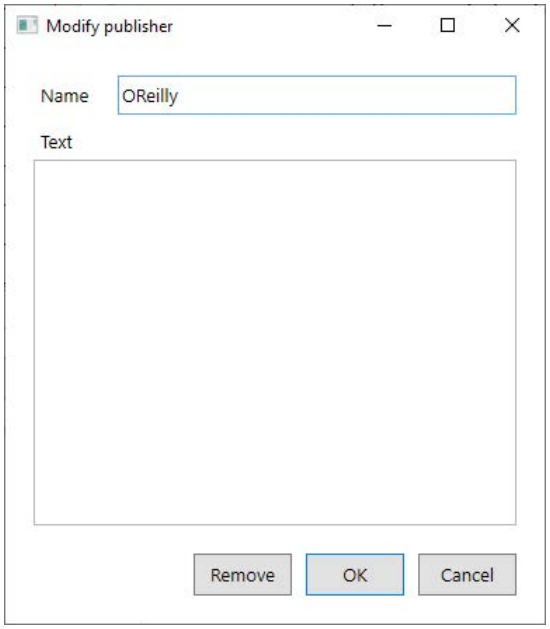
The copy of the project from the previous iteration is called *Library4*. In this iteration the task is to implement the three menu function for maintenance of publishers, categories and authors. The three functions are basically identical and I will only explain the maintenance of publishers. If the user selects the function from the menu the program opens the following dialog box:



The dialog box has *DataGrid* with a filter showing all publishers and four buttons:

1. *Clear*, clear the filter
2. *Add*, opens a dialog box to create a new publisher
3. *Edit*, opens the same dialog box for a selected publisher, and the user can edit the publisher
4. *Close*, close the dialog box

If you for example selects *OReily* and click on *Edit* the program opens the following window:



To implement this functionality there are added two classes to the *ViewModels* layer and two classes to the *Views* layer:

1. *ViewModelPublisher* which is a view model for the last dialog box. The class has a *PublisherModel* object and the purpose of the class is to publish this object so that the object's properties can be bound in XML as well as implement commands for the three buttons.
2. *PublisherWindow* which is the class defining the last dialog box. The most important thing is the data binding in XML, and in code behind the class creates a *ViewModelPublisher* object and binds this object to the user interface.
3. *ViewModelPublishers* which is a view model for the first dialog box. The class has an *ObservableCollection* of *PublisherModel* objects and the purpose of the class is to publish this object so it can be bound in XML as well as implement commands for the four buttons. It is also this class that implements the filter.

4. *PublishersWindow* which is the class defining the first dialog box. The most important thing is the data binding in XML and especially the binding of the *DataGrid* component, and in code behind the class creates a *ViewModelPublishers* object and binds this object to the user interface.

You should note that the data maintained by these classes is represented of the class *PublisherModel* and then as the wrapper of the entity class *Publisher*.

To implement maintenance of publishers is added two classes to *ViewModels* and two dialog boxes to *Views*. The same must be done for categories and authors but it is all most copy / paste.

7.8 MAINTENANCE OF BOOKS

Again I start this iteration with creating a copy of the project called *Library5*. The next iteration implements maintenance of books and is the program most complex function. The user interface is developed in the prototype and the wrapper for the data model is also ready and so the only thing to do is to write a *ViewModel* class called *ViewModelBook*.

Basically, the class is identical to the other classes for maintaining a database table, but the class is more comprehensive because one should be able to call the functions to create a publisher, a category and an author respectively. There is also another challenge as it must be possible to double-click authors in the *ListBox* and in the *DataGrid*, and since you cannot immediately implement double clicks as Commands, these functions are implemented as usual event handlers in code behind.

7.9 THE SEARCH FUNCTION

The project copy from this iteration is called *Library6*. The window to enter search criteria is defined in the prototype and the iteration includes the following:

1. Add a class *SearchModel* to the *ViewModels* layer which represent the search criteria. The class is simple and has no data validation.
2. Add a class *ViewModelSearch* to the *ViewModel* layer that is a model for *SearchWindow*. The class must implements the commands for the dialog box and the search command is not quite as simple as the method used to extract the books of the entity model that match the search criteria.

3. Implement data binding in XML.
4. Update *MainWindow* such the window shows the result of a search. The toolbar is expanded with a new button to clear a search.

7.10 THE LAST THINGS

Now, the program is finished in principle. Although there still is not talking about a large program, it consists nevertheless of some types in the form of interfaces and classes. No matter how careful you have been in the program development, there is a large chance that some of the code is improper, that there is code repetitions and something that could be written in a better way. It is therefore worthwhile to complete the development of a program with a code review, where you also should comment the code (to the extent it is not already done) and remove inconveniences (and of course also direct errors). It is a big work, not the world's most interesting work, but the work is certainly well spent. Especially should you pay attention to variables or methods that are not used, and that kind of items should be removed.

With regard to code repetition, they should be within reasonable limits removed because it may be important for the maintenance going forward, but you have to balance the benefit and one should not remove code repetitions, if the price is that you end up with a code, which no other can figure out. A code that is robust and easy to understand is one of the most important quality parameters, and you must be very careful, that code optimization is not at the expense of safe and readable code.

A code review will also often result in awareness of new features and minor enhancements that for some reason are not included, and such features may need to be implemented.

The code review starts, like the other iterations to create a copy of the project, and I created a copy named *Library7*.

In this case the following functions are added to the program:

1. When you create a new category the program should in the same way as for authors check where there already is a category with the same name and if so the program must show a warning.
2. The same function is implemented for publishers.
3. When you maintains authors the program shows a *DataGrid* showing all authors. The grid has only columns for first name and last name and the grid is expanded with a column for text.

4. The main window is extended with a status line that shows the number of books which from start is all books and else the number of books for the current search.
5. Commands are added to all menu items as some of the menu items has not been given a function in the first iterations.

Another extension has been added. The program is fine to maintain a private book collection as long as these are the usual paper books, but today there are several options with different electronic formats. It should could therefore also be possible to specify the book's media format.

However, such a change is more difficult because the database is not prepared for it (a column is missing). The problem could be solved by adding an extra column to the database, but it is not necessarily simple and can lead to adverse side effects, especially if there are other programs, that use the database. In this case, however, the program has a column that is not used (the *Book_Count* column in the table *Book*), so I will solve the problem by using it. However, you should note that here are the same options for side effects and at least if the database is used by other programs. The solution is somewhat of a work around, but you should note that such work arounds are not unusual in practice but always associated with a considerable risk.

To solve the problem the following must be done when it also should be possible to search the new field:

1. An enumeration is added to *ViewModels*.
2. The class *BookModel* is expanded with a new property.
3. *BookWindow* is expanded with a combo box to select the format type.
4. *ViewModelBook* is expanded with a property for an *ObservableCollection* with format types.
5. *SearchModel* is expanded with a new property.
6. *SearchWindow* is expanded with a combo box to select the format type.
7. *ViewModelSearch* is expanded with a property for an *ObservableCollection* with format types.

APPENDIX A: SQL

The following is a short introduction to SQL, and the purpose is to show the syntax for the most important statements. Now SQL is not just SQL, and the different database vendors have their variants of the language, and although the following examples are tested against a SQL Server database, they will practically all could be used by any database server. Thus, there will be a number of extensions to SQL, which is not dealt with, and the following should not be perceived as a complete manual for SQL.

Before addressing the actual SQL language, I will mention a few basic database concepts.

A database is a collection of related tables, and a table contains data organized into columns and rows. As an example is shown a part of a database table for books, where there are seven columns:

	Book_isbn	Book_title	Book_year	Book_edition	Book_pages	Book_Publisher_id	Book_Category_id
1	0-00-000001-9	Algebraic Topology	1966	1	528	486	165
2	0-00-000002-7	Characteristic Classes	1974	1	330	487	165
3	0-00-000003-5	Strong Rigidity of Locally Symmetric Spaces	1973	1	191	487	165
4	0-00-000004-3	Cohomology Operations and Applications in Homoto...	1968	1	214	488	165
5	0-00-000005-1	Cohomology Operations	1962	1	139	487	165
6	0-00-000006-X	Calculus on Manifolds	1965	1	146	489	165
7	0-07-054225-2	Functional Analyse	1973	1	397	486	165
8	0-07-085173-5	Linear Programming and Economic Analysis	1964	1	525	472	164
9	0-07-137540-6	Mobile Application Development	2002	1	305	486	164
10	0-07-222897-0	C++ from the Ground Up	2003	3	602	490	164

Each column has a name, and the name should be unique within each table. Each column has a data type that determines which data the column can contain. A row contains data about a particular book, and the row's value in a column is called a field and must match the column's data type. A row is also called for a record. The value of a particular field can be *null*, which simply means that the field has no value, and it is important to note that it is not the same as 0 or blank.

A column can be assigned constraints that one can think of as conditions, the values in that column must meet. If they do not, the database management system is rejecting the database operation. The most important constraints are:

- *NOT NULL*, which indicates that a column can not contain null values, and the row thus must have a value in that column.
- *DEFAULT*, where you can specify a default value that is used, if there is not given a value for the column.
- *UNIQUE*, which means that the column's values must be unique.

- *PRIMARY KEY*, indicating that all rows in this column should have a unique value that identifies the record.
- *FOREIGN KEY*, which defines a reference to a primary key in another table (possibly the same table). The column's value must be the value of a primary key in the other table or *null*.
- *CHECK*, where it is possible to define that the column's values must satisfy a condition.
- *INDEX*, defines an index (table of content) for the column's values.

The database management system is the family of programs that maintains the database's databases, and is a daemon that constantly runs in the background and waiting for requests from users in form of SQL commands. In addition to performing these commands, it is also the database management system that validates that the commands can be executed properly and in this context tests whether the above constraints are respected. The database management system has to ensure the integrity of the database, and you talk about the following integrity rules:

- *Entity integrity*, which mean that no rows must have a NULL value in the primary key column and all values in this column are different.
- *Domain integrity*, which ensures that the values in a column are in accordance with the type of the definition and the constraints.
- *Referential integrity*, which guarantees that a row cannot be deleted if the database contains a different row (typically in another table), which refers to this row.
- *User defined integrity*, where it is possible to define special rules that do not fall under the three above rules.

These rules help to ensure that data in the database always contains legal values, but they do not ensure efficiency, which is a matter of how the database is designed and created. There are several guidelines for good database design, and one of them is called normalization. Very briefly, it is a variety of conditions a database design must meet. Usually one uses only the first three (but there are several), and speaking about that the database is in third normal form. I will not mention normalization of databases here, but refer to books or articles on database theory or the next book in this series.

THE BOOK DATABASE

In order to show specific SQL commands I needs a database to be able to perform them. I want to use the database *Books* as defined in the introduction chapter which is created as:

```
create table Publisher
(
  Publisher_id                int identity(1,1),
  Publisher_name              nvarchar(100) not null,
  primary key (Publisher_id)
);

create table Author
(
  Author_id                   int identity(1,1),
  Author_firstname            nvarchar(100),
  Author_lastname             nvarchar(50) not null,
  primary key (Author_id)
);

create table Category
(
  Category_id                  int identity (1,1),
  Category_name                nvarchar(100) not null,
  primary key (Category_id)
);

create table Book
(
  Book_isbn                   nvarchar(20),
  Book_title                  nvarchar(100) not null,
  Book_year                   int,
  Book_edition                int,
  Book_pages                  int,
  Book_description            ntext,
  Book_Publisher_id           int,
  Book_Category_id            int,
  primary key (Book_isbn),
  foreign key (Book_Publisher_id) references Publisher,
  foreign key (Book_Category_id) references Category
);

create table Written
(
  Written_isbn                 nvarchar(20),
  Written_Author_id            int,
  primary key (Written_isbn, Written_Author_id),
  foreign key (Written_isbn) references Book,
  foreign key (Written_Author_id) references Author
);
```

SQL DATA TYPES

Each column in a database table has a data type. There are the following options:

- *Int*, that can contain an integer between -2147483648 and 2147483647.
- *Smallint*, that can contain an integer between -32768 and 32767.
- *Tinyint*, that can contain an integer between 0 and 255.
- *Bit*, there is 0 or 1.
- *Bigint*, that can contain an integer between the two integers -9223372036854775808 and 9223372036854775807.
- *Numeric* (or *Decimal*) to decimal numbers between -10^38 +1 and 10^38 -1
- *Float* to floating points from -3.40E + 38 to 3.40E + 38 (the type is also called *Real*).
- *Datetime* to timekeeping from 1. January 1753 to 31 December 9999.
- *Smalldatetime* to timekeeping from 1. January 1900 to 6. June 2079.
- *Date* to dates on the form YYYYMMDD
- *Time* to time on the form HHMMSS
- *NChar* to text of a fixed length with max 4000 unicode characters, where the field is filled with blanks.
- *NVarchar* to text of variable length, which can be up to 4000 unicode characters.

In addition, there are two types *NText* and *Image*, which are used respectively to store arbitrary text and binary data.

In addition to data types, there are similar to other programming languages operators, and in terms to arithmetical operators and operators for comparison the syntax is almost the same as in C#. However, there are some other operators, as I will explain in connection with examples, but the list is as follows:

- ALL
- AND
- ANY
- BETWEEN
- EXISTS
- IN
- LIKE
- NOT
- OR
- IS NULL
- UNIQUE

SQL COMMANDS

SQL is a command language, and there are the following commands:

- CREATE, that is used to create database objects such as tables
- ALTER, that is used to modify existing objects, such as tables
- DROP, that is used to delete objects such as tables

These commands are called DDL commands for Data Definition Language.

- GRANT, that is used to assign the rights for users
- REVOKE, that is used to remove rights from users

These commands are called DCL commands for Data Control Language.

- INSERT, which is used to insert a row in a table
- UPDATE, which is used to modify the content of one or more rows in a table
- DELETE, which is used to delete one or more rows in a table

These commands are called DML commands for Data Manipulation Language.

- SELECT, which is used for extracting rows from one or more tables

This command is called a DQL command for Data Query Language.

There are as such 9 commands so SQL should be easy to learn, and it also is, and it is only the last, which is complex with a very comprehensive syntax.

Note first that SQL does not differentiate between small and capital letters, but in the following, I generally write SQL names with capital letters and it is only because it for the reader should be clear what words are SQL keywords.

As an example is shown a command that creates a database:

```
CREATE DATABASE books;
```

If you want to delete the database again, you can do it with the command

```
DROP DATABASE books;
```

One must of course be wary of a command as above, that you do not delete a database with data - unless this is exactly what you want.

If the database management system is used to manage multiple databases (and it will always be the case), and you opens Microsoft SQL Server Management Studio, you must tell which database the SQL commands should works on. You do this with a command like

```
USE Books;
```

which makes the database *books* to the current database.

Is the database *books* the current database, you can create a table as follows:

```
CREATE TABLE Publisher (
Publisher_id INT IDENTITY(1,1) NOT NULL,
Publisher_name NVARCHAR(40) NOT NULL,
PRIMARY KEY (Publisher_id));
```

Here you creates a table named *Publisher*. The table has two columns, where the first is called *Publisher_id* and has the type *INT* and thus must contain integers, while the other is called *Publisher_name* and has type *NVARCHAR* to hold max 40 characters and then can contain text. Both columns are defined NOT NULL, and a row must have a value for both *Publisher_id* and *Publisher_name*. Finally is defined that the column *Publisher_id* must be the primary key. The command is written on several lines. It is not necessary and is only done for the sake of readability, but you should note that where there is a comma, as they must be there. The command in question may in addition also be written as follows:

```
CREATE TABLE Publisher (
Publisher_id INT IDENTITY(1,1) PRIMARY KEY,
name NVARCHAR(40) NOT NULL );
```

When I have not specified NOT NULL for the column *Publisher_id*, it is because a column defined as a primary key will automatically be NOT NULL.

Below is another example of a command that creates a table:

```
CREATE TABLE Book (
Book_isbn CHAR(13) NOT NULL,
Book_title VARCHAR(100) NOT NULL,
Book_edition INT,
Book_year CHAR(4),
Book_pages INT,
Book_Publisher_id INT, NOT NULL,
Book_Category_id INT,
PRIMARY KEY (Book_isbn),
FOREIGN KEY (Book_Publisher_id) REFERENCES Publisher (Publisher_id),
FOREIGN KEY (Book_Category_id) REFERENCES Category (Category_id));
```

Here you creates a table book with 7 columns, which should contain information about a book. Here is the first column is primary key. The next column is defined NOT NULL corresponding to that a book must have a title. The next three columns allow NULL values because you are welcome to create a book without knowing when the book is released, the edition and the page number. The column *Publisher_id* is foreign key to the *Publisher* table and is defined NOT NULL. This means that a book must have a publisher. The column *Category_id* is also a foreign key (to the table *Category*), but it may well be null. This means that a book need not be assigned to a category, but if there is a value in the column, it must be the number of an existing category.

If you want to delete a table, the syntax is

```
DROP TABLE Book;
```

which deletes the table *book* with all its data. You should note that if you instead tries to execute the command

```
DROP TABLE Publisher;
```

it would result in an error when the table *Book* via the foreign keys refers to the table *Publisher*.

It is also possible to modify database objects. As an example the following command adds a new column to the table book:

```
ALTER TABLE Book ADD Info NVARCHAR(1024);
```

Similarly, the following command to change the data type of the new column:

```
ALTER TABLE Book MODIFY Info Text;
```

If you want to delete the new column again, you can do it with the following command:

```
ALTER TABLE Book DROP COLUMN Info;
```

If, for example it is such that the book's *Book_edition* must be less than 20, you can execute the command:

```
ALTER TABLE book ADD CONSTRAINT Book_edition_
Check CHECK (edition < 20);
```

where the constraint has been given a name, so you can refer to it in SQL commands. Subsequently, one could modify this constraint as follows:

```
ALTER TABLE book ADD CONSTRAINT Book_edition_Check
CHECK (edition IS NULL OR (edition BETWEEN 1 AND 19));
```

there would be a more accurate control.

As a final example the below shows a command that changes the name of a column:

```
ALTER TABLE Book CHANGE Book_edition Book_edit INT;
```

I mention the command because it can be useful, but also because it is an example of a command whose syntax depends on the current database manager.

There are many other options with the command ALTER but the above gives an impression of the syntax and what is possible.

DML COMMANDS

The above commands are all examples of DDL commands and thus commands that modify the structure of the database. The main use of these commands are in scripts that create databases. In this section I will show the syntax of the three DML commands, and unlike DDL commands, these are commands that change the content of the tables.

In this section and also in the rest of this appendix, it is assumed that the database *Books* are as described in the introduction and is loaded with data as described there.

I will start with the command INSERT that insert a row in a table. As an example, the following command inserts a row in the *author* table:

```
INSERT INTO author VALUES ('Mogens', 'Trolle');
```

In order that the command can be executed, there are two requirements:

1. After *VALUES* must be in parentheses a value for each of the two last columns in the table and the types must match.
2. There is no value (and it must not be) for the first column as it is an auto generated primary key

In many ways it is the simplest INSERT command, as one can imagine.

Often, the insertion of a row, however, require the insertion of rows in multiple tables. If for example, I want to create the book

78-7900-910-7, Afrikas dyreliv published on *Globe* as edition 1 and written by *Mogens Trolle*

then the publisher does not exists and must first be created. When the author is already created (with the command above), there should also be added a row to the relation table *written* and the book can be created as follows:

```
INSERT INTO Publisher VALUES ('Globe');
INSERT INTO Book (Book_isbn, Book_title, Book_
edition, Book_Publisher_id)
  VALUES ('87-7900-910-7', 'Afrikas dyreliv', 1, 506);
INSERT INTO written VALUES ('87-7900-910-7', 1946);
```

The first command requires no additional comments, but for the following command you must know the auto generated *Publisher_id* which in this case is 506.

The second command does not insert values in all columns, and therefore must specify in which columns to be inserted values. This is done by listing the column names in parentheses after the table name. Then VALUES must specify values for these columns. You should note that *Book_isbn* is necessary because it is the primary key. Also *Book_title* is necessary, when this column is defined NOT NULL and when there is no value for *Book_year* and *Book_pages* it is fine as these columns may contain NULL values. *Book_Publisher_id* is defined NOT NULL, and you must indicate a publishing number, and when the column is a foreign key to the table *Publisher* it must be a number on an existing publisher. It is therefore necessary, that the command that creates the publisher is executed before the command that creates the book. Note, finally, that there is not defined a category, although this column (column *Book_Category_id*) is a foreign key to the table *Category*. It is not necessary, because the column allows NULL values, which in turn corresponds to, that a book does not need to have a category.

Then there is finally the last command, as there is not much to say, but you should note that the two values is a composite key, and each part of the key is a foreign key, respectively to the table *Book* and the table *Author*, and again you must know the value of the author's key in the table *Author*.

The command UPDATE is used to change the values in the columns, and as an example, the following command updates a row in table *book*:

```
UPDATE Book SET Book_year = '2010', Book_pages = 255
  WHERE Book_isbn = '87-7900-910-7';
```

The command is easy enough to understand. After SET follows a comma-separated list of column names and values, where the values are the new values. Then there is the WHERE clause, which is followed by a condition that determines which rows to be modified. A WHERE clause can be extremely complex, which also will appear from the following about SELECT, but in this case it defines exactly one row. A WHERE may well define multiple rows, and where appropriate, updated all the rows that satisfy the condition. You must specifically note that it is not a requirement that an UPDATE command has a WHERE part, and if not, all the table's rows are updated. Also note that you cannot change the value of the primary key.

As another example the following command shows how to set value in a column to NULL:

```
UPDATE Book SET Book_edition = NULL WHERE Book_isbn = '87-7900-910-7';
```

The command assumes of course that the column allows NULL values.

Then there is the DELETE command that is used to delete rows. Suppose you have completed the following command:

```
INSERT INTO Category VALUES ('Test category');
```

that inserts a row in the table *Category*. If the new category has the key 197, and you want to delete the row again, you can do it in the following way:

```
DELETE FROM Category WHERE Category_id = 197;
```

So it is simple to delete rows in a table, but there are a few things you should be aware of. First, the WHERE part can specify multiple rows, and if so all rows that satisfy the condition after the WHERE clause are deleted. Moreover, it is allowed to completely omit the WHERE clause, and if so, the command deletes all rows in the table. One should therefore be careful with DELETE, since it's easy to delete more than the thought is.

In this case, the table *book* has a foreign key to the table *Category*, and if there is a book that refers to the *Category* with the key 197, the row is not deleted. It is at least the default, but there are other options. In the table *Book* could you have defined the foreign key as:

```
FOREIGN KEY (Book_Category_id) REFERENCES Category
(Category_id) ON DELETE SET NULL
```

Where appropriate, the references with value 197 for *Book_Category_id* in the table *Book* will automatically be set to NULL. Another option is to write:

```
FOREIGN KEY (Book_Category_id) REFERENCES
Category (Category_id) ON DELETE CASCACE
```

If so, the rows of the table *Book* that refers to the number in the *Category* table will also be deleted. Cascade can be important for ensuring the integrity of the database, but there is still reason to warn a little against this clause. If you have defined the foreign key *Book_Category_id* as ON DELETE CASCADE, and you performs the command

```
DELETE FROM Category WHERE Category_id = 1;
```

then you deletes not only a row in the table *Category*, but you also deletes all rows in the table *Book* that refers to this row, and that is the vast majority (if not some of them are prevented to be deleted by a foreign key in the table *Written*).

THE SELECT COMMAND

Then there is the SELECT command, which absolutely is the command where there is most to say. The simplest command you can think of is a command of the form:

```
SELECT * FROM Publisher;
```

which extracts all rows from the table *Publisher*. A * means all columns and hence the result consists of rows that should contain values from all columns. You can replace * with the names of the columns whose values you wants to show:

```
SELECT Book_title, Book_pages FROM Book;
```

A SELECT can have a WHERE part, where you specify the rows to extract:

```
SELECT Book_title, Book_pages FROM Book WHERE Book_edition = 4;
```

Book_title	Book_pages
Structured Computer Organization	669
Java, How to Program	1546
Introduction to Java Programming	952
Software Engineering	649
Teach yourselft Visual Basic 5 in 21 Days	798

The WHERE part can by using *boolean* operators to define more complex conditions:

```
SELECT Book_title, Book_pages FROM Book WHERE (Book_edition = 1 OR
  Book_edition = 3) AND Book_pages > 650 AND Book_pages < 700;
```

Book_title	Book_pages
Introduction to Digital Communications	672
Understanding Java	680
The Object of Java	670
Beginning Active Server Pages 2.0	652

It is also possible to specify sub-strings. As an example the following command extracts all rows where title starts with the word Java

```
SELECT Book_title, Book_pages FROM Book WHERE Book_title LIKE 'Java%';
```

while the following command extracts all rows where the title contains the word Java

```
SELECT Book_title, Book_pages FROM Book
WHERE Book_title LIKE '%Java%';
```

The rule is that % matches 0 or more arbitrary characters. As another example extracts the command

```
SELECT Book_title, Book_year, Book_pages FROM
Book WHERE Book_year LIKE '19__';
```

all rows where the year (publisher year) starts with 19 and is followed by two characters, since the rule is that _ exactly matches one character.

The following command extracts the first three titles in the table *book*:

```
SELECT TOP 3 Book_title FROM Book;
```

TOP can also be combined with a WHERE:

```
SELECT TOP 3 Book_title FROM Book WHERE edition = 2;
```

You should be aware that other database systems instead of SELECT TOP uses LIMIT after the WHERE part.

A SELECT can have an ORDER BY, where the rows are sorted according to values in a particular column. Thus, the following command sorts the rows by title in ascending order:

```
SELECT Book_isbn, Book_title FROM Book ORDER BY Book_title;
```

If you wish instead that the rows are sorted in descending order, one can write:

```
SELECT Book_isbn, Book_title FROM Book ORDER BY Book_title DESC;
```

You can also sort by multiple criteria. Consider the following command, which extracts *Book_isbn*, *Book_title* and *Book_edition* for all rows where the category number is 164, but so that the rows first are sorted by *Book_edition*, and within each edition by the *Book_title*:

```
SELECT Book_isbn, Book_title, Book_edition FROM
Book WHERE Book_Category_id = 164
ORDER BY Book_edition, Book_title;
```

GROUP BY

For a SELECT you can also define GROUP BY, which indicates that the result should contain one row for each value in the column that will be grouped. For example

```
SELECT Book_edition FROM Book GROUP BY Book_edition;
```

that will show all editions. There are six rows with the numbers 1, 2, 3, ..., 6. It is not so interesting for the same result could be achieved in other ways, and GROUP BY has also only of interest if you want to do something by the rows that fall within the individual groups. If, for example you were interested in determining the sum of all pages distributed on edition you could use the command:

```
SELECT Book_edition, SUM(Book_pages) FROM Book GROUP BY Book_edition;
```

Book_edition	(No column name)
NULL	255
1	54369
2	16966
3	6478
4	5314
5	471
6	1487

You should note that the result shows 7 rows. This is because there is a book in which the value of *Book_edition* is NULL.

GROUP BY can be combined with both the WHERE and ORDER BY, and the order must be as shown below:

```
SELECT Book_edition, SUM(Book_pages), COUNT(*) FROM Book
WHERE Book_pages < 1000 GROUP BY Book_
edition ORDER BY Book_edition DESC;
```

Here is selected the group (the edition), the sum of all pages within the group, the number of rows within the group, but only for the books where the page number is less than 1000:

Book_edition	(No column name)	(No column name)
5	471	1
4	3768	5
3	5469	9
2	13390	20
1	39794	92
NULL	255	1

A WHERE clause defines which rows to be extracted, and you can therefore think of the WHERE clause as a filter that filters the rows. Similarly, in connection with GROUP BY there is a HAVING clause, that is a filter that specifies which groups to include. The syntax is as follows:

```
SELECT Book_edition, SUM(Book_pages), COUNT(*) FROM Book
WHERE Book_pages < 1000 GROUP BY Book_edition
HAVING Book_edition = 2 OR Book_edition = 4 OR Book_edition = 6
ORDER BY Book_edition DESC;
```

where the HAVING clause specifies that only groups with a value of 2, 4 or 6 should be extracted:

Book_edition	(No column name)	(No column name)
4	3768	5
2	13390	20

If you execute the command

```
SELECT Book_edition FROM Book;
```

you get shown all editions, and there are 87 rows. This means that the same value appears several times. If you do not want that, you can write

```
SELECT DISTINCT Book_edition FROM Book;
```

which simply means that all rows must be different.

JOIN

It is also possible to perform a SELECT command that extracts rows from multiple tables, and we often talk about a JOIN. As an example is shown a JOIN command, that extracts the title from the *Book* table and the *name* from the publishers table, but such that there only are extracted a row where the value in column *Book_Publisher_id* and *Publisher_id* is similar in the two tables:

```
SELECT Book_title, Publisher_name FROM Book, Publisher
WHERE Book_Publisher_id = Publisher_id;
```

Put slightly differently, so are each row of the *Book* table combined with each row in the *Publisher* table, but only rows with same value in the columns for *Book_Publisher_id* and *Publisher_id* are included in the result. The result has then 87 rows with two columns. You should note that the two columns names *title* and *name* are unique determined in the combination of the two tables.

If you do not specify concrete columns in a join such as

```
SELECT * FROM Book, Publisher WHERE Book_Publisher_id = Publisher_id;
```

the result is still the 87 rows, but all columns from both tables is there.

There are several types of JOIN on tables, and an example as the above is called an INNER JOIN and can also be written as follows:

```
SELECT Book_title, Publisher_name FROM Book INNER JOIN Publisher
ON Book_Publisher_id = Publisher_id;
```

An INNER JOIN between two tables returns rows in which there are matches in both tables. An INNER JOIN (and all other JOIN operations) can also be combined with a WHERE clause as such

```
SELECT Book_title, Category_name FROM Book INNER JOIN Category ON
Book_Category_id = Category_id WHERE Book_isbn LIKE '87-%';
```

that returns the title and category name for the books where *Book_isbn* starts with 87- (all Danish titles):

In addition to INNER JOIN, there are the following JOIN operations:

- *LEFT JOIN*, which returns all rows from the left table, even if there is no match in the right table.
- *RIGHT JOIN*, which returns all rows of the right table, even if there is no match in the left table.
- *FULL JOIN*, which returns all rows, even if there is no match in one of the two tables.
- *CARTESIAN JOIN*, that returns the cartesian product of the two tables.

You should be aware that the two last is not always supported. As an example I will show a LEFT JOIN:

```
SELECT Book_title, Category_name FROM Book LEFT JOIN Category ON
Book_Category_id = Category_id WHERE Book_isbn LIKE '87-%';
```

Note that in principle it is the same JOIN as above, but this time the result shows seven rows. This is because the *book* table has a row, which is NULL in the column *Book_Category_id* and therefore does not match a row in the *Category* table, but it is included in the result as opposed to an INNER JOIN where there must be a match in both tables.

Below is a RIGHT JOIN:

```
SELECT Book_title, Category_name FROM Book RIGHT JOIN Category ON
Book_Category_id = Category_id WHERE Book_isbn LIKE '87-%';
```

It also shows 6 rows (that is the same result as the corresponding INNER JOIN) because the table *Category* does not have rows with a value in column *Category_id*, which does not match a row in the *Book* table.

The next command is again an example of an INNER JOIN:

```
SELECT Book_title, Publisher_name FROM Book, Publisher
WHERE Book_Publisher_id = Publisher_id AND Book_isbn LIKE '87-%';
```

and the result is 7 rows. If you delete the JOIN condition, you get a CARTESIAN JOIN:

```
SELECT Book_title, Publisher_name FROM Book, Publisher
WHERE Book_isbn LIKE '87-%';
```

and the result is 147 rows. The reason is that each row of the table *Book* (there are 7 meeting the WHERE clause) is combined with all rows in the *Publisher* table (which is 21), and the result is a total of 7 x 21 = 147 rows.

It is also possible to join more than two tables. The following command is a JOIN between three tables showing for each *title*, where the *isbn* starts with 87 a row with the *title* and author's *name* (one row for each author):

```
SELECT Book_title, Author_firstname, Author_lastname FROM Book
INNER JOIN Written ON Book_isbn = Written_isbn
INNER JOIN Author ON Written_Author_id = Author_id
WHERE Book_isbn LIKE '87-%';
```

By combining (join) *Book* and *Written* you get for each book the author numbers for the book's authors, and by combining this result with the table *Author* you can obtain the authors' names. The command in question can also be written as follows:

```
SELECT Book_title, Author_firstname, Author_
lastname FROM Book, Written, Author
WHERE Book_isbn = Written_isbn AND Written_Author_id = Author_id
AND Book_isbn LIKE '87-%';
```

As another example, shows the following command isbn and title of all mathematic books published by Prentice Hall:

```
SELECT Book_isbn, Book_title FROM Book, Category, Publisher
WHERE Book_Category_id = Category_id AND Book_
Publisher_id = Publisher_id
AND Category_name LIKE '%Matematik%'
AND Publisher_name LIKE '%Prentice Hall%';
```

As appears from the JOIN operations NULL values can sometimes lead to unexpected results. Although it has nothing to do with joins, please note the following syntax

```
SELECT * FROM Book WHERE Book_edition IS NULL;
```

which returns all books whose value for *Book_edition* is NULL. Similarly, one can write

```
SELECT * FROM Book WHERE Book_edition IS NOT NULL;
```

When you joins several tables that can occur name matches, where two columns has the same name. You can solve this problem by qualifying the name with the table name. It can lead to a somewhat clumsy syntax, and additionally, it may mean that there are two columns in the result, with the same name. To solve these problems, you can use the concept of an alias, where you can give a table or a column a custom name. The following command determines the title, publisher name and category name for all Danish books:

```
SELECT B.Book_title AS Text, P.Publisher_name AS 'Publisher name',
C.Category_name AS 'Category name'
FROM book AS B, publisher AS P, category AS C
WHERE B.Book_Publisher_id = P.Publisher_id AND
B.Book_Category_id = C.Category_id AND B.Book_isbn LIKE '87-%';
```

You should note that in this case there is no justification for using qualifying names as all columns in the database have different names.

There is extracted data from three tables: *book, publisher* and *category*, but these tables are assigned names *B*, *P* and *C*. In addition are each of the three columns assigned a name, and you should note that this names are used in the result:

Text	Publisher name	Category name
Politikens Store Bog om Digital Foto	Politikens Forlag A/S	Foto
Programmering i Prolog	Borgen	Informations teknologi
Vin 2005	Politikens Forlag A/S	Vin
Rhone Vinene	Politikens Forlag A/S	Vin
Struktureret Programudvikling	Teknisk Forlag	Informations teknologi

Set operations

If two SELECT commands are union compatible, that is they results in the same number of columns, columns of the same types, and the columns in the same order, one can perform a UNION:

```
SELECT Book_isbn, Book_title, Book_edition FROM
Book WHERE Book_isbn LIKE '87-%'
UNION
SELECT Book_isbn, Book_title, Book_edition FROM Book
WHERE Book_edition = 3 OR Book_edition = 4;
```

The first command returns 7 rows and the second 11 rows. Because UNION removes duplicate rows, and there are two rows that are identical, the command will result in 16 rows. If you instead writes

```
SELECT Book_isbn, Book_title, Book_edition FROM
Book WHERE Book_isbn LIKE '87-%'
UNION ALL
SELECT Book_isbn, Book_title, Book_edition FROM Book
WHERE Book_edition = 3 OR Book_edition = 4;
```

are duplicate rows retained, and the command will result in 18 rows.

Instead of UNION you can write INTERSECT, and the result is then the intersection of the result from the two SELECT commands. It is also possible to write EXCEPT that means set difference.

SQL FUNCTIONS

As part of SQL are a number of functions that can be used in SQL commands. Above I have already used the SUM() and COUNT(). The following is a listing of the most important of those functions, and there are many others, but they can be divided into groups:

1. general functions
2. numeric functions
3. functions to strings
4. functions to date and time

and in addition there is a (sometimes large) number of functions, depending on the database product.

General functions

This group of functions include:

- COUNT
- MAX
- MIN
- SUM
- AVG

and are the classic functions, where I above has used COUNT and SUM. As an example determines the following command the number of books, the total number of pages (total number of pages of all the books), the average number of pages, the smallest number of pages and the largest number of pages:

```
SELECT COUNT(*), SUM(Book_pages), AVG(Book_
pages), MIN(Book_pages), MAX(Book_pages)
FROM Book;
```

(No column name)	(No column name)	(No column name)	(No column name)	(No column name)
146	85340	584	50	1632

Numeric functions

ABS	ACOS	ATAN	ATAN2	BIT_AND
BIT_COUNT	BIT_OR	CEIL	CEILING	CONV
COS	COT	DEGREES	EXP	FLOOR
FORMAT	GREATEST	INTERVAL	LEAST	LOG
LOG10	MOD	OCT	PI	POW
POWER	RADIANS	RAND	ROUND	SIN
SQRT	STD	STDEV	TAN	TRUNCATE

The meaning of most of the functions is indicated by the name, and you cannot be sure that all database systems implements all this functions. Consider as an example the following statements:

```
DECLARE @my REAL;
SET @my = 0;
SELECT @my = AVG(Book_pages) FROM Book;
SELECT RAND(), PI(), SQRT(2), STDEV(Book_pages),
SQRT(SUM((Book_pages - @my) * (Book_pages - @
my)) / COUNT(*)) AS Sigma FROM Book;
```

(No column name)	(No column name)	(No column name)	(No column name)	Sigma
0,751648222094409	3,14159265358979	1,4142135623731	329,914225408051	328,782852934215

Here are the results of the first three functions simple enough, while the fourth function determines the standard-deviation of the books pages. The first statement defines a variable and the next statement set this variable equal to the average (mean value) of the number of pages. The value of this variable is used in the last formula, which calculates the standard deviation of the number pages. It is of course no reason for that calculation, because there is a function STDEV that do the same, but the example should partly show how you can use a variable and partly an example of a complex expression.

Functions to strings

ASCII	BIN	BIN_LENGTH	CHAR_LENGTH
CHARACTER_LENGTH	CONCAT	CONV	ELT
EXPORT_SET	FIELD	FIND_IN_SET	FORMAT
HEX	INSERT	INSTR	LCASE
LEFT	LENGTH	LOAD_FILE	LOCATE
LOWER	LPAD	LTRIM	MAKE_SET
MID	OCT	OCTET_LENGTH	ORD
POSITION	QUOTE	REGEXP	REPEAT
REPLACE	REVERT	RIGHT	RPAD
RTRIM	SOUNDEX	SOUNDEX_LIKE	SPACE
STRCMP	SUBSTR	SUBSTRING	SUBSTRING_INDEX
TRIM	UCASE	UNHEX	UPPER

Most of the functions are easy enough to understand, but not all, and there it is necessary to look up the meaning for the function. What features are actually available depends on the current database product, and you can only use the above table as inspiration for what functions you can expect to find. As an example determines the following statement the title's length and the first 10 characters in the title of all books whose title contains the word Java:

```
SELECT LEN(Book_title), SUBSTRING(Book_title, 1, 10) FROM Book
WHERE Book_title LIKE '%Java%';
```

Functions to date and time

As discussed above, SQL types for dates could be difficult to use correct, and there are a number of functions that specifically are used for dates:

ADD_DATE	ADD_TIME	CURDATE	CURRENT_DATE
CURRENT_TIME	CURTIME	DATE_ADD	DATE_FORMAT
DATE_SUB	DATE	DATEOFMONTH	DAYOFWEEK
DAYOFYEAR	EXTRACT	FROM_DAYS	FROM_UNIXTIME
HOUR	LAST_DAY	LOCALTIME	LOCALTIMESTAMP
MAKEDATE	MAKETIME	MICROSECOND	MINUTE
MONTH	MONTHNAME	NOW	PERIOD_ADD
PERIOD_DIFF	QUATER	SEC_TO_TIME	SECOND
STR_TO_DATE	SUBDATE	SUBTIME	SYSDATE
TIME_FORMAT	TIMESTAMP	TIMESTAMPADD	TIMESTAMPDIFF
TO_DAYS	UNIX_TIMESTAMP	UCT_DATE	UCT_TIME
UCT_TIMESTAMP	WEEK	WEEK_DAY	YEAR

As the table shows, there are many functions and it is not so easy to figure out the meaning of all of them, but the result is that there are many opportunities to manipulate date and time in SQL. As a small example, the following statement use two of this functions:

```
DECLARE @d DATE;
DECLARE @t TIME;
SET @d = GETDATE();
SET @t = CONVERT(TIME, GETDATE());
SELECT @d, @T;
```

(No column name)	(No column name)
2020-05-05	10:58:59.6930000

VIEW'S

A view is basically nothing more than a SQL statement that is stored in the database, but it can be seen as a form of virtual table, as you in principle can use in the same way, as you uses other tables. A view can be created on basis of one or more tables, but what you can do with a view is determined by how it is created. The purpose of a view is

- to structure the contents of a database corresponding to the users' use of the database
- limiting access to the data that users can work with
- allowing data from multiple tables to appear as a single table

In SQL Server Management Studio you can create a view by right-click on *Views* and choose *Create View*. You then get a wizard where you can select tables and columns and edit the SELECT statement:

```
SELECT Book_isbn, Book_title, Book_year, Book_edition, Book_pages
FROM   dbo.Book
WHERE (Book_Publisher_id = 459)
```

When you close the window you are asked for a name, and in this case I saved the view called *Prentice_hall*. The view consists of a SQL SELECT which find all books where *Book_Publisher_id* is 459 and that is *isbn*, *title*, *edition*, *year* and *pages* for all books published on Prentice Hall, and then a view of the columns in the table *Book* of all books published by Prentice Hall.

This view may then be used in the same way as the other tables, and for example you can write:

```
SELECT * FROM Prentice_hall;
```

Indeed, one can also to some extent perform SQL INSERT, UPDATE and DELETE statements in a view. Consider as an example the following view, which extracts first and last names of all authors where the last name starts with a K:

```
SELECT DISTINCT Author_firstname, Author_lastname
FROM   dbo.Author
WHERE (Author_lastname LIKE 'K%')
```

I have called the view *Knames*, and if you then performs the statement

```
SELECT * FROM Knames;
```

you will see that the view contains 8 rows. If you then performs the following statements:

```
INSERT INTO Author VALUES ('Poul', 'Klausen');
SELECT * FROM Knames;
```

you will see that the view *Knames* now has 9 rows. This means that after the table *Author* is updated is also the view updated.

Next, if you try:

```
INSERT INTO Knames VALUES ('Jens', 'Kristensen');
```

you get an error where you are told that the parent table cannot be updated. The problem is DISTINCT which is an operator and means that the selected rows are modified. Consider the following view called *Lnames*:

```
SELECT     Author_firstname, Author_lastname
FROM       dbo.Author
WHERE     (Author_lastname LIKE 'L%')
```

that this time selects all rows from the table *Author* where the last name starts with L. If you then performs the statements:

```
INSERT INTO Lnames VALUES ('Knud', 'Larsen');
SELECT * FROM Author WHERE Author_lastname = 'Larsen';
```

and you will see that both the view and the original table is updated. This means that an updates of the view also updates the parent table, but there are the following conditions:

- the SELECT statement may not use DISTINCT
- the SELECT statement may not contain functions
- the SELECT statement may not contain operators
- the SELECT statement may not use ORDER BY
- the SELECT statement can only use one table
- the SELECT statement's WHERE part may not use a SELECT statement (see below)
- the SELECT statement may not use GROUP BY or HAVING
- the SELECT statement must include all columns from the parent table that is defined NOT NULL or is an auto generated primary key

INNER SELECT STATEMENTS

It is possible to use an inner SELECT statement in a WHERE clause where an inner SELECT returns data that is used in the condition. An inner SELECT can be used both in a SELECT, INSERT, UPDATE and DELETE - typically associated with operators. A typical example is, using SELECT IN:

```
SELECT Book_isbn, Book_title FROM Book WHERE Book_Publisher_id IN
(SELECT Publisher_id FROM Publisher WHERE
Publisher_name = 'Prentice Hall' OR Publisher_
name = 'Addison Wesley');
```

The result is *isbn* and *title* of all books published by either Prentice Hall or Addison Wesley. Of course one can achieve the same otherwise, but SELECT IN is easy to understand.

As another example, creates the following script a new table with all the math books when the table must contain the same columns as table *Book*, but except column *Book_Category__id*:

```
use books;

create table Math (
isbn char(13) not null,
title nvarchar(100) not null,
edition int,
year char(4),
pages int,
pubid int not null,
primary key (isbn),
foreign key (pubid) references Publisher (Publisher_id));

INSERT INTO Math (isbn, title, edition, year, pages, pubid)
SELECT Book_isbn, Book_title, Book_edition, Book_year, Book_pages,
Book_Publisher_id FROM Book, Category
WHERE Book_Category_id = Category_id AND Category_name = 'Matematik';
```

What is interesting is the last *INSERT INTO* statement that inserts data from a *SELECT*.

STORED PROCEDURES

I will conclude this appendix with a very brief introduction to stored procedures, which is actually a large area. A stored procedure is a routine consisting of SQL statements that are stored on the database server together with the database tables, and the idea is of course to save the SQL procedures which are often needed, but also that a stored procedure is translated into an internal format and thus is effective.

Below I will show a few examples of simple procedures, and how they are created using SQL Server Management Studio. In the database (*Books*) right-click on *Programmability* and here on *Stored Procedures* and select *Stored Procedure* and SQL Server Management Studio creates a skeleton for a stored procedure where I have removed the comments:

```
SET ANSI_NULLS ON
GO
SET QUOTED_IDENTIFIER ON
GO
CREATE PROCEDURE <Procedure_Name, sysname, ProcedureName>
  <@Param1, sysname, @p1> <Datatype_For_Param1, , int> =
    <Default_Value_For_Param1, , 0>,
  <@Param2, sysname, @p2> <Datatype_For_Param2, , int> =
    <Default_Value_For_Param2, , 0>
AS
BEGIN
  SET NOCOUNT ON;
  SELECT <@Param1, sysname, @p1>, <@Param2, sysname, @p2>
END
GO
```

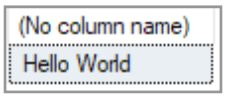
You can then write a stored procedure named *Hello* as follows:

```
USE Books;
SET ANSI_NULLS ON
GO
SET QUOTED_IDENTIFIER ON
GO
CREATE PROCEDURE Hello
AS
BEGIN
  SET NOCOUNT ON;
  SELECT 'Hello World';
END
GO
```

and if you then click *Execute* the procedure is translated, and if there is no errors, it is stored in the database. It is a very simple procedure which only print a text, but if you performs the procedure as:

```
EXEC Hello;
```

the result is that the procedure prints *Hello World* on the screen:



It is obviously not a particularly interesting procedure, but it shows the principle and that is simply the case that one or more SQL statements can be executed under a common name. In addition to seeing some more examples, there are basically two things that must be addressed, namely parameters and program logic.

As an example is below shown a procedure, that calculate the number of books, where the number of pages is greater than or equal to *a* and less than or equal to *b*, and where *a* and *b* are input parameters. The result is saved in an output parameter *c*:

```
SET ANSI_NULLS ON
GO
SET QUOTED_IDENTIFIER ON
GO
CREATE PROCEDURE Counter
  @a INT,
  @b INT,
  @c INT OUTPUT
AS
BEGIN
  SET NOCOUNT ON;
  SELECT @c = COUNT(Book_isbn) FROM Book WHERE
@a <= Book_pages AND Book_pages <= @b;
END
GO
```

You should notice how the procedure stores the result in c. The following shows how the procedure can be used to calculate the number of books, where the number of pages is greater than or equal to 200 and less than or equal to 500:

```
use Books;
DECLARE @num int;
EXEC Counter 200, 500, @num output;
select @num;
```

It is also possible to use a parameter for both input and output and you must just define it as OUTPUT parameter. The following procedure determines the average of number of pages in books where the page number is less than or equal to the value of the parameter *t*:

```
SET ANSI_NULLS ON
GO
SET QUOTED_IDENTIFIER ON
GO
CREATE PROCEDURE Average
  @t INT OUTPUT
AS
BEGIN
  SET NOCOUNT ON;
  DECLARE @s INT;
  DECLARE @n INT;
  SELECT @s = SUM(Book_pages), @n = COUNT(*)
  FROM Book WHERE Book_pages <= @t;
  SET @t = @s / @n;
END
GO
```

and the procedure can be used as follows:

```
use Books;
DECLARE @Num INT;
SET @num = 500;
EXEC Average @num OUTPUT;
select @num;
```

The following script creates a database with one table that has only one column:

```
use master
go
drop database if exists Primes;
create database Primes;
go
use Primes
go
CREATE TABLE Prime (Number INT PRIMARY KEY);
```

To this database I have added a stored procedure that adds the value of a parameter *n* to the table, if *n* is a prime and not already is in the table. The result of the procedure is stored in the parameter *r*.

```
SET ANSI_NULLS ON
GO
SET QUOTED_IDENTIFIER ON
GO
CREATE PROCEDURE IsPrime
  @n INT,
  @r INT OUTPUT
AS
BEGIN
  SET NOCOUNT ON;
  DECLARE @c INT;
  SELECT @c = count(*) FROM Prime WHERE Number = @n;
  IF @c > 0
  BEGIN
    SET @r = 0;
  END
  ELSE IF @n = 2 OR @n = 3 OR @n = 5 OR @n = 7
  BEGIN
    SET @r = 1;
  END
  ELSE IF @n < 11 OR @n % 2 = 0
  BEGIN
    SET @r = 0;
  END
  ELSE
  BEGIN
    DECLARE @t INT;
            SET @t = 3;
    DECLARE @m REAL;
            SET @m = SQRT(@n) + 1;
    SET @r = 1;
    WHILE @t <= @m AND @r = 1
            BEGIN
      IF @n % @t = 0 SET @r = 0;
      ELSE SET @t = @t + 2;
    END;
  END;
  IF @r = 1 INSERT INTO Prime VALUES (@n);
END
GO
```

and below an example of an application of the procedure:

```
use Primes;
DECLARE @Res INT;
EXEC IsPrime 2, @res OUTPUT;
SELECT @res;
EXEC IsPrime 3, @res OUTPUT;
SELECT @res;
EXEC IsPrime 11, @res OUTPUT;
SELECT @res;
EXEC IsPrime 10, @res OUTPUT;
SELECT @res;
EXEC IsPrime 7, @res OUTPUT;
SELECT @res;
EXEC IsPrime 11, @res OUTPUT;
SELECT @res;
EXEC IsPrime 5, @res OUTPUT;
SELECT @res;
SELECT * FROM Prime;
```

(No column name)
1

(No column name)
1

(No column name)
1

(No column name)
0

(No column name)
1

(No column name)
0

(No column name)
1

Number
2
3
5
7
11

The above procedure does not have great practical interest, and the goal is alone show that in a stored procedure can use program logic in the same way as in for example C#.

As a final comment should be added that there is much more to say about stored procedures, and you may also have stored functions, but the above should suffice to illustrate what a stored procedure is. Also note that for stored procedures there is a big difference in the syntax between the different SQL variants.