

# C# 7

## About System Development

Software Development

POUL KLAUSEN

---

# **C# 7: ABOUT SYSTEM DEVELOPMENT**

SOFTWARE DEVELOPMENT

C# 7: About system development: Software Development

1<sup>st</sup> edition

© 2020 Poul Klausen & [bookboon.com](https://bookboon.com)

ISBN 978-87-403-3535-4

# CONTENTS

	<b>Foreword</b>	<b>6</b>
<b>1</b>	<b>Introduction</b>	<b>8</b>
<b>2</b>	<b>The waterfall model</b>	<b>10</b>
2.1	The task-formulation	11
2.2	Analysis	12
2.3	Design	18
2.4	Programming	24
2.5	Test	26
2.6	Delivery	27
<b>3</b>	<b>A system development method</b>	<b>29</b>
3.1	Other methods	30
3.2	Code and test	32
3.3	Unified Process	33
3.4	SCRUM	39
3.5	Extreme Programming	48
<b>4</b>	<b>Unit test</b>	<b>50</b>
	Exercise 1: Test library	54
<b>5</b>	<b>Test-Driven Development</b>	<b>56</b>
	Exercise 2: The test library again	62
<b>6</b>	<b>Design patterns</b>	<b>66</b>
<b>7</b>	<b>Refactoring and other</b>	<b>69</b>
7.1	Refactoring	69
7.2	Log files	71
<b>8</b>	<b>Database design</b>	<b>72</b>
8.1	The ER diagram	72
8.2	Mapping to relational model	76
8.3	Normalization	80
8.4	Other database improvements	83
8.5	The use of a class diagram	84
8.6	Create the database	87

<b>9</b>	<b>More on databases</b>	<b>92</b>
9.1	Transactions	92
9.2	Concurrency	98
9.3	Other database products	105
9.4	History people	107
	Problem 1: The World	111
<b>10</b>	<b>Final example: MyWines</b>	<b>112</b>
10.1	The task	112
10.2	Development plane	115
10.3	Analysis	116
10.4	Database design	121
10.5	The model layer	128
10.6	The MainWindow	133
10.7	The dimension tables	135
10.8	Maintenance of wines	139
10.9	The search functions	142
10.10	The last things	145

# FOREWORD

This book is the seventh in a series of books on software development. The programming language is C#, and the language and its syntax and semantics fills obviously much, but the books have also largely focus on the process, and how to develop good and robust applications. In this book focuses is primarily on system development and method. The book does not focus directly on a specific system development method, but addresses a number of principles which are general and can be applied in all system development projects and which may be useful guidelines for developing applications in practice. The aim is to set up a framework in which, that for at least development of small programs, can be useful to provide both progress during the development and quality of the finished product. In addition, the book contains more about database programming which are concepts that were not accommodated in the previous book.

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#. It can be learned 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 being treated in the books. All books in the series is 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 that is presented in the text, and furthermore it is relatively accurately described what to do. Problems are in turn 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 it is a program that you quickly become familiar 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

System development is the name of the process to develop a program from start to finish, and thus from the idea or the task is presented, to the program is written, tested and put into operation at the future users. In practice, a system development is performed as a project that is characterized by a longer period, and the task is typically solved in a project group that performs all the work. Both the time schedule and the size of the project group is obviously determined by the task's scope. The time horizon can be anything from a few days to several years in extreme cases, while the project group can be anything from just a single person to many. For it makes sense to talk about a system development project, it will often be a project that will be performed over several weeks and perhaps even months or years, and which involves several people.

Large system development projects is characterized by many uncertainties which also means that one often hears about IT solutions, which do not end up with the desired result or maybe even running completely wrong. There are several reasons, but in my opinion are the two essential:

1. That it is difficult (probably impossible) from the start to define all the requirements, as you continuously during the development process gain greater insight and understanding of the task to be solved, and that along with it, there are creating new wishes and requirements for the finished system. It is thus impossible to estimate the resources that the project requires.
2. That you will not recognize the cost, and the development organization, to get the job, deliberately underestimates. The result is that the resources used by that task is not present, which in turn means that the finished product will be of poor quality and, at worst, useless.

You hear sometimes argued that IT solutions are so complex that it is impossible to develop programs that do not contain errors. However, it is not an acceptable explanation. It is true that IT systems can be complex, but IT systems are not the only complex thing that are developed in this world, and the demand for finished software solutions must be, that they meet the customer's requirements and works without error.

Now it is not always as bad as suggested above, and luckily it such, the vast majority of IT projects are carried out to the satisfaction of both users and developers, and here it is important to remember, that it are the projects that derail, you hear about. Fortunately, there are and will be developed many excellent programs, but effective system development requires techniques and methods. It is important to work systematically to gets everything to work. It is what the following book is about.



In addition, for programs must work correctly they must also be maintained. At least has larger programs to be maintained for a long period. It requires many resources to develop a large program, and the program should usually be used for many years. In that time, the program must often be modified as new demands occurs, or there is something you want to be done in a different way. For that to be possible, it is important that the program from the start is written in a way so it later - and probably by someone else than the persons who originally developed the program - can be modified and expanded without the changes means, that large parts of the program has to be rewritten. Whether or not depends on how the application is made and its quality. Therefore, program quality plays a major role in system development. Not only in the interests of future maintenance, but also because the programs must be effective, and if a program works slowly, it almost always has to do with how the program is a written. Finally may be mentioned that programs should be robust, so they do not fail because of improper use, so also the robustness is part of the program's quality.

All that should emphasize that at the development of any program - however small - it is important to work systematically using proven techniques and methods. It is important to use and exploit other people's experience. The conclusion is that system development is much more than clever ideas and programming.

This book is an introduction to system development, but unlike the previous books, there are no exercises. The book describes some guidelines for the development of a program and ends with two examples of the development of a program along these lines.

## 2 THE WATERFALL MODEL

Over time we have developed a variety of system development methods, all of which have had the object of providing guidelines on how, in practice, to develop a program from start to finish. The methods practical importance to system development has been variable, probably mainly because it is difficult to develop a method that is general enough to accommodate the many different systems to be developed, but also a little because the methods have often been static and easily becomes methods for the methods own sake. Therefore, it is typical in practice, to use elements of several methods, so to give some guidelines to suit the tasks you works on, but for that, everything that is said and written concerning system development methods, will not be worse. It says only that it is not so easy to find the right method, and it is hardly, but there are a number of good principles that are worth following, and that is what this book talks about.

However, there are some activities that are repeated in all system development methods:

- Analysis, which examines and determines what it is for a task to be solved, and thus formulate the requirements for the finished program.
- Design, where you decides how the task should be solved.
- Programming, where the product itself is written.
- Test, which examines that the finished program works as it should.

Even for the development of small programs you must go through these activities, but the content clearly varies depending on the size of the task. So far, I will therefore use a method, which consists of six steps or phases:

1. Task formulation
2. Analysis
3. Design
4. Programming
5. Test
6. Delivery

It is a very simple method, and for larger projects, the method is not comprehensive enough or appropriate, but for smaller programs it works very well, and as mentioned, all the six activities are parts of all system development methods. In the literature, the method is known as the *waterfall model*, since the idea is that you start to define the requirements (analysis), then outlines a solution (design), after which the solution is programmed. Finally the finished program is tested (and errors are corrected if necessary) before you are ready

to take the program in use. In connection with the development of today's complex IT solutions, it is a narrow vision of system development, but more on this later.

Since the above 6 activities are included in all system development methods, I will start with a brief description of the content of these activities. I will not describe specific tools such as diagrams and the like, and it's actually something that I only to a limited extent use in practice, but sometimes a diagram can be used to illustrate the relationship between components of a program, and then of course, make use of them.

## 2.1 THE TASK-FORMULATION

When writing a computer program, you have to start to describe what the task is, and what it is for a program to write. There is a person, a company or an organization (in the following called the customer) that has an idea for a program that they want developed. It can be anything from a few scattered thoughts, the result of a meeting or perhaps the result of a survey, but before the job can be transferred to a development team, the task must be presented. It may be verbal or in a document or a report, but in any case the first thing is to write a document that describe the task short and concisely. The task-formulation can be prepared by the customer or the development department, but typically it will be done in collaboration over a shorter period.

There are no particular requirements for neither the process or the form. For big jobs, the work can be comprehensive and take time as there must be kept several meetings, collected information etc., and we often talk of a preliminary analysis. The result can be a report, where the task is described in general terms, and how important success criteria are established. For smaller tasks, it may suffice with a single meeting, and the task formulation is perhaps no more than a single sheet of paper. In any case, the result of the task formulation is the first step towards the goal.

In general, the task formulation should not be too detailed, but it should only give a brief description of the task to be solved. The detailed requirements are established first during the analysis. The form is a document, but there may be attached other material to the description. Firstly, when writing the description of the task you already here can encounter other documentation that demonstrates important issues relating the program to be made, and thus knowledge, which is necessary for the system development. Where appropriate, this documentation must also be stored. For larger projects, it may also be that you has to document the actual writing process, for example meeting notes, and the like, such that on completion of the work you have a whole report for the task-formulation. Finally, the task-formulation and especially the process of its creation is used for estimating the task

both in terms of time schedule and resources, and in practice will to the task formulation be attached both a time schedule and an estimate of costs to complete the task.

The task formulation must therefore tell the developers what it is for a task that should be addressed and tell the customer when the task can be done, and what it cost.

Seen from the developers, the result of the activity is a project directory with a sub-directory for the task formulation, which partly includes the document with the project description and all other documents gathered or prepared and concerning the task.

## 2.2 ANALYSIS

When the task-formulation is in place, and you agree with the customer, that the task should be solved in accordance with the agreed schedule and price, the actual system development starts. The first step is to complete an analysis for the purpose

1. to clarify all ambiguities regarding the content of the task-formulation
2. to determine if there are any risks and other significant uncertainties
3. to determine external references and partners
4. to draw up a final requirement specification
5. to update the schedule, including the requirements for resource consumption

### Ambiguities

The task-formulation describes the task overall, but not the details. This must be done as part of the analysis. Therefore, there will typically be a number of questions and interpretation back, and things that are not written in the task-formulation. Everything must be covered before we can address the development of the program. It can only be done in cooperation with the customer, and the start of the analysis will typically include meetings involving the customer, but also further along in the process it may be necessary to contact the customer to clarify matters regarding the requirements for the task. The analysis is a process extending over time when there is frequent contact between the customer and the developers.

The task of clarifying the ambiguities is the actual analysis work and it is here that the developers inform themselves about the task to be solved. The developers are experts in software development, but they do not necessarily have knowledge in the field where the program should be used and the solution of many tasks require that developers gain understanding

and knowledge about how the program should be used, including the organization where the program must be used. Finally, it may happen that the solution of the task requires products and tools that the developers do not have knowledge of, and if so, it is also a knowledge to be obtained. The analysis is thus also a phase for knowledge gathering.

In practice playing the analysis phase another important role, that one should not underestimate. Under a system development project - especially a project that extends over time - there is regular contact between the developers and the customer, and as mentioned above in order to clarify matters regarding the task. It is therefore important during the analysis to obtain a relationship of trust between the developers and the customer. First, it is crucial that you as a customer can be sure of the developers privacy. Many tasks means that a developer have access to confidential information, and secondly, it is important to trust that the developers aim to do the job as well as possible, and timely reporting in with adjustments and changes compared to the already established requirements to the extent that further work reveals that something should be resolved differently than already agreed. Seen from the developers can trust of the customer be important if, for example the schedule progresses, and it may need to be adjusted.

## **Risks**

It is also during the analysis, you have to identify specific risks associated with the project. Many, especially smaller tasks is generally problem-free, but at other jobs, there may be things that you simply do not know how to be solved, and if one at all is able to solve the problem. Contains the project such challenges, it is important that they as soon as possible come to light. At worst, it may well mean that the project cannot be implemented, and if it is the case, the project should be terminated as soon as possible so as not to sacrifice more useless resources than necessary. Risks can also mean that it is necessary to include it in the contract, where certain parts of the project cannot be priced, and may not be time estimated.

Now there may also be less challenges which you instead can call uncertainties, challenges that may not directly threaten the project, but things you should be aware of as something that may take longer than expected. It can also be things that can be solved in several ways, and where it may be necessary to carry out experiments to select the right solution. Here the same applies to that kind of uncertainties, that must be recognized such you not further in the process suddenly are surprised by the sub-tasks that drains the project for all resources.

In conclusion, regardless of the degree of uncertainty that may be attached to the project, so it is extremely important that they are located as early as possible, and you make a decision on how to handle them.

## References

An IT solution and its development is not necessarily an isolated entity, but both the finished program and its development may have references to other organizations and systems.

Often the current program should exchange data with other programs and, where applicable, you must have clarified how it should be done. It can be something with the data formats or something with services, which the program has to use. In any case, information must be obtained, and is an exercise during the analysis. Perhaps it is not possible - may be because the other part will not disclose the necessary information - and if it is the case, at worst the project must be terminated. It may also be something, you have to pay for and maybe pay a license for a particular service.

Now the opposite party does not have to be an external organization, but it can, for example be a department in the customer's business. Then there are not the same problems, but the same information must be obtained and is therefore also something that must be done during the analysis.

There may also be other business partners, for example it may be, that the task only is a part of a major IT system. If applicable, you must have formalized the cooperation, and especially, it is important that the calendars are synchronized, so you do not suddenly have to wait for a delivery from a partner - and the other way around.

It may also be that you simply must have obtained information from other companies and organizations, and as an example of public authorities. It is also information, which must be provided as part of the analysis phase, or at least that the provision of the corresponding information has started.

The conclusion is that during analysis you must have obtained all the information necessary to carry out the current project, or at least agreeing when such information is available, so that the development not suddenly must be suspended because of lack of information.

## Requirements specification

In a way, the requirements specification is the core of the analysis as a determination of the total requirements for the finished program. If you have a good requirements specification, both parties - the customer and the development department - agree on what it is for a task to be solved. Many have had an opinion about the shape of the requirements specification and a template also has the advantage that it is easier to remember to get it all, but in this book the requirements specification will simply be a document without any formality, which lists the requirements for the finished program.

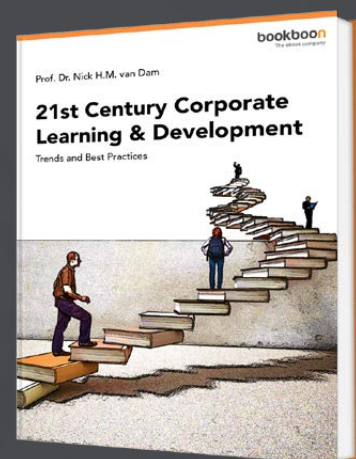
The document is, however, important, for it is the agreement, and it is the document which must be taken in the event of a disagreement between customer and development department, on what the task is and what needs to be with and possibly not. You could say that the more care taken with making the requirements specification, the greater the chance that there will be no need for it, and that the finished program is delivered to everyone's satisfaction.

The requirement specification may also contain other than documents, and I would especially point to prototypes. One of the tasks that must be clarified in connection with the development of a program, is the requirements for the user interface, and for GUI

# Free eBook on Learning & Development

By the Chief Learning Officer of McKinsey

[Download Now](#)



applications, especially web applications there are often large demands on the development of the user interface. The best way to document these requirements is the development of a prototype, that can best be explained as a program that does not do anything but showing the design of the program's windows. It is far easier to the customer to relate to a prototype than descriptions and drawings, and with modern development tools (as Visual Studio) it is easy for developers to develop good prototypes. The work is also not wasted, as it often includes something that still at a later time need to be developed.

The requirements specification is typically done at the end of the analysis, and the foundation is the task-formulation and the results of the analysis work with clarification of doubts.

### **The schedule**

I mentioned above under the task-formulation that you also has to draw up a schedule and an estimate of the resources that the task requires. It is actually something of a dilemma in system development. During the drafting of the task-formulation the development department simply lacks the necessary knowledge to prepare an estimate of resources. Conversely, the customer needs to know what the solution will cost.

The problem can be solved by letting the estimate from the task-formulation be an overall estimate. Does it seem reasonable, the customer and the development department can agree to carry out an analysis, and only then, when you have gained sufficient knowledge and uncovered all risk factors, you can conduct a detailed scheduling with its estimate of resource consumption. Only then can you make the final price calculation and possible sign a contract on the overall system development project.

The question is of course who then should pay if the result of the analysis is that the project should not be implemented - for example because it becomes too expensive. For large IT projects, it is common that the customer pays for the implementation of an analysis - probably just for a part of the overall system. Maybe the customer even has a third party to do the analysis, and in this way the task can be based on a detailed specification of requirements and put out to tender. In any case, it means that the final decision for the development of the system can be postponed until after the analysis. This strategy keeps in return not to smaller tasks. Here after the development of the task-formulation and after some general estimates - and of course a good deal of experience - the development department has to come up with a price of the task. It sounds risky, and it is that too, but nevertheless true. It's a dilemma, and there is no doubt that the source of many unfortunate IT projects must be applied here. Since the customer in his eagerness to get the job done with the least cost, he gets a product of poor quality, and when the development department in fear of not getting the job provides a low offer, so that in the end may be sacrificing quality.



Where the price is set at one time or another, then after the analysis, it is typically necessary to adjust both the time schedule and resource consumption - perhaps only for internal use - an adjustment that may result in the need to bring the project more resources as in system development is the same as man hours.

### **Documentation of the analysis**

Like the rest of the system development phases, the results of the analysis are documented in the form of an analysis report. I have already mentioned the requirement specification and the time schedule as key parts of this document, but the rest of the work must also be documented, a documentation which often will mainly consist of meeting summaries and documents collected during the analysis. There are two reasons for this documentation:

- Firstly, the documentation should be used as basis for the rest of the system development project.
- Secondly, the documentation is used in future maintenance of the program.

Documentation does not occur by itself, and it can be extensive to get it all written down. It is also something of what system development methods have been criticized, and where development departments failed to make documentation to save the work. There is, however, just as many examples that the documentation has been missing. You must then only write what is necessary, and never write documentation for the documentation's own sake - it must have value. It is just one aim of the projects in this book only to prepare the documentation, which I think is useful.

There are special IT tools to help manage and maintain this documentation and in general help manage the entire project. I will not mention such tools, but I will mention what I do in practice. If I have to write a program (implementing a system) I start as mentioned in task-formulation to create a folder for the entire project - hereafter called the project folder. Here I create sub-folders according to the magnitude of the task, but always the documentation in the form of the task-formulation, the requirements specification, other project reports and more. There may also be a sub-folder of meeting summaries, a sub-folder to other material, etc. The project folder will also contain Visual Studio projects, and in practice and by reference to other system development methods, there will be several. Already here during the analysis there could be projects for example experiments or prototypes. In the same way I typically documents the design with a Visual Studio project, a project which I will continue to work on in the programming phase, but so that I will continue the work on a copy (or more accurately initially create a copy of the project). As I work iteratively, I will start each iteration with a copy of my project from the previous iteration, and the project folder will thus contain many Visual Studio projects.

The above approach to the administration of a project is extremely simple and not much more than a simple procedure for managing a project's documents and versions, but for small projects and especially one-man projects, the procedure is working well. It is important to emphasize that for large projects with many team members you have to do it better. Here it is necessary to use IT tools, to manage different versions of documents and also the program code.

## 2.3 DESIGN

The next activity is called *design*, and short, you can say that the analysis is concerned with what the program should be able to do, while the design deals with how the program should be written. This means that the design shift the focus in system development, where the analysis has to determine the requirements and collect information while the design tackles how to construct the product. It is not supposed to write the program in the design phase, but the the gap between design and programming is not sharp, so that the design also can include programming, and individual decisions that really are design decisions can with benefit be deferred to the programming phase.

Design of IT systems is very similar to the design of other products or construction projects. If you are building a house, then you also starts making drawings showing the rooms location and size, where the windows and doors must be and so on. If not the result would probably be a house that were somewhat random, and you would risk that doors was placed so they could not be opened, or there was rooms without doors or whatever. The quality of a house requires planning and models. It is the same with IT systems. If you do not implement a proper design when considering different solutions against each other, you get perhaps a program that solves the task, but the program structure is random, and it becomes impossible to maintain in the future. The goal of the design is precisely to ensure decisions that meet program quality.

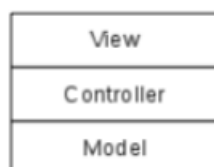
In contrast to the analysis the design activities varies highly dependent on what the task is, and it is difficult in the same way to establish guidelines for the implementation of the design phase, but the following are typical activities:

- Design of the overall program architecture.
- Design of the model layers classes.
- Design of the program's repository.
- Design of algorithms.
- Design of the user interface.

but it may also become necessary to carry out experiments for testing solutions and technologies.

## Architecture

Applications may consist of many classes and can easily be 100 or more. In this case, it can be hard to keep track of the meanings of all classes, and often one choose therefore to organize the classes as concerning the same concept in modules. One speaks in this context about application architecture, and one should generally aim for a modular architecture. There are no clear best architecture, but experience has shown that programs with a graphical user interface with advantage can be based on a three-layer architecture



or an architecture that is a variant thereof. Here you define the model as classes that models the program's data, so you have an object-oriented representation of the data that the program should work with. The view layer contains everything regarding the user interface in the form of windows. In between these two layers you have as a controller layer receiving input and converts it into commands for the model.

There is nothing magical in this architecture, but experience has shown to be advantageous to gather everything regarding the user interface in a separate layer, and everything that has to do with the program's data in a model layer. Between these two layers are classes that will tie it all together. The architecture is often called the MVC for *Model View Controller*, and it is an example of a design pattern. It comes in several flavors, and the reason is partly due to the development tools and the types of applications as for example GUI applications, web applications, apps for phones and more. The questions that a specific variant addresses is what each layer should contain and especially how the communication between the layers should be.

The layers are a draw for a modular program architecture, but there may well be multiple layers, and each layer can - and will often be - divided into modules across.

In the previous book I presents the Model-View-ViewModel pattern which is a variant of the MVC pattern for developing GUI applications in Visual Studio.

## Classes

Modern programs are built from classes. A running program consists of objects that work together to perform the task, which the program must solve. These objects are defined in terms of state and behavior, and you do that with classes, where a class defines or describes a particular kind of object. Perhaps the most important design activity is to find and define these classes, so they have the right properties. An object can be seen as an element of a program, a component that you can do anything with, and what you mainly doing during programming is to write the classes used to create the program's objects. A true and fortunate choice of classes has great influence on the quality of the finished program.

A program consists of many classes, and the classes that you primarily work with at design time, are the model classes, which are classes that define the essential data that the program should work with. This means that many classes first are defined later in the programming phase.

A different question is how to find the classes that the program should consist of, and there are several guidelines on what you can do, but no matter what, there are no instructions that necessarily give a good result. The classes has to model the application's problem area - and that alone - and must reflect what the program should consist of and work on, but it is important to remember that you are talking about model classes and thus classes describing data. Design of classes based largely on experience, and although through books about system development there are many recommendations for selecting classes the best way is to study other people's examples, as the examples in these books, and thus get inspiration from finished programs and the classes, which they are composed.

I will mention a single recommendation that you sometimes can see used to determine an application's classes. Based on the task-formulation and the requirements specification you can highlight all nouns, and then make a list of these nouns. They are candidates for classes. Subsequently, you can review the list and use it as inspiration for the classes that the program should consist of. Not all nouns represent a class. First, several nouns can describe the same concept, and secondly, some of the nouns instead describes the properties (variables) of the program's classes, and finally there will be names that do not relate to the program's problem area and therefore are not candidates for classes. It is a simple process, and it obviously cannot stand alone and requires many considerations for the candidates that have been found, but until you become practiced, the method is better than nothing.

A class consists (at design time) basically of two things:

1. data definitions that define an object's state, and thus what an object should represent
2. methods which define the behavior of objects, and thus what you can do with objects

The classes you find, must be documented. This can be done with diagrams (for example UML diagrams), but I will usually do it directly in the current programming languages. The reason is, that it is difficult to draw diagrams and cumbersome to maintain them (and the drawing rules can be so complex that it is in itself a challenge to draw the diagrams correctly). Modern programming languages and development tools have such good documentation options that you in most contexts can achieve the same documentation as you can with diagrams. However, I would not totally reject the use of diagrams (and the examples come). Diagrams used correctly can provide an overview, and so they do not require knowledge of specific programming languages.

## **Repository**

Most applications use persistent data and thus data to be saved between different runs of the program. It is a design task to decide how it should happen. In most cases, a database is used, and in that case, the database must be designed with respect to tables, relationships between tables, etc. The design of the databases are not necessarily simple, and there are very well-proven and effective techniques for the design of databases (see later in this book). When the design of databases requires many considerations, is due to an unfortunate designed database can have very large impact on the application response times - negatively.

There are also other opportunities where data is stored in regular files. It may be text files or binary files, and it may for example also be XML documents. It is often a simpler way and does not require that the program is connected to a database system, but if you have transaction-intensive applications with many updates and queries databases are the only sensible solution.

In addition to a program's repository - especially if there is a database - there are also other issues in the form authorization (login) and other security issues that must be clarified.

## **Algorithms**

An algorithm is the procedure for solving a specific problem. It is a solution method that addresses the solving of the problem through a finite number of steps, and the development of a program is to write algorithms. Above, I mentioned that a class basically consists of data definitions and methods. Here are methods algorithms. Methods perform something, and how the method has to do its work must be defined in the form of an algorithm.

In a program and its classes are by far the most methods (and the corresponding algorithms) simple, but some methods can be complex, where it is not clear how the algorithm should be written, and there will even often be several options, each with their advantages and disadvantage. The description of complex algorithms are an important part of the design, and you has to select the correct algorithms, but also to describe how the algorithm works.

Description of algorithms can be done in several ways. There is on the one hand formal algorithm languages, where you can write algorithms accurately, and so they are independent of a specific programming language. That means both that you should learn the algorithm language, and secondly there is a tendency for language and its rules are just as difficult to learn as a programming language, and many algorithm languages use a notation that is not very intuitive. On the other hand you can describe algorithms using a conventional structured language - often called pseudo code, and it certainly has its uses, but inversely with the disadvantage that it can be difficult to be precise enough, without the algorithm starts to get the character of an entire text document. A programming language is in itself a algorithm language, but were previously considered to be too detailed to be used to write algorithms at the design level. Modern programming languages is far more flexible, and by combining language and plain text in the form of comments, a programming language like C# is actually a quite effective method for describing algorithms. This is the approach that I everywhere will use the following as it provides a good opportunity structure, with sufficient flexibility to, at you at design time can describe algorithms at an appropriate level. I make no general requirements for the design of algorithms and what to include, just the design should be compiled.

### **The user interface**

To the extent that is required significant decisions about an application's user interface, it is also a design activity. It is concerning typically, the program's windows that may be outlined by means of drawings or actual prototypes. In particular, is the last interesting, because in Visual Studio you easily can create windows components, and it is similar easy to add application logic so the user can navigate between the windows. Such a prototype can be presented to the users and can be a great help to ensure that the developers has understood the task properly, and that there are not things that are overlooked, or there are functions that are missing. I have already mentioned that you can let prototypes be included as part of the requirements specification. Especially for large projects, it pays to do a comprehensive prototype, although it is only an empty shell, since it is a highly effective way to catch mistakes and shortcomings. The work on development of the prototype can be extensive, but a large part of the user interface can be used later during the programming, so it is by no means wasted.

## The documentation of the design

The result of the design must be documented and it can take the form of a text document in the same way as the result of the analysis. Especially if the design is documented using diagrams or there are important arguments for the decisions taken, a design document may be appropriate, but in general I would document the design using a C# project and often both. The Visual Studio project that contains the project (possibly several projects in the same solution) I will continue working with during the programming phase.

## Design principles

The design includes a program's architecture and classes, and there is established many principles of what in this context is a good design, and I will mention two, which is usually called coupling and cohesion, addressing both the program's architecture and classes.

One must strive that the program's classes has low couplings. Two classes are coupled, if they use the properties of each other. That is, if one of the classes calls a method on the other class, we say that the classes are coupled - the class that calls a method is coupled to the other class that is it knows the other class. In particular, be aware that if a class creates an object of another class, there is a coupling through the constructor. When couplings are bad, it is because it makes it more difficult to maintain the code. If a class has a method that others use, you cannot just change it, without it matters the other classes.

Now it is not such that you can avoid couplings. A program obviously cannot consist of isolated classes that have nothing to do with each other. The idea of a class is precisely that the class offers services that others need, but you want as low couplings as possible. It is difficult in general to say what it is, but you can be aware of the following:

- Couplings must always be done via properties (get- and set-methods) or other methods, but never via variables. This is achieved by always making a class's variables private, and so it is up to the class's programmer to define the properties and methods that are necessary for the class to make its services available.
- All properties and methods that represents internal services and thus services that should not be used by other classes, must be defined *private* - and exceptionally *protected*.
- You should program to an interface. Classes characteristics (services) should be defined by interfaces that the classes can implement. An object that uses a class, only has the knowledge of what are defined in the interface, but not how the properties are implemented.

- You must avoid couplings both ways. That is, where a class is using a service of a second class and the second class uses a service of the first class. Where applicable, it means that both classes must know each other, resulting in a very strong dependency between an application's classes.
- One should be aware that inheritance is a strong coupling. If the base class has protected properties they cannot be changed without this influences the derived classes. This does not mean that you should avoid inheritance, but simply to be aware of what it means for coupling.

Just as one wishes that a program has weak couplings, one would like the program's classes and modules have high cohesion. A class has high cohesion if it concerns a specific thing within the program's problem area. A class should not be a collection of methods that have nothing to do with each other. Sometimes one may think of it in that way, that a class representing customers and products have low cohesion. It has overall characteristics of the two things, and so it must be divided into two classes. Low cohesion leads to classes (or modules) that are hard to understand, and that in turn means that it becomes more difficult to maintain the code. You can note the following:

- High cohesion leads to many classes, but it is generally not a problem.
- If a class implements many interfaces, it points in the direction of low cohesion.
- If a class is composed exclusively of static methods, it must be methods that relate to the same subject.
- A package must include classes that relate to the same subject. Otherwise the classes must be divided into into multiple packages.

Both coupling and cohesion are general design principles that you must constantly keep in mind when designing an application's architecture and classes. Cohesion is rarely a major problem, but you should focus on couplings and constantly strive so weak couplings as possible.

## 2.4 PROGRAMMING

With the design in place, you can write the program's code. The programming starts with a copy of the Visual Studio project created during the design phase, and in principle it is about to implement the classes that are defined at the design phase. If you have implemented a good design the most of the important decisions concerning the program's architecture and core classes should be in place:

- all model classes are defined
- all important decisions about the program's windows and look and feel are in place




- the program's repository is defined
- all complex algorithms are located

However, there will still be many details that remain unresolved and are left to the programmer, but it is basic details, which are technical and geared towards the specific programming language. The programming phase is extensive and it is typically the phase that takes the longest time, but with a good design as a starting point, you have created the best basis to ensure progress in the programming and that the the code you writes is of high quality.

The next phase is called test, but the test also plays a crucial role during programming. It is the programmer's task to test the details, and ensure that the code does not fail. In simplified terms, this means that the programmer should test individual classes for errors, so that you in the following work can be sure that a class is a finished and tested component. This can be done in several ways. One can draw up actual unit test where classes are tested (se later in this book). You can write small test programs to test each class, and finally you can perform inspection of code using the debugger. Also the inspection in connection with the documentation of the code, as you, while documenting the code, automatically will consider why the code is written as is, and possibly sees inexpediciencies and errors.

It is actually difficult to test the program code, and whether it is in one way or another way, it is difficult to ensure that all cases are handled. It requires that you are careful, and



**Do you want to make a difference?**

Join the IT company that works hard to make life easier.

[www.tieto.fi/careers](http://www.tieto.fi/careers)

Knowledge. Passion. Results.

**tieto**

it requires that you estimate that it takes time. However, you can be sure that the time spent during programming to test is well spent. It is much more difficult and more time consuming to find a mistake that is recognized at a later time, for example in the subsequent test phase, or even worse after the program was put into operation.

Above I mentioned documentation of the code and code inspection. Documentation is also part of the programming phase. It is important for several reasons than inspection since it is all the necessary information for the people who later must maintain the program.

### **Maintenance of the documentation**

The waterfall model is a progressive process from task-formulation to the finished program, but no matter how careful you are, you cannot avoid that there are changes on the way. For example it may happen that during the programming are recognized any further claims or already found demands that must be changed. In the same way it may happen that you have to realize that the already adopted design decisions need to be changed, and later again during the test, the same can happen. The question is what to do with documentation respectively from the analysis and the design, and whether it need to be updated.

In principle it should, but I'm a little shaky here. The documentation must document the decisions about the development of the program and how the program is made. In case of changes that are so big that what it says in the documentation, is totally wrong or misleading, then it of course must be updated. Otherwise, the documentation is impossible useful. Now it is far from always the case with such radical changes and have you made a good analysis and design work, should it be the exception. Is there rather minor adjustments, and they will certainly be there, you can document the changes in the code. You can write documentation comments in front of each class, and in some cases you can even add text documents to the project, documenting the changes. In most cases, I prefer to document changes in the code, instead of to go back and modify the documentation from the analysis and design.

## **2.5 TEST**

As mentioned above play test an important role in programming, but the waterfall model also has an actual test phase. The goal here is to test the finished program in an environment that resembles the future operational situation and in this context, test whether the program meets the requirements of the specifications. It is a phase that usually will involve the customer and the future users to find inconveniences and errors before using the program for the task it is intended.

The extent of the phase is of course determined by the specific program, and in larger projects, the extent being large and take time. You should be aware that simple testing, where you let a few users try out the program, rarely finds many errors. It may be part of a larger test, but the test requires planning where to exactly decide what to test and how. In particular, it may be difficult to establish an environment simulating the daily operation conditions. To the extent that the program is included in or using other systems, communications and data exchange must of course be tested, and it is rarely possible to perform any tests before the program is finished.

The result of the test phase will typically be a to-do-list, which lists the errors and discrepancies found. With this list available the programmers may correct the errors and omissions, and then the program must be tested again to ensure that the corrections not have introduced new errors. This process continues until the test results in an empty to-do-list, and then the program is ready for use.

The test phase is a follow-up to the test, as during the programming, and must the phase make sense, it is crucial that the testing effort during programming is done carefully and accurately. It is important that the program be transferred to the testing phase, is of such a quality that it is worth testing on. Actually it is not unusual for a program to leaves the programming phase with too many errors, and the result of that is that the test work has to be stopped immediately, and the program is returned to the programming team called unusable. If this happens, you have only succeeded in having the people who need to test the program, have used their time to no avail, and the programmers cannot do much other than to address the errors without knowing much about what to look for. So there is reason to once again emphasize the importance of the test, which takes place as part of the program development.

## 2.6 DELIVERY

The final stage will really take the program in use and thus hand it to the future users. It is difficult to precisely mention the content of this phase, so instead I will give some examples of tasks that the phase may include:

- Development of an install program or script.
- Upload a program to an App Store.
- Hosting a program or service on a web server.
- Create a database that the program can connect.
- Create users, possibly with varying rights.
- Customizing the configuration files to applicable safety policies.

- Conversion of existing data to a new format.
- Training of future users.
- Evaluation of the project in order to gain experience.

There may be mentioned more tasks, but the above should suffice to suggest that much can be back after the program is written and tested.

### 3 A SYSTEM DEVELOPMENT METHOD

In this book, I have mentioned the so-called waterfall model, which is a method to system development. I have described the waterfall model as a system development method consisting of 6 phases, and there are other descriptions of the method, some of which are simpler and others more extensive, but common to them is that they describe the system development as a course

analysis – design – programming – test

These are four key activities in any system development method, but in many contexts, particularly in the development of larger IT solutions, it is a simplistic view on system development. Firstly, experience shows that it is unrealistic from the start to establish the final requirements specification and then develop a design for the program. It is necessary to work iteratively and develop a part of the program that can be tested by the future users before proceeding with other parts towards the finished program. This means that you work with analysis, design, programming as to return and continue with additional analysis activities and so on. Second, the system often - at least for larger solutions - include many other activities and, finally, the programs to be developed are so different that a method like the waterfall method is too limited to accommodate the development of all kinds of programs. Thus, it is too simple to carry out a system development using a number of stages carried out in a particular order. There is a need for more flexible methods that can be adapted to the given task and situation.

The waterfall model, however, has its uses and benefits, and here I would mention:

- The model is simple – there is not much to learn.
- The method describes and applies the basic activities in the system development.
- The method is better than nothing.

The conclusion is that the waterfall model is very applicable for the development of small programs, which have not many uncertainties and the model is used in the next books, but with the important addition that it is adapted by my way to the development of a program.

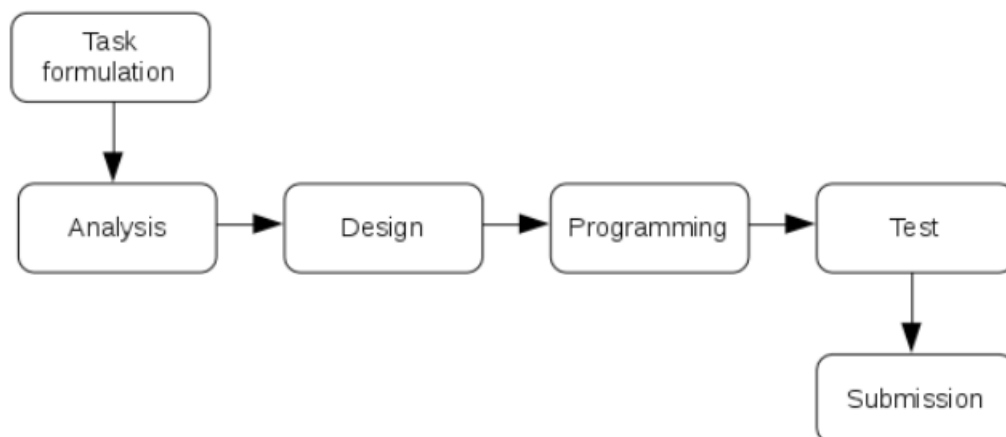
As mentioned, I below will look at other system development methods, and I also mentioned that these methods practical success is somewhat variable and ranges from developing programs without at all using a method, to developing the department's own system development

method based on experience and current tasks and inspired by specific theoretical methods. Although the system theory and method has been established as part of IT education for many years, the methods are not always applied in practice, and there is in my opinion for two reasons:

1. The methods have been perceived as a waste of resources.
2. The methods have required too much documentation.

Are these the reasons for not to use system development methods in practice, they are both wrong, but has the reliability that one should seek methods that limits the documentation.

The above is not really a system development method, but rather some guidelines that you can follow to develop a program. It is not a system development method, because there is no precise guidelines on what to do, but the goal is that it can become so if you compares the contents of this book with the following examples and thus work out best practices for application development. The result is that I shall want to consider a system development project as a process consisting of six activities:



### 3.1 OTHER METHODS

All the previous books in this series have been about system development and how to write programs using the C# programming language, and if I look backwards, it is probably the last that fills most. It is actually not fair because if you are going to be a good software developer, you should strive to free yourself from the tools and instead focus on the methods and principles common to all system development tasks. However, it is not so easy, because in the end, one must necessarily become concrete and learn how to work with the technologies and tools needed to implement solutions practices. In fact, it is a bit of a dilemma in software development, where you have always been emphasized the importance of learning good

system development methods and focusing on analysis, design and patterns, and where the technology and programming in some language was secondary. As mentioned above, the intentions of this series of books were to write about software development, but with the emphasis on the process, and perhaps the final examples should have been more important than the case is. Much of the foregoing has focused on C# and details, but maybe it does not matter, because that's what it's all about.

We have gradually developed software for over 50 years, and during that time many new programming languages have been developed, many different development tools, and several system development methods and other guidelines for developing a program has also been developed, so in principle, it should be easier to develop a program today than before. It also is - you simply have better tools than before, but conversely, the programs you are developing today are often much bigger and more complex than before, and also they have a greater diversity. For example, there is a big difference in writing a PC application and a web application, and again it is different to write an app to a mobile phone. In particular, the last means that it is hard to be good all the way around and that it is necessary to specialize as software developer. Finally, new tools (including programming languages) are constantly being marketed to address specific tasks, so there are always new tools to be used to and use in the daily work as a professional software developer.

Fortunately, there is also something that is stable, such as algorithms and other solution techniques, and this is where you have to stay focused. In fact, programming languages and other development tools do not mean much. If you have first learned a programming language, it is easy to change to another language if needed. This is often encountered, typically because different development organizations do not use the same tools and because there are always new ones that might be particularly suitable for the type of tasks the current development department is working on. It is far from the fact that, as a developer, you have influence on the choice of tools, as they are often based on basic decisions in the organization, but you are influenced by the need to be critical and test the tools thoroughly before using them in the specific projects. Those who have elaborated new tools often promise a lot, and although there are also new and good tools on the market, it is important to be aware that it takes time and thus has the cost to take new tools in use. You have to consider the context of whether it is no better to use the time to become even more adept at using the tools that you already do, and adjust and refine the use of the known tools. Often, new tools promise a lot, but when you first dive into them, it is far from the assumption that the news is so big.

The primary tools in these books are C# and Visual Studio, but there are actually many other tools that relate to the C# world, and all have the same purpose to streamline the software development process. Among other things, there are tools that can interfere with

Visual Studio. Many of these tools are good and are widely used in practice, but although there is no room for a presentation in these books, I will mention project management tools (actual a part of Visual Studio). There are many, each with their advantage, but they are tools that support, the implementation of large projects with many project participants. They have facilities to manage large projects where multiple developers work closely together, including controlling versions, so that you can always go back to a previous version, and so some developers will not override others' code without being sure that you can recreate the code (in such situations my solution with sub folders and many project versions will in no way works). It is comprehensive tools, that it takes time to learn to use, and partly are either or tools. These are tools that are implemented on the basis of a decision in the development organization and after extensive investigations, which all have to undertake and apply. On the other hand, they are tools used by all major development organizations, and thus tools that you will certainly meet, and it is a must in the development of large and complex IT solutions.

As a software developer, you must know your tools and development methods, and apart from C#, none of the subjects are presented in these books, but below I want to mention three system development methods:

1. Unified Process
2. SCRUM
3. Extreme programming

None of the methods will be discussed in details, and I would recommend that you find literature that explicitly addresses each of the three methods. In particular, SCRUM is a very popular system development method, although the two others certainly also have their followers.

## 3.2 CODE AND TEST

It is not a system development method, but it is mentioned here because many programs in practice are developed according to this model, and it also has the right to small programs. The method is quite simple where the developer starts to understand what it is for a program to be written, and then the developer programs some code and tests the result, reprograms, and tests again, and continues until the program is written.

You can say that you actually work without using a method, and it is contrary to everything that has been said and written about system development, but yet the method is widespread, and under certain assumptions it also gives good opinion in practice.



Of course, you cannot tackle the code and test process before you understand what it is for a program you has to write, and the process therefore starts with a form of analysis where you can clarify the requirements of the program with the customer. Then you start programming and testing whether the code you have written works according to the requirements, and so proceed with a large number of small steps until the task is solved.

In order for such a simple approach to lead to a successful outcome, there are two prerequisites that must be met:

1. It must be a small project and a project solved by a single person (such as the final examples in these books).
2. The task must be well-defined and there must be no greater uncertainties about how the task should be solved and what the final result should be.

If these prerequisites are present, there is nothing to solve the task after a code and test model, and in reality there may not be much else that makes sense. Here it is important to note that in the real world there are many tasks that just meet the above two assumptions, so code and test programming has its uses and has been used far more frequently than the many books on system development prescribe.

### 3.3 UNIFIED PROCESS

*Unified Process* is an object-oriented system development method, but in the literature, the method is often termed the *Rational Unified Process*, which is a further development of the original method. It is an iterative system development method, solving the current task by working iteratively towards the solution through 4 phases:

1. Inception
2. Elaboration
3. Construction
4. Transition

One has the greatest success with unified process as a method for larger projects, where the task is relatively well defined and where the project is associated with more developers. In many ways, the method can be seen as an alternative to the waterfall model, recognizing that this method is not suitable for the development of large and complex IT solutions. It is an iterative method in where the task is solved through a number of iterations, and one can reasonably perceive each iteration as the partial solution implemented after the waterfall method.

The unified process is also an agile system development method, and thus a method where the goal is quickly to deliver value solutions for the customer, and at the same time after customer's testing of the delivered sub-solutions, it can easily and accurately adjust the requirements according to the customer's wishes. All modern systems development methods today are call agile, and it is perhaps not much more than an observation that today's IT systems are far larger and more complex than before.

I do not want to review the method in detail, but I strongly recommend reading a book or something that presents the method and provides instructions on how the method in practice can be used.

As mentioned, the method comprises 4 phases, and each phase is performed as a number of iterations. How many iterations to be in the individual phases depend on how complex the task is, but the whole idea is, that each iteration should not take too long. It is recommended that an iteration should have a deadline of not more than a few weeks, but conversely, an iteration should not be too short and should at least take several weeks. Before starting an iteration, you have to define which tasks should be solved. These tasks depend on the phase, but in principle, each iteration can include tasks as analysis, design, programming and test, however, such that the weight changes through the course, where an iteration during elaboration largely focuses on analysis and design while the tasks under construction at most emphasis on programming and test.

## **Inception**

This phase will usually consist only of a single iteration. It is a start-up phase where you must analyze what it is for a task to be solved. It is not the goal to prepare a completed requirement specification, but inception is a short phase in which you analyze critical requirements and determine the basic visions of the system. You should not try to make a detailed list of as many system requirements as possible. A risk analysis for the development of the system must be carried out and a decision must be made whether or not to complete the project. The objective is:

- to gain an understanding of what it is for a system to be developed
- describing the most important functions, and often as use cases
- to develop an overall design of the system's architecture
- to develop a development plan for the subsequent phases, including identifying the project's costs and risks

Any system development starts with an idea that is presented to the person(s) to solve the task. During the inception phase there will be relatively few people. The system development project starts with the presentation of the idea, and the phase will provide guidelines for how the system development group comes from idea to the completed solution that the task manager (the customer) can apply.

The first phase is a clearing, phase and is a short phase and is typically performed as a single iteration. How long does of course depend entirely on the specific task, and maybe the phase takes only a few days, and I find it hard to imagine an inception phase that spans more than a few weeks. During the start, the project team must clarify the following:

- What is the task going on?
- What is the technical platform?
- What risk factors are there?
- What is the time horizon?
- What resources are required?
- How is the project plan?
- What are the criteria for the task being solved?

The result of the inception will typically be:

- A task formulation
- A requirement specification
- One or more prototypes
- Attachments
- Conclusion

A task-formulation is a text document organized in relation to the relevant of the above headings. The aim is to keep the assignment form brief and accurate, and the task formulation is perceived as a document that is continuously updated after each of the following iterations.

As mentioned, it is not a goal to prepare a final requirement specification during the inception phase, but you will start it. Often you choose one or a few of the issues that the system will solve and start formulating the requirements. This is typically done in the form of use cases, and it is even said that the unified process is use case driven. During start-up, focus will be on the most central issues and the issues with the greatest risks.

The task formulation may be supplemented by one or more prototypes that should indicate to the task manager the important decision regarding the finished project's user interface.

Often it is a good idea to have important decisions regarding the user interface in place before continuing with the next phases.

Finally, the inception may include attachments or references to sources that are important for the following system development.

The inception phase concludes with a conclusion where the system developers emphasize the consequences of continuing the project, and possibly highlighting proposals for adjustments. If the result becomes a decision that the project is to proceed, a project plan is prepared, including a planning of the first iterations during elaboration.

## **Elaboration**

In this phase, one or more iterations are performed (and usually there will be more and maybe many), in which it will develop a part of the overall system in each iteration. One can thus think of an iteration as a mini development project, and the result will basically be a program (a product) that the task manager can test. One can think of an iteration as a development process that brings the entire project from one state to another.

An iteration should take shortly in time, but conversely, it should also be so extensive that the customer experiences progress. An iteration length is determined of the current project, but the timeframe will typically be from two to a few weeks. Of course, the number of iterations will also be determined by the scope of the project, and in small projects, there will in principle only be a single iteration, but if you use unified process, there will usually be more.

As mentioned, an iteration is a development process, and it will consist of

1. analysis
2. design
3. programming
4. test

It is not a fact that you in an individual iteration follow a waterfall model, but more typically you will work iteratively, where you work with analysis, design, programming and then return to some analysis tasks.

The result of an iteration is:

- The project is in a stable state, which can be presented to the task manager or customer. It will usually be a software product, that the task testers can test and comment on.
- An updated version of the task formulation with regard to decisions taken in connection with the implementation of this iteration, as well as an extension of a section with important results for the current iteration.
- An updated version of requirement specification based on the analysis performed during the iteration.
- A conclusion that includes the customer's remarks after testing the product and the developers' recommendations regarding the continued system development.
- Adjustment of the project plan, including planning of next or the few next iterations.

Looking at the above, it may seem that you work without having a schedule and decides what should happen as you move forward in the project. Such, of course, you cannot develop IT systems, because the task manager or customer must know when the system is expected to be delivered and what it will cost, but vice versa, it should be noted that when unified process continuously adjusts the requirements and continuously plans iterations, it is exactly, what's means that it's an agile method. This does not mean that you are working without planning, but it is a recognition that for large and complex systems it is impossible to describe a completed requirement specification and estimate how long the development will take and what it will cost, and the question is, what you then can do it. The idea of unified process is that you continuously work through the requirements specification and the project plan so that after the completion of elaboration, the requirements specification is complete and including the final price for what the solutions cost and when it can be delivered and applied. Until then, it must all be based on estimates that are continually underpinned and adjusted, and that is exactly the dilemma of system development. However, in the unified process, the goal is that after elaboration the requirements must be in place, as well as the completed project plan and project requirements for resources.

As an addition to the above, I would like to mention that when it is so difficult to estimate a system development project, it is not only because developers are bad planners, but equally because the task manager and customers of large and complex IT solutions cannot formulate the task. It requires a process of continuously identifying the requirements and deciding what to do, and what should be delineated, and remember that delineation may also be postponed to later. This is exactly the process that elaboration deals with.

The central concept in unified process is an iteration, which typically has a time span of about one month. As mentioned, an iteration must bring the system from one stable state to another and it is important that the result of an iteration is a step, so the task manager experiences a progression through a presentation of the result. The planning of the individual iterations starts during the inception phase and continues throughout the elaboration phase, with continuous being expanded with new iterations. For each iteration one must

1. plan what the iteration should include
2. and when the iteration must be completed

With regard to the first, the iteration must include a well-defined work of the entire project process, and the iteration should typically end in a month and should rarely last much longer. More people may work on the same iteration, but there will be few. Another question is what should happen if the iteration is not completed on time. Here it is an imperative requirement in unified process that the iteration is to be terminated on time, and what may be missing has to be postponed to a later iteration.

## **Construction**

It is the longest phase in time. It consists in the same way as elaboration of a number of iterations, where each iteration brings the project from one stable state to another. There is no sharp boundary between elaboration and construction, but the difference is that at the start of construction, the remaining iterations will be planned and under the construction phase, the individual iterations will only contain to a limited extent analysis and, and to a lesser degree, design. One can think of it that when you get to the construction phase, all requirements are formulated and are in place (the requirement specification is complete) and all important decisions and risk assessments are clarified. Therefore, programming and testing are primarily what lacking.

The construction phase is usually extensive in time as many tasks can be deferred to construction. It is also typical that each iterations under construction to a greater extent than during the elaboration can be carried out in parallel, where several developers works on each their iteration.

The result of an iteration is in principle the same as during the elaboration phase, but with the difference that there will typically not be updates of the task formulation and the requirement specification and in worst case only minor adjustments. In addition to the product, the result is just a short and accurate summary of the current iteration and including the results of the iteration's test.

One can also think of construction as a phase, where the project has now moved into a stable and planned course, through which the project moves forward towards the finished product through a carefully planned course. At that point, the risk is gone and there is only “hard” work left.

### **Transition**

The last phase is also an iterative phase, but typically there are few iterations and often only one. It is the delivery phase and the tasks can vary a lot, but are the same as I above described as the last phase in the waterfall method. The planning of the phase’s iterations takes place during elaboration.

## **3.4 SCRUM**

Everything that is said in these books about system development relates solely to the developers’ specific tasks, including how the developers can go forward to write and test the individual programs. Of course, it is also important, in one way or another it is, what it is all about, but for large projects where a team of developers collaborates to develop the current system, it is necessary with project management, and one of the reasons why IT projects sometimes goes wrong is actually lack of management. Over time, several project management methods have been developed, and one of those who has been very successful is called SCRUM. In fact, the method is not directly aimed at system development of software, but the method has proved particularly suitable for managing IT development projects. The following is a brief presentation of SCRUM.

One of the principles behind SCRUM is the recognition that, due to a number of uncertainties, it is not possible in advance to define all requirements for a system, thus completely the same as what I have mentioned above during unified process. It is therefore necessary to have a project management method that is flexible enough to handle the development of a system that is characterized by continued change of the requirements throughout the development process while ensuring progress in the project. Such system development processes are called agile, and SCRUM is a project management method for agile projects, and is the most widely used project management method in practice.

Now SCRUM is not always the right method, and for SCRUM to makes sense, the project must be of a reasonable size and be a project that should be solved by a team of developers. In addition, it usually has to be a project characterized by many uncertainties and complexity, and where there is a need for agile methods, but can a project to some extent be characterized in that way, SCRUM has proven itself as an effective project management method.

## The elements in SCRUM

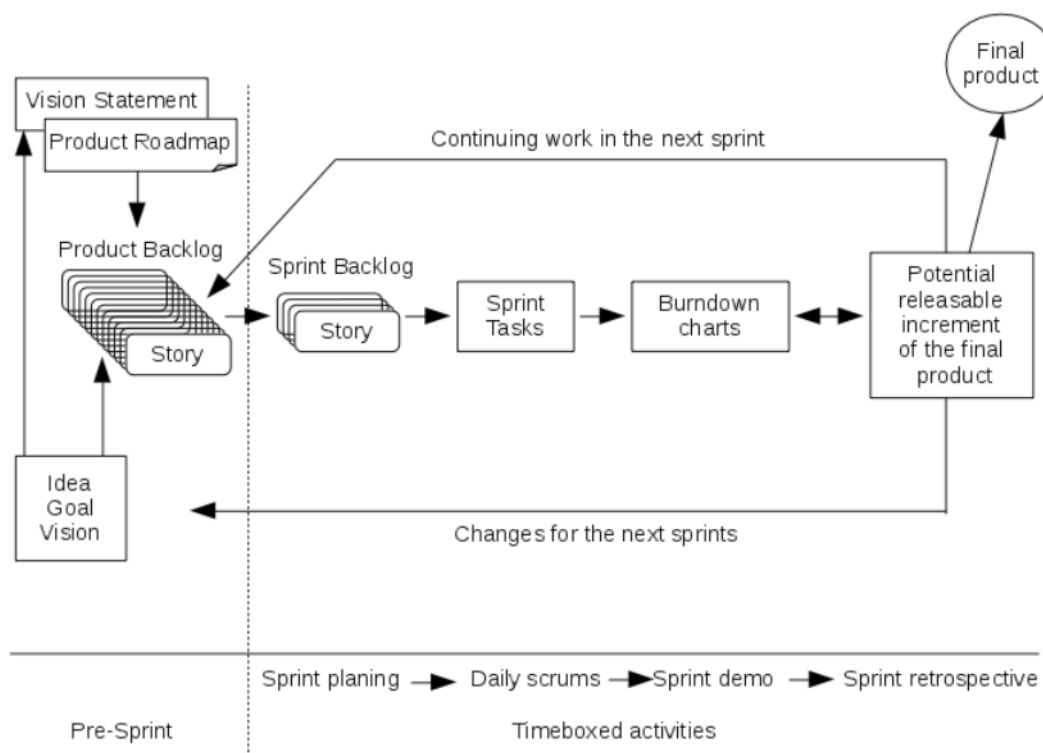
Basically, the principles of SCRUM are to divide a project into a series of small iterations, which are called *sprints*. The starting point is that it has already been decided to develop a system (and in this book an application), and the task manager or customer already has a relatively accurate meaning of what it is for a system to be developed. The project starts with the development of two documents:

1. *Vision Statement*, which gives a brief description of the goals of the project, which in the course of the project process will help the project team to focus on what is important seen from the task manager.
2. *Product Map*, which is an introductory schedule for when important sub-products are delivered.

The project process is divided into two phases, called respectively

1. Pre-sprints
2. Time-boxed activities

where you can think of the first one as planning and the other as production, but how far the largest part of time is used in the final phase. In fact, phases are not the right word, as it is not so that you are either in one phase or another, but it all starts with *pre-sprints*, after which you perform a series of sprints and eventually (and often) return to the pre-sprints phase. You usually outline the SCRUM with a figure similar to the following:





During pre-sprint, one focuses on injecting the user requirements and describing them as tasks to be solved. These descriptions are called stories. The result will typically be a large collection of stories, called *Product Backlog*. Typically, you do not complete Product Backlog before starting the individual sprints, and as soon as you find that enough information has been gathered that some stories are well described, you can start the first sprints. The goal is to get something done that end users can relate to and evaluate, and then to return to the pre-sprint phase and possibly update Product Backlog.

Looking at the sprints phase, it includes a sprint planning for every sprint, a meeting where you plan what stories from Product Backlog to be transferred to *Sprint Backlog*, and thus what stories to work on in it a following sprint. A sprint is scheduled to take a certain period of time, which may vary, but typically takes a sprint of about 4 weeks, and it is recommended that a sprint should not take longer than one month. The team breaks down the selected stories for tasks that are then delegated among the team's members. During a sprint, daily meetings, called *Daily Scrum*, are held, where each team member reports briefly the progress (or the opposite) in connection with the current tasks. These Daily Scrums are short meetings, and may take up to 15 minutes. At the end of a sprint, a meeting called *Sprint Demo* (or *Sprint Review*) is held. This meeting is held together with the task manager, where the team demonstrates the result of the current sprint and thus how the individual stories are resolved. The entire sprint finishes with a *Scrum Retrospective* that is a meeting that is a review of the sprint, where the team looks back on the course for improvements.

In connection with a SCRUM project, there are three roles (and there may be no other roles), and these roles are the completed scrum team:

1. *Product Owner*, which is one person who is part of the project on full-time or on part-time, and which represents the business area within the project is anchored.
2. *Scrum Master*, who is one person who is full-time or on part-time on the project and who is a coach and facilitator for the development team.
3. *The Development Team*, which is 3 - 9 full-time employees (also called specialists) and is part of the development organization as those who perform the individual sprints.

In addition to this, there may be other stakeholders, who typically represent the customer or his organization.

It is assumed that the team meets two basic assumptions that are considered necessary to ensure flexibility, creativity and productivity and are considered necessary for the agile environment:

1. The team is self-organizing and plans its work without explicit leadership from the outside. One or more people in the team can be in charge of management while other participants are solely responsible for special specialist activities, but in SCRUM there is no division of management and specialist functions.
2. The team participants have all the necessary expertise and competencies needed to get the task solved and without the need to get help from outside.

### **The Product Owner**

Any project needs a person familiar with the business area whose task is to maximize the value of the team's work and it is this person that is called *Product Owner*. Generally, the person comes from the development organization. It is important to be aware that the Product Owner must know the problem area, but the person does not need to have special knowledge about system development. It is the Product Owner that is responsible for the project's Product Backlog, which is a priority list of stories to be solved and it is the central planning tool in SCRUM. Stories are prioritized in terms of their value for the user organization, so that high-quality stories are solved first. It is also the Product Owner who is responsible for that each user story being easy to understand for the Scrum Team and other stakeholders.

It is the Product Owner who communicates with the task owner and user organization and uses this information to update the project's Product Backlog. It is also the Product Owner who has an overview of the progress in the project and keeps the project stakeholders updated with regard to the final end date.

The entire organization must respect Product Owner decisions, which is considered a prerequisite for the success of the project. Even Product Owner's chief may not generally correct Product Owner decisions, and no one other than Product Owner may decide what the development team is supposed to deliver. However, it is permitted for the Product Owner to delegate decisions to the Development Team, such as adding items to Product Backlog, as long as the Product Owner has the full overview.

### **Scrum Master**

*Scrum Master's* task is to assist the Scrum Team by coaching them and ensuring that all principles of SCRUM are respected. Scrum Master must therefore have in-depth knowledge of SCRUM and the principles of the method. It is a management function, but it is not a

project manager. Scrum Master will lead the SCRUM process, but not the Scrum Team. In addition to ensuring that the Development Team understands and uses SCRUM correctly, it is also the Scrum Master's task to remove obstacles to the Development Team, ease everyday life and generally coach the participants.

Scrum Master must also support Product Owner to pass the right information to the project's stakeholders and generally provide expert knowledge available to the *Product Owner*.

It is allowed for a person to be both *Scrum Master* and a member of the *Development Team*, but it is not recommended.

### **The Development Team**

Members of the *Development Team* are IT experts and are responsible for delivering solutions to the elements of the backlog. Members generally have knowledge to do everything from A to Z in connection with the development of a task from the backlog, and they must be able to work independently themselves, and be able to solve the challenges that arise. One assignment can be awarded to a particular member through a sprint, but the entire *Development Team* is responsible and accountable for the task and there is thus no single person who owns a particular task.

The Development Team delivers the finished product developed through an iterative process managed by the project's Product Backlog. It is highly recommended that team members work full-time on the current project, and efforts should be made that the Development Team members not being replaced along the way, as it is considered important to ensure that members are focused and agile.

Experience has shown that SCRUM is most effective with a Development Team of 3 - 9 members, but for large projects it is possible to establish more *Scrum Teams*.

It is important to note that members of the Development Team do not have titles, such as designers, architects, managers, etc. Everyone must have the same role, and the title is only a member of the Development Team. The whole idea behind SCRUM is teamwork. The reason is that the members will otherwise focus on their own roles and not sufficiently focus on the finished product. Each member is responsible for all project results.

## The sprints

Then there is the content of the individual sprints, where the actual system development takes place. A system development project managed by SCRUM solves the task through a number of iterations, called sprints, where a sprint is time-boxed and takes a certain period of time. A sprint contains 4 activities:

1. *Sprint Planning* is the first activity in a sprint, where the Scrum Team plans what will be the result of the sprint and what to deliver.
2. *Daily Scrum*. The Development Team starts the work immediately after Sprint Planning is completed. During the sprint, the Development Team keeps daily meetings, called Daily Scrum. It is a short meeting of approximate 15 minutes where the work for the next 24 hours is coordinated. As part of SCRUM it is recommended that these meetings be held standing.
3. *Sprint Review*. Before the sprint finishes, the Development Team presents the sprint's results for the user organization and retrieves feedback.
4. *Sprint Retrospective*. After Sprint Review and as the last in the sprint, the Development Team holds an internal meeting as a kind of review of the sprint in order to learn from the course.

Time boxing is considered to be central to SCRUM, and is the means to ensure that things are done, and for a project, each sprint should generally have the same fixed length. The length of a sprint is typically about one month, but may be shorter, but hardly for a week. The product is finished after a number of (and possibly many) sprints, and each sprint brings the product a step toward the finished result. If the result of a sprint is a potentially releasable part of the finished product (what to aim for is the result of each sprint), it is said that an *increment* has been reached, and an increment is thus the sum of all the elements in *Product Backlog*, which has been completed so far in the project. One can therefore consider each increment as an update of the previous increment and an extension of the product with new functions and features. An increment may not be applied, but in case of an increment, it should be possible.

Typically, an increment (result of Sprint Review) requires changes, and they will then be added the Product Backlog as new stories.

Each story in the Product Backlog should usually be developed in a single sprint, as it makes it much easier to keep an overview. Product Owner and the Development Team selects a number of stories from Product Backlog for each sprint, with the goal of getting them completely completed in the upcoming sprint, so after the sprint the team has a new increment. It is therefore important that from the start you have decided when a story has been completed.

## Sprint Planning

Sprint Planning is a time-boxed meeting, where all team members participate as a starting point. If it is a sprint of 4 weeks, the meeting may last 8 hours, but at short sprints the meeting will be shorter. Participants in the Development Team will estimate how many hours they can deliver in the sprint. Prior to that, the Product Owner prioritized the individual elements of the project's Product Backlog and ensured that each story is easy to understand. The Development Team then selects an appropriate number of items from the top of Product Backlog and moves them to Sprint Backlog as the work to be performed in the current sprint. The work for each story is estimated by the Development Team and the total work for Sprint Backlog tasks should as best as possible correspond to the capacity available to the team.

Then, a *Sprint Goal* is written, which is nothing else a short text describing the target of the current sprint, including what are the criteria for the sprint to finishing and what the tasks are solved. Sprint Goal must also contain at least a detailed plan for the first days of the sprint. Sprint Goal is part of Sprint Backlog, and there are no special requirements for documentation or form, and often a board is used, which makes it easy to maintain the plan, and ensure that the plan is visible to everyone. Sprint Backlog is thus made up of

1. *Sprint Goal*
2. The stories selected from Product Backlog for the current sprint.
3. A detailed work plan for the sprint, where the tasks (stories) can be categorized as *To Do*, *Doing* or *Done*.

In principle, a sprint cannot be interrupted, but Product Owner may, as a sole exception, decide to interrupt a sprint typically caused by events / changes in the project's environments / prerequisites.

## Daily Scrum

As its name says, it is a daily meeting held throughout the sprint period. At the meeting, only members of the development team participates. The meeting lasts no more than 15 minutes, and at this meeting each team participant must answer three questions:

1. What has been completed since the last meeting?
2. What will be done before next meeting?
3. What obstacles / difficulties are there?

Team members must compare the progress with the Sprint Goal, but should also estimate the likelihood that a task will be completed before the sprint is complete.

## **Sprint review**

It is a meeting to be held at the end of a sprint and the meeting will take for a sprint of 4 weeks max 4 hours, but for a shorter sprint the meeting will usually be shorter. The meeting includes the entire Scrum team but also other stakeholders, and at the meeting the solution of all tasks from the current sprint is presented. The goal is to get feedback and as early as possible to adjust the requirements so Product Owner can update the project's Product Backlog.

It is the Development Team that presents the product, but only for tasks that are *Done* and not for tasks that are “almost” Done. It is the Product Owner who before the meeting must ensure that the products presented also are Done.

At the end of the whole sprint, another meeting called Sprint Retrospective is held, which is a meeting of 2-3 hours. It is a meeting where you look back on the sprint for improvements, and it applies to both the working procedures for the individual team participants, processes and the tools that you use.

## **More about SCRUM**

An activity that in principle always takes place in parallel with the individual sprints is called *Product Backlog Grooming*. It is always about reviewing and adjusting items in Product Backlog, and typically includes adding details, adjusting estimates, and so on. It is Product Owner who is responsible for prioritizing the items in Product Backlog, and it is the Development Team that is responsible for estimating the items.

Although it is not part of SCRUM, it is recommended that you plan a day off or two between each sprint, which you can use to read articles, attend courses or workshops or otherwise orientate yourself to new subjects, or you can simply go for a walk and enjoy the nature. Perhaps this idea may be difficult to sell to a manager, but studies shows that the time is actually well done, partly in order to make progress in the project and partly to ensure that team participants stay up to date with the technology development. The idea supports SCRUM, where it is absolutely crucial that you are constantly product-oriented rather than activity-oriented.

Another question is what you do at the end of a sprint if you find that some of the work is not done. Here the rules are quite accurate:

1. Unfinished or undone elements (stories) are placed back on the Product Backlog.
2. The Product Owner re-orders the backlog, where typically unfinished elements are placed at the top, but not necessarily.
3. A story that is undone must not be resized to represent only the remaining undone work. The story that was worked on must not be split in and what was done and what was not done. A story is done or not.

In general, it can be said that SCRUM includes the following artifacts:

1. *Product Backlog*, that is an ordered list of all stories that might be needed in the final product
2. *Sprint Backlog*, that are selected stories from Product Backlog to be delivered through a sprint, along with the *Sprint Goal* and sprint plans for all stories in the Sprint Backlog
3. *Increment*, that is the set of all the Product Backlog stories completed so far in the project
4. The definition of *Done*, that is the understanding of what it means that a piece of work to be considered completed
5. *Monitoring Progress* towards a goal, that are the performance measurement and forecast for the whole project
6. *Monitoring Sprint Progress*, that is the performance measurement and forecasts for a single sprint

You should be aware that the literature regarding SCRUM defines the exact content of these artifacts and also typical tools that you can use.

As a last comment on SCRUM, it is important to emphasize that the goal is to provide a framework for how you effectively can organize the development of a larger and complex (IT) system, and the method expresses the experience of professional developers in implementing complex development projects. On the other hand, SCRUM does not say anything about what the individual sprints should contain and how each team participant should perform the daily work. Here you must apply to what else you have learned about system development (for example, something of what is explained in this series of books) or better what is illustrated in the following chapter on extreme programming. In practice, you will combine a variety of system development and programming techniques.

### 3.5 EXTREME PROGRAMMING

*Extreme Programming*, also called XP, is a system development method that, as the name says, unlike the methods mentioned above, focuses on programming. You should not perceive XP as an alternative to, for example, Unified Process or SCRUM, but rather a recommendation for how to successfully complete the programming part of the individual iterations. XP, like the other system development methods, consists of a number of iterations where each iteration brings the project from one state to another. There are the same activities as in other methods, but in XP, the three activities design, program and test together form a single activity.

The method is based on four main principles:

1. *Heavy customer involvement*, where a representative of the user organization is included in the development team, and continuously participates and actively contributes to the planning of the individual iterations and the various accept tests and to a large extent also conducts these tests.
2. *Test-driven development*, where developers are required to write unit tests for each new feature before any of the code is written. In this way, a function cannot be transferred for testing before all unit tests are performed with success (see the next chapter on test-driven development).
3. *Pair programming*, where all code is written by two programmers, called a *driver* and a *navigator*. Both programmers are at the same computer. The driver writes the code while the navigator monitors the process, comes with suggestions, thinking about design and program quality and so on. Every half hour, they switch the two roles. The idea is quite simple, that two thinks better than one. Pair programming is of course less effective than two programmers working on each their code, but experience shows that code produced is of better quality and with far fewer errors.
4. *Short iteration cycles and frequent releases*, where you develop software in short iterations that do not extend beyond a few weeks. The goal is frequent releases that can be handed over to and in principle taken in use by the user organization. A release period spans no more than a few months and is performed over few iterations. The idea is that the user organization must experience progress and that something happens.

As with all other system development methods, the goal of XP is doing better, and XP rightly considers the biggest issue in software development as risks. It is evident that schedules that do not hold increases error rates as projects grow bigger and more complex, and misunderstanding the task, which means that the product contains features that the



customer does not demand at all, and at worst, one may risk that the project completely must be abandoned. XP tries to minimize risk by controlling four variables:

1. cost
2. time
3. quality
4. features

where the latter might be the most important as a measure of what the program should be capable of, but the four variables are of course not independent and in order to meet the cost and time requirements, it can easily mean that it is necessary to quench either quality (what is always short-term) or features. The only thing new in these variables is that the method from the outset focuses on the four key areas and controls them.

In addition to these control variables, participants in XP projects need to work from multiple core values, where I want to mention *communication*, where all participants commit to spreading valuable knowledge to the entire team. This can be achieved through relatively small development teams, like pair programming, and collective ownership of the code also help ensure effective communication.

In addition, extreme programming defines 12 principles for the development process:

1. Planning
2. Small releases at short intervals
3. System metaphors
4. Simple design
5. Test
6. Frequent refactorization
7. Pair-programming
8. Common ownership of the program code
9. Continuous integration
10. Affordable work rate
11. An overall development team
12. Common code standard

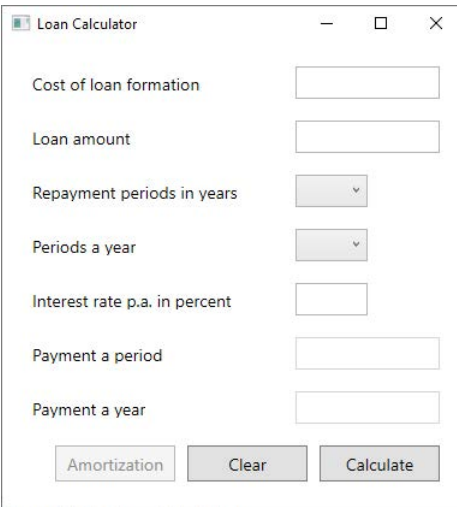
and to be successful with the method, it is considered crucial that these principles are respected.

There is much more about XP, and much are written about XP and gained experience, and it all shows - at least if you ask the method's advocates - that the method is particularly useful when the goal is to develop quality software.

## 4 UNIT TEST

Above I have mentioned unit test, which is a systematic way to test classes using a test project. It is an alternative to write your own test programs, and in this chapter, I will show what it is and how you use unit test to test a finish class.

As an example, I will use a program from from the book C# 2. The program is the solution for problem 3 in this book and is called *LoanCalculator*. It is a program used to calculate the payment, repayment and so on for a loan, and if you run the program, it opens the following window:

The screenshot shows a Windows application window titled "Loan Calculator". It contains several input fields and three buttons. The fields are: "Cost of loan formation", "Loan amount", "Repayment periods in years" (a dropdown menu), "Periods a year" (a dropdown menu), "Interest rate p.a. in percent", "Payment a period", and "Payment a year". At the bottom, there are three buttons: "Amortization", "Clear", and "Calculate".

where the user must enter information for an annuity loan (the upper 5 fields), and if you then click *Calculate* the program calculates the payment. You can also click *Amortization* and the program opens a window which shows the repayment, interest and outstanding for each period.

The program has a class *Loan* that represents a loan and has methods for the calculations. I will show how you can test this class using unit test, and for that, I have created a copy of the project.

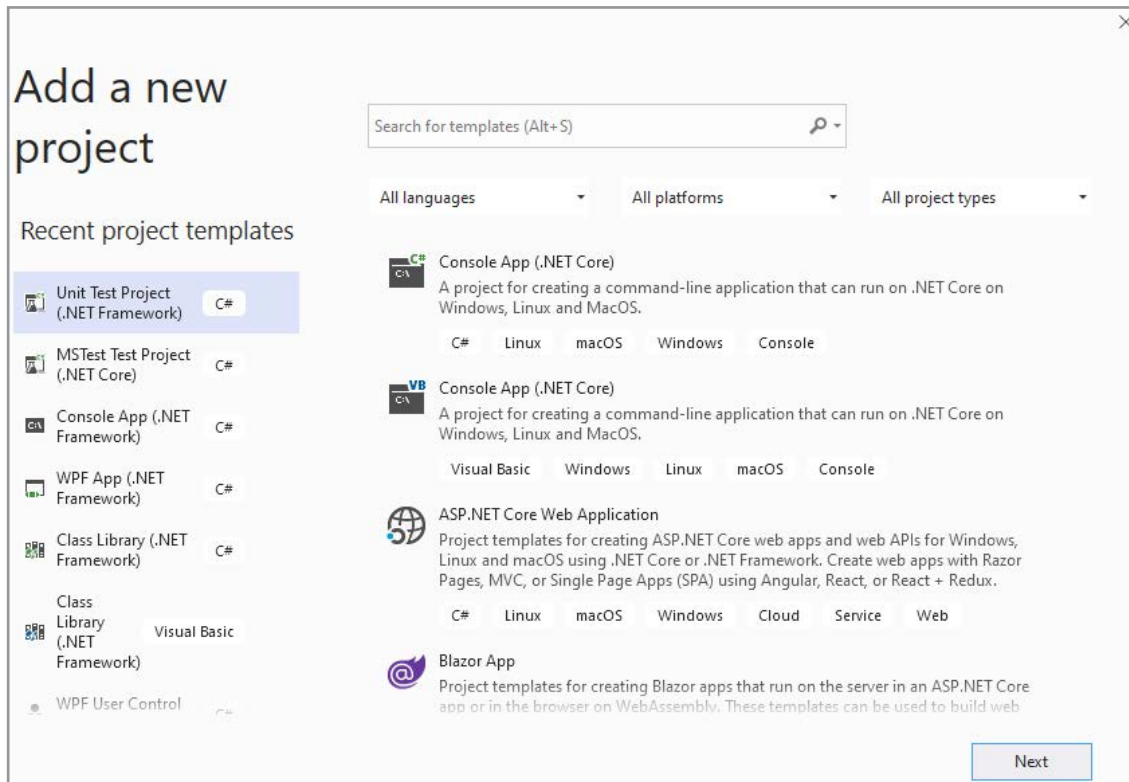
In Solution Explorer I right click on *Solution LoanCalculator* and select

*Add | New Project*

(see below). Here I selects

*Unit Test Project (.NET Framework)*

and I call the project *LoanCalculatorTest*. Visual Studio will then create a new project in the same solution, which is a test project. The project contains one test class with one test method:



```
using System;
using Microsoft.VisualStudio.TestTools.UnitTesting;

namespace LoanCalculatorTest
{
    [TestClass]
    public class UnitTest1
    {
        [TestMethod]
        public void TestMethod1()
        {
        }
    }
}
```

You should note that both the class and the method is decorated with an attribute. All test classes must be decorated this way and the same goes for all test methods. You can have other classes in a unit test project that do not have the *[TestClass]* attribute, and you can

have other methods in test classes that do not have the *[TestMethod]* attribute, and you can use these other classes and methods from your test methods.

As the first step I have renamed the test class to *LoanTest* and I have added a reference for the assembly for the program whose classes I want to test. Then I have written the test, which means written the test method and added another test method:

```
using System;
using Microsoft.VisualStudio.TestTools.UnitTesting;

using LoanCalculator;

namespace LoanCalculatorTest
{
    [TestClass]
    public class LoanTest
    {
        private double epsilon = 0.0001;

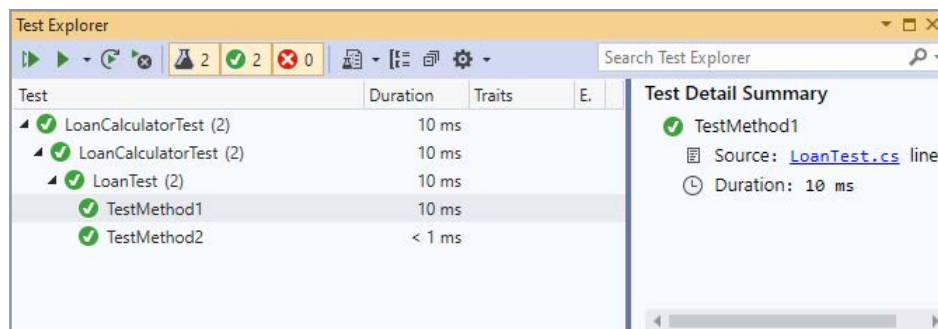
        [TestMethod]
        public void TestMethod1()
        {
            Loan loan = new Loan(500, 10000, 6, 4, 10);
            Assert.AreEqual(loan.Fformation, 500, epsilon);
            Assert.AreEqual(loan.Amount, 10000, epsilon);
            Assert.AreEqual(loan.Rate, 0.015, epsilon);
            Assert.AreEqual(loan.Periods, 40);
        }

        [TestMethod]
        public void TestMethod2()
        {
            Loan loan = new Loan(500, 10000, 6, 4, 10);
            double y = 0.015 * (10000 + 500) / (1 - Math.Pow(1 + 0.015,
                -40));
            double g5 = (10000 + 500) * Math.Pow(1 + 0.015, 5) -
                y * (Math.Pow(1 + 0.015, 5) - 1) / 0.015;
            double i6 = g5 * 0.015;
            double p6 = y - i6;
            Assert.AreEqual(loan.Payment, y, epsilon);
            Assert.AreEqual(loan.GetOutstanding(5), g5, epsilon);
            Assert.AreEqual(loan.GetInterest(6), i6, epsilon);
            Assert.AreEqual(loan.GetRepayment(6), p6, epsilon);
        }
    }
}
```

The class has a variable *epsilon*, which defines how large a deviation I will allow in the tests. A test method consists of test cases, and the first method creates a *Loan* object and define four test cases for this object. A test case consist of comparing a property for the object with a value, and if the difference is less than *epsilon*, it means that the test is accepted. The value *epsilon* is only needed if the values to compare are floating points, and it is not used in the last test case, where the values are *int*.

The other test method works in principle in the same way. There are also four test cases, but this time for methods instead of properties. The difference is that values to compare with this time are more complex, and the method calculates these values before the test cases are performed.

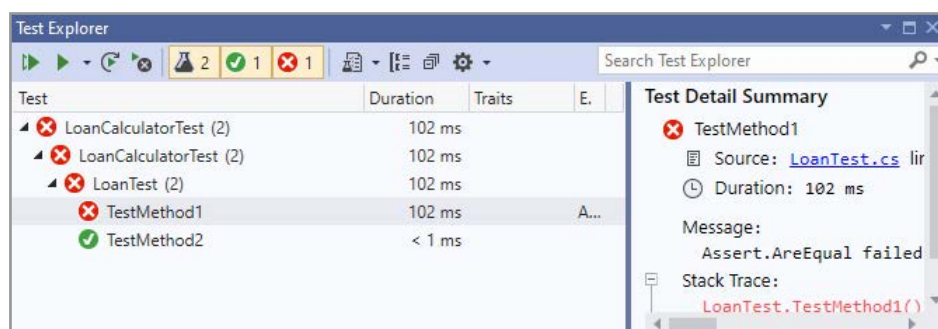
When you have written the test class, you can perform the test by right clicking the test project in Solution Explorer and select *Run Test*:



and in this case all test cases are performed without errors. If you as example in the first test method change the third test case to

```
Assert.AreEqual(loan.Rate, 0.15, epsilon);
```

and run the test again the result is:



and the test is performed with errors. It is then your job to find out if it is the class that is giving the wrong result or whether it is testing the case that is wrong.

In this example a test case is written as

```
Assert.AreEqual(...);
```

but there is other possibilities. You are encouraged to explore while options are available with Visual Studio, but they all basically works the same way. The best way to explore the possibilities is to use unit testing in practice and test completed classes.

Looking at the above example, it is not so easy to see the benefits of unit testing. To test a class you have to write many test cases, and writing unit tests for classes can be a great deal of extensive work, that can take a long time, but especially for class libraries, it's all worth the work. When you write a class, it must be tested, and if you later change the class (and it often happens) you must test the class again. The idea behind unit test is, if you have first written unit tests for a class the test can be performed, as often as you need and thus it becomes much simpler or safer to modify the class.

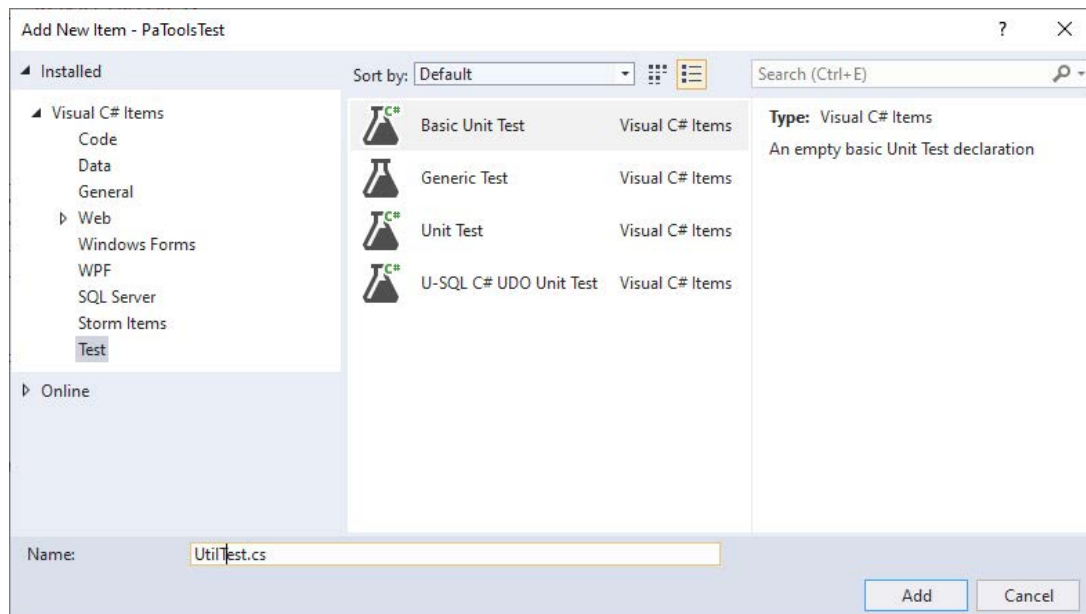
It is not all classes, which are suitable for unit testing, but especially model classes and classes pertaining to program logic and thus classes with complex algorithms. Therefore, classes in class libraries are especially suitable for unit testing. If you have a class library and it is necessary to change a class you can immediately execute the unit tested again and ensure that the changes did not cause any unwanted side effects. Similarly, if you need to expand with a new class it is easy to add a unit test for this class as well.

## EXERCISE 1: TEST LIBRARY

Create a class library project which you can call *PaTools*. Add a folder called *Math*. The source files for the book C# 5 has a class library called *PaLib*. The library has a class *Str*. Add this class to your new class library. *PaLib* also has a class *Util* in a folder *Math*. Add also this class to the folder *Math* in your new class library. Change the namespace for the two classes where you change *PaLib* to *PaTools*. Build your class library.

Add a test project to your solution. You can call your project for *PaToolsTest*. The project has a test method. Delete this method. Add a reference for your class library, and you should then write unit tests for the two classes.

Add a new item for your test project:

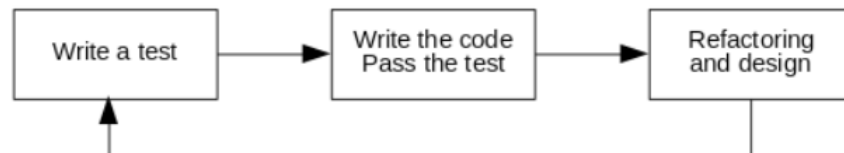


where you select the category *Test* and here item *Basic Unit Test*. When you click add the result is a new test class called *UtilTest*. You must then write test methods for the class *Util*.

Next you should add another test class, but this time you should select *Unit Test*. The result is a test class prepared for several help methods that you do not need in this case, but in other contexts they are useful. You must then write test methods for the class *Str*.

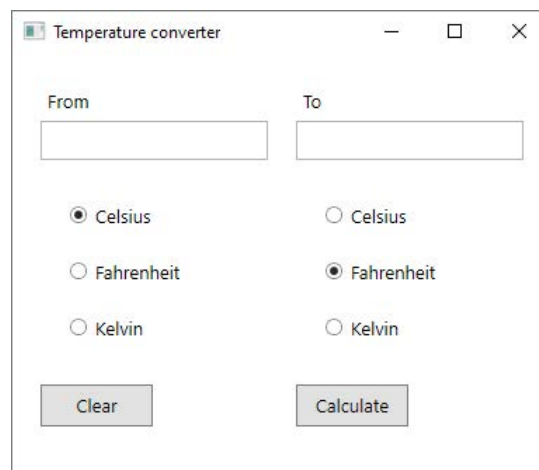
## 5 TEST-DRIVEN DEVELOPMENT

As mentioned in the previous chapter, *Test-Driven Development* - also shortened TDD - is part of the requirements for *Extreme Programming*. It is very simplified, that you start by writing some unit tests. Then you implement the code to be tested and perform the unit tests on the code and finally perform a refactoring. This process is repeated until complete:



TDD is based on the use of unit test, and I will in this chapter illustrates some of the idea with TDD I will show the development of a program where you can convert between three temperature units:

1. Celsius
2. Fahrenheit
3. Kelvin



The following shows the procedure.

I start with a project, which I have called *Degrees*. In this case, the program must consist of a user interface class *MainWindow* and a class *Converter* that will implement the necessary conversion methods. I start to create the XML for *MainWindow*, and when I run the program, it shows the above window.



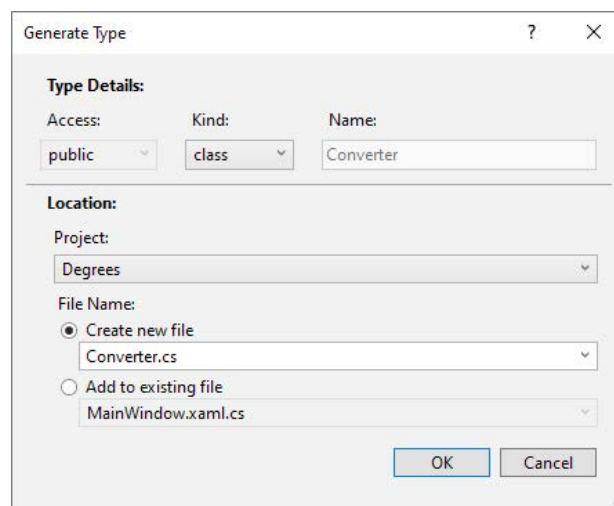
I want to think of the class *Converter* as the program's model and to write this class I will start with a test class. I add a test project to the same solution called *DegreesTest*, and I rename the test class to *ConverterTest*. Next, I add a test method (or rename the one created of Visual Studio):

```
[TestMethod]
public void FahrenheitToCelsius()
{
    Converter converter = new Converter();
}
```

Obviously, it results in a translation error, since the class *Converter* does not exist, but if you place the cursor over the class name (*Converter*) Visual Studio show *the light bulb menu* and from here I select

Generate type 'Converter' | Generate new type

Then Visual Studio opens the following window, where I can define the new class as a member of the project *Degrees* and tell that class should be created in a new file:



The result is that Visual Studio add a class *Converter* the project *Degrees*. The class is for now empty. The class must have a method *FtoC()* that can convert a temperature measured in Fahrenheit to Celsius. If *f* stands for degrees measured in Fahrenheit and *c* for degrees measured in Celsius, then the relationship can be expressed by the following formula:

$$c = (f - 32) * 5 / 9$$

In order to test the method *FtoC()*, I have expanded the test class with the static array, which for selected degrees measured in Fahrenheit shows the corresponding degrees in Celsius:

```
private static double[][] FC = {
    { -459.67, -273.15 },
    { -50, -45.56 },
    { -40, -40.00 },
    ...
    { 1000, 537.78 }
};
```

Note that you can create the table either by finding coherent values on the Internet or by calculating the values yourself. With this static array in place, the test method can be completed as follows:

```
[TestMethod]
public void FahrenheitToCelsius()
{
    Converter converter = new Converter();
    for (int i = 0; i < FC.GetLength(0); ++i)
        Assert.IsTrue(Math.Abs(converter.FtoC(FC[i, 0]) - FC[i, 1]) < 0.01);
}
```

Again, you get a translation error as the class *Converter* does not have a method *FtoC()*, but in the same way as for the class you can place the cursor over the method name (*FtoC*) and let Visual Studio create the method:

```
namespace Degrees
{
    public class Converter
    {
        public double FtoC(double v)
        {
            throw new NotImplementedException();
        }
    }
}
```

If you then try to perform the unit test, it will fail because the method has not yet been implemented. Next step is therefore to implement the method *FtoC()* - write code and pass the test:

```
public double FtoC(double v)
{
    return (v - 32) * 5 / 9;
}
```

Then you should be able to test the method (perform the above test method) successfully.

The next step is to perform a refactoring. In this case, both the class *Converter* and the method *FtoC()* must be *public*. In addition, -459.67 is the absolute zero point for temperatures measured in Fahrenheit, and the method should therefore raise an exception if the value of the parameter is less than the absolute zero, so the method can be refactored to the following:

```
public double FtoC(double v)
{
    if (v < -459.67) throw new Exception("Illegal temperature value");
    return (v - 32) * 5 / 9;
}
```

The class *Converter* must also have a method *CtoF()* that can convert from Celsius to Fahrenheit, and I therefore start adding a new test method to the test class:

```
public void CelsiusToFahrenheit()
{
    Converter converter = new Converter();
    for (int i = 0; i < FC.GetLength(0); ++i)
        Assert.IsTrue(Math.Abs(converter.CtoF(FC[i, 1]) - FC[i, 0]) <
            0.01);
}
```

The only difference is that the test method now tests a method *CtoF()* and that it is the Celsius value in the table, which is used as a parameter. The method *CtoF()* must be created, which I let Visual Studio do in the same way as above. Next, the method must be implemented, which consists simply of isolating the variable *f* in the above formula:

```
double CtoF(double d)
{
    return d * 9 / 5 + 32;
}
```

Then the unit test is performed again and it should be without error. Again, a refactoring of the method *CtoF()* must be made, so it raises an exception if the value of the parameter is below the absolute zero point of Celsius, which is -273.15.

The class *Converter* must have four additional methods:

1. *KtoC()* - convert Kelvin to Celsius
2. *CtoK()* - convert Celsius to Kelvin
3. *KtoF()* - convert Kelvin to Fahrenheit
4. *FtoK()* - convert Fahrenheit to Kelvin

The formula for converting Kelvin to Celsius is:

$$c = k - 273.15$$

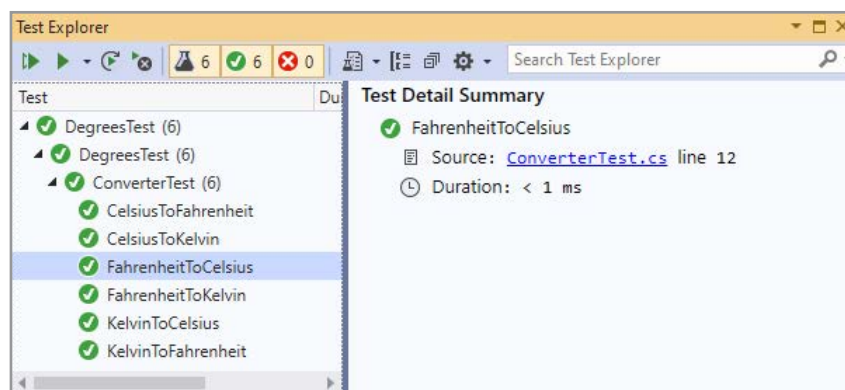
where *k* indicates degrees measured in Kelvin. Similarly, the formula for converting Kelvin to Fahrenheit is:

$$f = k * 9 / 5 - 459.67$$

First, the test class must be expanded with two new static tables:

```
private static double[][] KC = {  
    { 0, -273.15 },  
    { 10, -263.15 },  
    ...  
    { 1000, 726.85 }  
};  
  
private static double[][] KF = {  
    { 0, -459.67 },  
    { 10, -441.67 },  
    ...  
    { 1000, 1340.33 }  
};
```

where the first is a conversion table from Kelvin to Celsius, while the other is a conversion table from Kelvin to Fahrenheit. Next, the test methods are written for each of the four new methods and the new methods are implemented. I do not want to show the code for the four new test methods or the implementation of the four methods, as they are all in principle identical to the above - just other test data and other formulas are used. When all unit tests are performed without errors:

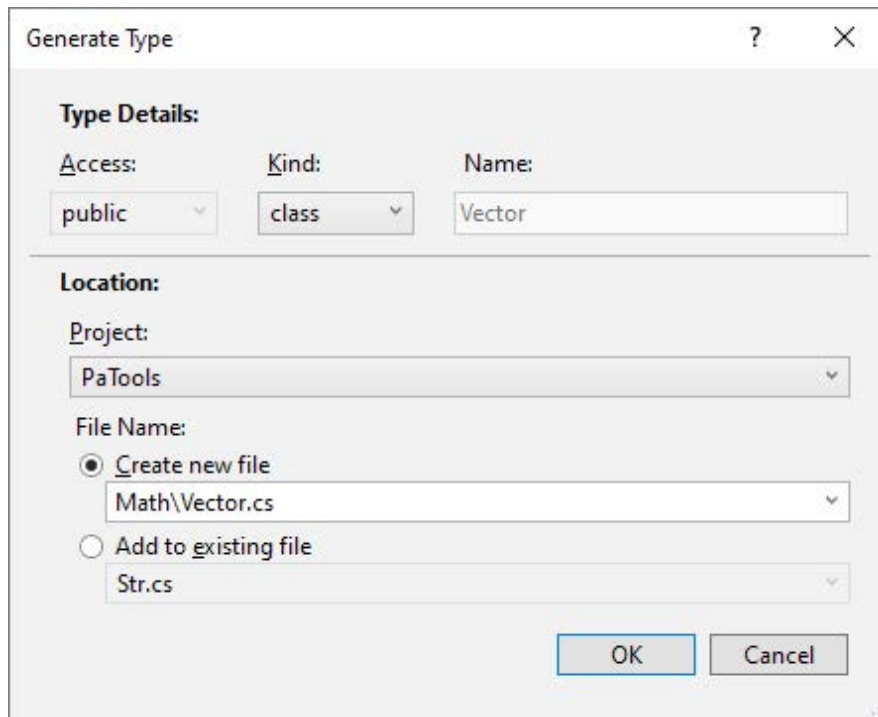


and you can then assume that the class *Converter* is complete and ready to be used. It happens in the class *MainWindow*, which opens a simple window, nor here I will show the code.

The class *Converter* is quite simple, and it is hardly clear what you've achieved by starting typing the test methods, but basically you get two things:

1. When writing a class, you decide through the test methods how the class' methods should work and when the methods are implemented, you can test them immediately using the test methods.
2. If you later update or change a method, you have the necessary test method, so you can immediately test where the method has errors.

Test Driven development is one of the many initiatives in system development that are worth learning and being good at, as the method gives very good results both in small and large projects.



## EXERCISE 2: THE TEST LIBRARY AGAIN

Create a copy of the project folder from exercise 1 and open the copy in Visual Studio. Note that the solution contains both projects, that is the class library and the test project.

The class library has a folder *Math*. Add the following interface to this folder:

```
namespace PaTools.Math
{
    public interface IVector : IEnumerable<double>
    {
        /// <summary>
        /// The number of elements in this vector, also called the
        /// dimension.
        /// </summary>
        int Size { get; }

        /// <summary>
        /// The i'te element in this vector or the i'te coordinate.
        /// The index i must be none negative and less the size of the
        /// vector.
        /// </summary>
        /// <param name="i">The element index</param>
        /// <returns>The value of the element at index i</returns>
        double this[int i] { get; set; }

        /// <summary>
        /// Multiply all coordinates with t, and returns a new vector with
        /// the
        /// result. The value of the current vector is unchanged.
        /// </summary>
        /// <param name="t">The value to multiply the elements in this
        /// vector</param>
        /// <returns>This vector multiplied by t</returns>
        IVector Mul(double t);

        /// <summary>
        /// Add v to this vector by adding the coordinates in pairs. It is
        /// assumed
        /// that the two vectors have the same length. The result is a new
        /// vector,
        /// that is the sum. The value of the current vector is unchanged.
        /// </summary>
        /// <param name="v">Vector to be added with the current vector</
        /// param>
        /// <returns>Sum of this vector and the parameter v</returns>
        IVector Add(IVector v);
    }
}
```

```

    /// <summary>
    /// Subtract v from this vector by subtracting the coordinates in
    /// pairs.
    /// It is assumed that the two vectors have the same length. The
    /// result
    /// is a new vector, that is the difference. The value of the
    /// current
    /// vector is unchanged.
    /// </summary>
    /// <param name="v">Vector to be subtracted from this vector</
    param>
    /// <returns>Difference of this vector and the parameter v</
    returns>
    IVector Sub(IVector v);

    /// <summary>
    /// Creates and returns the dot product of the current vector and
    /// the
    /// parameter v. The dot product is the sum of the products of the
    /// two vectors
    /// coordinates in pairs, that is coordinates with the same index.
    /// It is
    /// assumed that the two vectors have the same length.
    /// </summary>
    /// <param name="v">The vector used as argument for the dot</
    param>
    /// <returns>The dot product</returns>
    double Dot(IVector v);

    /// <summary>
    /// Calculates and returns the norm of the current vector. The
    /// norm is the
    /// square root of the vector's dot product with itself.
    /// </summary>
    /// <returns>The norm of this vector</returns>
    double Norm();

    /// <summary>
    /// A clone of this vector as an array.
    /// </summary>
    double[] Values { get; }
}

```



The interface defines a vector which is basically nothing more than an array of elements of the *double* type and some relatively simple methods.

You must then implement this interface, but you should do that trying to use test-driven development. Start to add a test class to the test project. Then add a class *Vector* to the class library:

```
public class Vector : IVector
{
    private double[] values;

    public Vector(int n)
    {
        values = new double[n];
    }
}
```

Then right click on the interface name *IValue* and *Quick Actions and Refactorings..* and let Visual Studio create a default implementation of all methods from the interface.

Then goto the test class:

1. create a test method for one of the methods in the class *Vector*
2. and write the test cases for this method
3. implement the method
4. run the test and repeat until the test is performed without errors

Repeat for all methods in the class *Vector*.

## 6 DESIGN PATTERNS

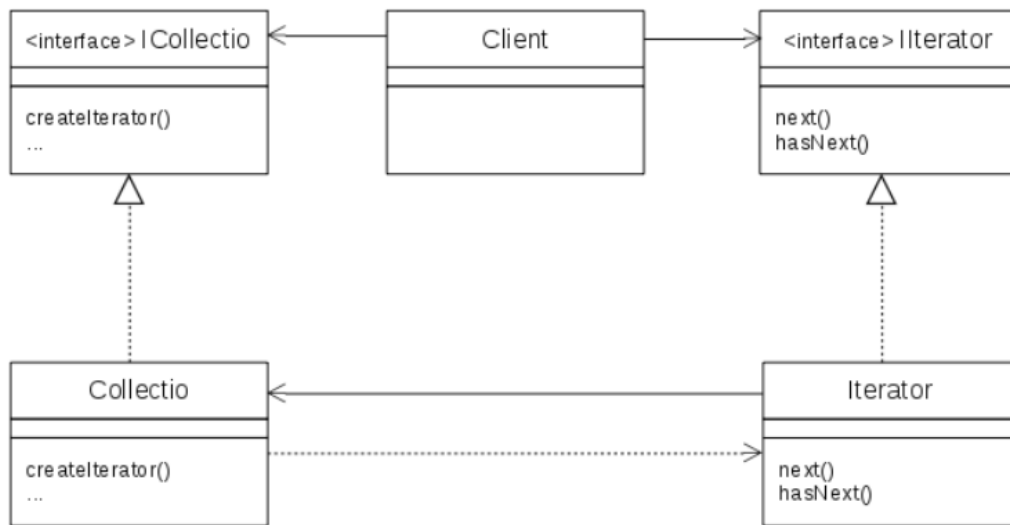
At some point, system development started to be talked about design patterns, and like everything else new, design patterns received great attention - and perhaps even some interpreted as the solution to this world's problems in software development. It is probably a little over-interpreted, but conversely, knowledge of design patterns is important for software development in practice. It all started with a book

*Design Patterns: Elements of Reusable Object-Oriented Software*

written by four writers, referred to as the *Gang of Four*. They described 23 patterns, and this book has since been the main source of design patterns. It is certainly recommended to read the book and study the 23 patterns. Since then, others have described other patterns, so today there are many to choose from.

A design pattern is a description of a solution for a particular problem. The starting point is a simple observation that certain issues appear more or less directly over and over again in many different software projects. If that is the case, one can try to describe the problem abstract and subsequently indicate a concrete solution. That way, as a developer, if you have a problem that can be categorized under a design pattern, you can easily figure out how to solve the problem. This means several important advantages, the most important being that you use the experience of other developers to solve a specific problem, rather than having to find in your own way to do it. This means cheaper and far more stable programs.

As an example of a design pattern, I would mention the *iterator pattern*, as I have previously mentioned and used several times. Given a collection of objects of one kind or another, the problem is that you need to traverse all objects and do something with them. If you use the design principle, to program to an interface, and the collection class *Collection* is defined by an interface *ICollection* so that the *Client* program only knows the collection class through the defining interface. The interface defines a method *createIterator()* that returns an iterator, which is an object that initially refers to the first object in the collection and has two methods, one that returns the object that the iterator points to and step the iterator forward, while the other method tests if the end is reached. The class *Iterator* can also be defined by an interface, and the iterator pattern can thus be illustrated as follows:



If a collection class implements this pattern, it means that the class can be treated in the same way, regardless of whether it is for a collection. For example, think about C#'s implementation of collection classes.

I have used other design patterns in the previous books:

- Singleton
- Factory
- Adapter
- Observer

and there are many others, and since *Gang of Four* defined 23, many others have been defined. The latter can be a challenge as many patterns look similar and can be difficult to distinguish, and there has also been a tendency to define patterns for anything, but it does not change that patterns are important and that it pays to teach them.

The patterns of *Gang of Four* and many others concerns concrete issues in programming, but other patterns are more general and are often composed of several patterns. One example is MVC (or MVVM), which is a pattern for the design of a GUI program. It is also a design pattern that expresses others' experience of how to design a GUI program. If you do, others who must maintain the program knows how the program is designed. The pattern is described in the previous book. It is a complex pattern, and it is a pattern that is found in several variants. It expresses the fact that it is actually difficult to define design patterns if they are to be general and therefore MVC exists in so many variants, which are determined by both software platforms and development tools.

Another important reason for using design patterns is that developers in this way have a family of common concepts. For example, if I say that a class in a program is an adapter, I told others that it is a class that is used to smooth the differences to other classes, and I can even look up a book about design patterns and see what an adapter is and how it is used.

# 7 REFACTORING AND OTHER

In order for a development project to be successful, the completed program must meet:

- that the program meets the requirements
- that the program works without errors
- that the program is robust
- that the program is easy to maintain

and if you read the literature you will be presented with a number of other quality factors, but in addition to that, you can add that the project is completed within the estimated resource consumption and that it is completed on time.

## 7.1 REFACTORING

In order to meet these requirements, you have to work systematically and follow a method, and the four basic activities in system development are, as already mentioned many times:

1. analysis
2. design
3. programming
4. test

and then combined with iterative system development, but in addition to that, there will always be an activity called *refactoring*, where you to some extent has to rewrite the code to make it easier to read and maintain, while you have to retain the code's functionality.

Once you have written the code and tested it, and found that it meet the requirements, the code has been through a process where you have developed on the code, tested it, corrected, adjusted, expanded and so on and the result will very often be a code that is characterized by:

- Variables not used and which must therefore be removed.
- Methods not used, which should therefore be removed.
- Two or more methods that almost do the same, and where you can reduce to a single method using a parameter.
- Methods that may be called something different, but do the same where you should only have one method that may be placed in a particular tool class.

- Instance methods that do not depend on the instance, where the method maybe can be defined static or move to a tool class.
- Classes that are almost identical, where classes can be changed to specializations of a common super class and where the methods that are the same are moved to the super class.
- Code commented because the code should not be used. Such a code must be removed.
- Code that solves the task, but makes it inappropriate. Here you should find a better code, which usually is the same as a better algorithm.
- Missing comments, which should be written.
- Classes with very high coupling. If so, you should consider whether you can reduce the coupling in one way or another. An example could be to implement the observer pattern using an interface.
- Classes with very low cohesion. If so, one should consider dividing the class into several classes.
- Using statements that are unnecessary. They should be removed, so you only have the using statements that are necessary.

and many other examples could be mentioned. These relationships mean two things:

1. That the code fills more than necessary is.
2. That the code is far harder to maintain than necessary is.

and here it is absolutely the last, which is the biggest problem. Therefore, the code must be refactored, which can be considered as a cleanup process to clean the code for the above inconvenience.

As mentioned, the aim of a refactoring is to get a simpler code and thus a code that is easier to maintain. Conversely, you should note that refactoring does not develop into direct reprogramming. If that is the case, the cause is usually serious error decisions during the development process, and if necessary, you should postpone a refactoring to a new project where the task is directly a refactoring of an existing program. To avoid getting in that situation, the solution is frequent refactoring, and maybe it should be part of each iteration, and here's the worth to note that in Extreme Programming refactoring is an integral part of each iteration.

## 7.2 LOG FILES

As a last comment on system development, I want to mention log files, which are typically text files, where a program intermittently stores data relating to the application's use. Now it does not have to be text files, but it can also be a database table, and if necessary, make sure that the program has database access. If you are using text files, make sure that the log file is created in a directory with a write access, and often you want to use the user's home directory, but it is of course not an option if the log is to be used by multiple users. If a program uses log files, it can of course be a little dangerous to let a program write unrestricted and unchecked to a file or database table, and therefore log files should usually be combined with an option to disable / enable this functionality.

There may be several reasons for using log files, and one is to collect statistic data relating to the application's use, perhaps in order to improve the program later. Another and perhaps even more important application is logging of an error message. Once a program has been put into operation and errors occurs, they are reported to developers who then have to correct the errors. In order for this to be possible, you need some information, and here I would like to say that an error handling, where the program just crash and the screen is filled with errors regarding exceptions, simply cannot be used. Firstly, users will not accept such malware, and it creates frustration and unwillingness towards the development organization, and secondly, it is far from the assumption that users are reported to such errors, and at best, you will be notified that the program is not working. As a developer, you know critical places where a program can go down and here you can encapsulate the code that can fail in a try / catch block and in case of an exception write a message in a log file. As a developer, in case of an error, you can ask to get the log file, and here you can see when and where in the program there have been errors.

Such error logging can be a very big help in troubleshooting, just be careful about where in the code you write in the log file. In principle, the error log must be empty and therefore it may be a nice check of the program periodically to inspect the log file.

It is important to note that error logging is by no means an alternative to testing and it is only something that can be used when the program is finished and tested and ready to deliver to the customer, but conversely, has the program first been put into operation, it can also be an extremely important source of reporting errors.

## 8 DATABASE DESIGN

In this chapter, I will look a little on database design. When designing a program, you determines which data has to be stored in the database, and the question is how to design the database and including which tables there needs to be. There is not a clear answer, but there are some pretty precise recommendations, which could be followed, and it is the subject of this subject.

In practice system development, database design is such a frequent activity that there is a need for special focus on that side of the case, but also because the database design has a decisive influence on the efficiency of the program.

### 8.1 THE ER DIAGRAM

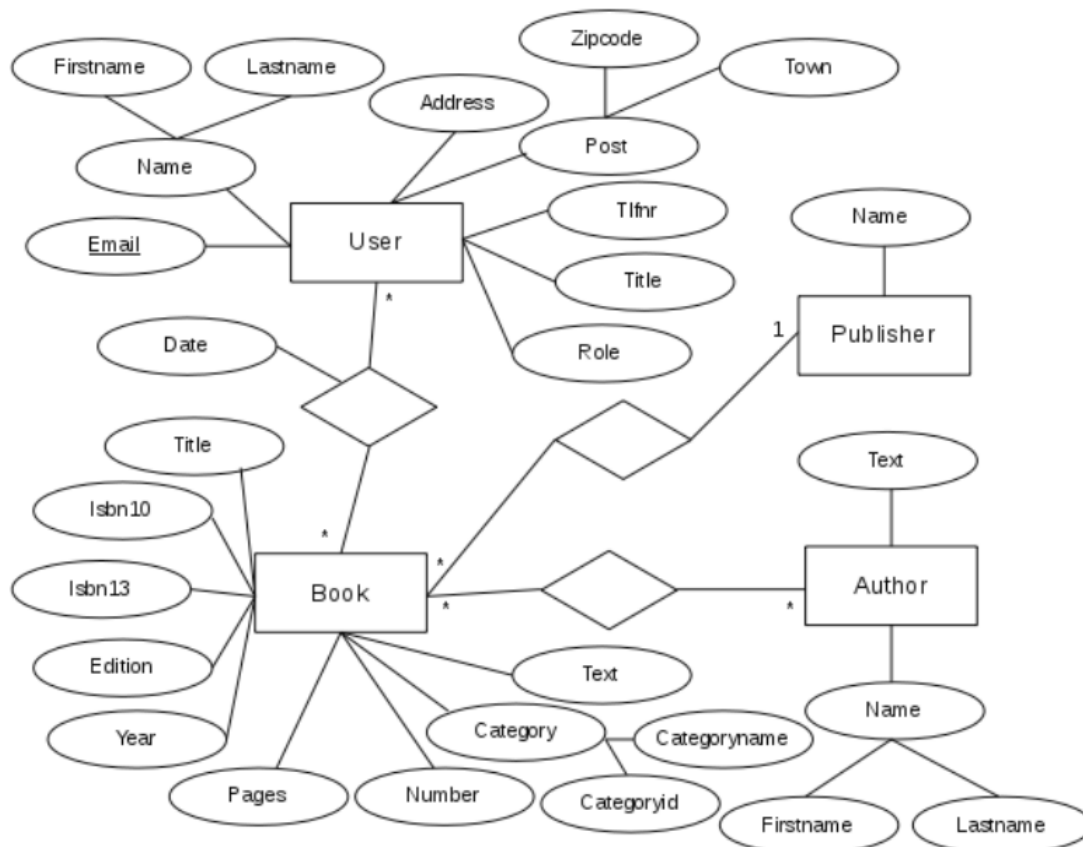
The first step is to draw an ER diagram. Indeed, one could instead use a class diagram, which offers the same opportunities, but the ER diagram is directly developed for the design of relational databases, and it is also a diagram that I and other use a lot in practice. The diagram uses three symbols



where an entity describes an element that is a candidate for a table. An entity plays a bit the same role, as a class do in a class diagram. An attribute is a property of an entity and corresponds thus to a variable in a class, and an attribute is a candidate for a column in a database table. Finally, a relation is a relationship between entities and corresponds thus to relations in a class diagram.

As an example, is shown an ER diagram for the library database from the final example in the previous book, but the database is expanded with an entity user which are people who use or lend the books in the database:





There are four entities which are respectively *Book*, *Author*, *Publisher* and *User*. You can find these entities in the same way that you find classes to the model layer, and of course there is no clear solution. It might also not be so important, for if you have found the right attributes, then the following rules results in almost the same tables. The best thing is to think about what data is needed to save, and then organize this data in the appropriate entities.

If you considers the entity *Book* it has 9 attributes, and here is the one *Category* an example of a complex attribute, which is composed of *Categoryid* and *Categoryname* indicating respectively an identification of a category and the category name. In addition, all the other attributes also should be documented so that it is unambiguous what they are used for. That there exactly should be those attributes are a result of the analysis that has taken place, where you have identified the data elements to be stored in the database.

The entity *Author* has two attributes, one of which is a complex attribute. The entity defines information about an author, and again the analysis has justifying that the entity should have these attributes.

Between the two entities *Book* and *Author* is a relationship that is a many-many relation that appears with an \* on both sides. The relation indicates that a book may be related

to several authors, corresponding to that a book may have multiple authors. Similarly, an author may be related to several books as an author may have written several books.

The entity *Publisher* represents publishers and has only one attribute, which is the publisher's name. Here too there is a relationship between *Book* and *Publisher*, but it is a one-many relation. There is a \* against the entity *Book*, indicating that a publisher may have published several books while the multiplicity against *Publisher* is 1. It says that a book must have exactly one publisher.

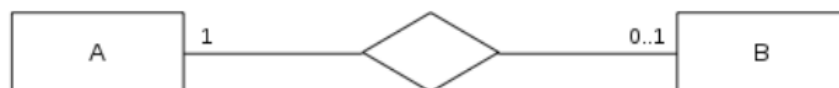
Finally, there is the entity *User* having 7 attributes, wherein the two are composed. The *Email* attribute is underlined, which means that it can be used as a primary key. That is, it is assumed that all users have different email addresses. The other three entities have no obvious key, and specifically the *Book* has no key when a book does not necessarily have an ISBN. In *Publisher* I could use the *Name* as key, as there are not two publishers that has the same name, but it is not an appropriate key, as it is a string that can fill much.

Between *User* and *Book* is also a many-many relationship, which indicates that a user has lend a book. Since a user may lend more books, there is a \* on the book side. When there is a \* on the other side, it is because you can have multiple copies of the same book, and therefore it can be borrowed to more users. The relation is also an example of a relation with an attribute that here indicates when the book is lend.

The above is a typical ER diagram, but there are actually several drawing rules. Specifically, you can have one-one relation:

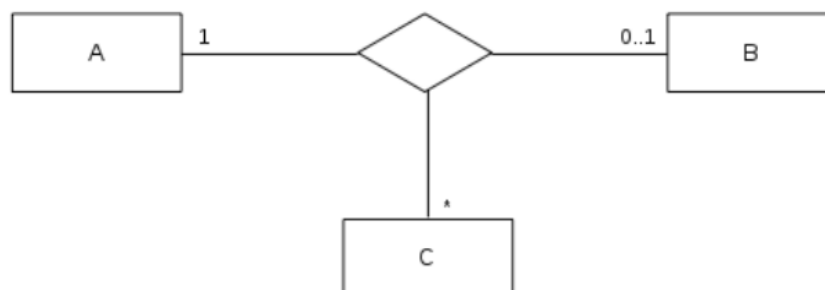


This figure indicates that an entity A must correspond to exactly one entity B, and conversely that an entity B should correspond to exactly one entity A. The relation could also be

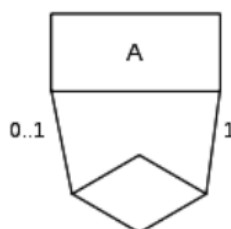


indicating that an entity B must correspond to exactly one entity A, while an A should correspond to only one or no B. In this case we say that there is total dependence on the A side, while there is partial dependence on B side.

One can also have relations between more than two entities, for example is shown a relation between three entities:



In particular, you should note that an entity can have a relation to itself:

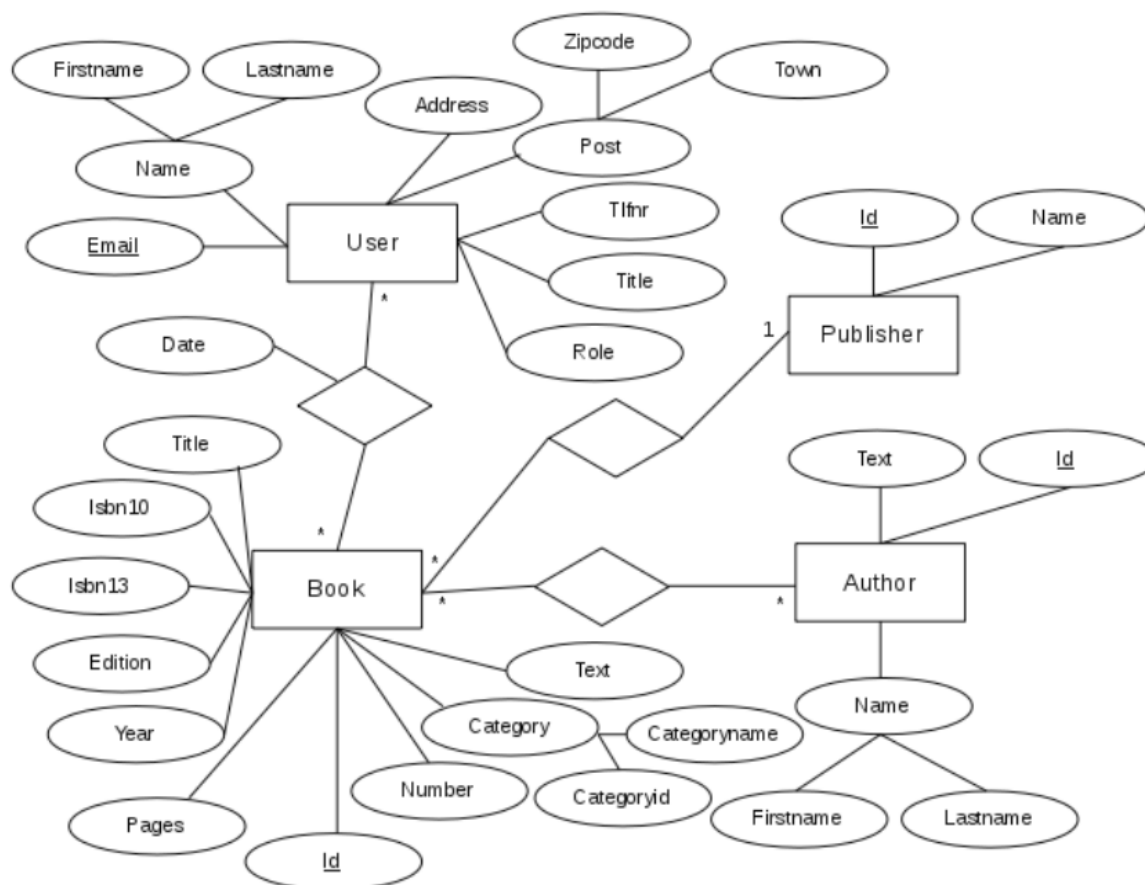


An example would be “son of”.

Finally, there is something called a weak entity, and it's an entities, which cannot exist unless they are related to another entity. A weak entity is displayed as follows:



A non-weak entity is called a *strong* entity and is characterized in that it has a key. The above diagram is not drawn correctly when three of the entities do not have keys. It is however nor weak entities, as they all cover concepts that can exist without being related to another entity. When you have an entity in which none of the attributes are candidates for keys, you has to use a surrogate key, which is an attribute that has no other purpose than to ensure uniqueness of entities. With three surrogate keys, the diagram can be drawn in the following manner, where all entities now have keys:



The diagram has no weak entities, but a weak entity is typically an entity that has a partial key, and thus an attribute which, together with the key from the entity that the weak entity is attached to are a composite key for the weak entity.

An entity that is not a weak entity is sometimes referred to as a regular entity.

## 8.2 MAPPING TO RELATIONAL MODEL

When the ER diagram is finished, it must be mapped to a relational model, as you do in 7 steps. In this context denotes a relation, what that turns into a table in the database, and it consists of fields that later become columns. The seven steps of mapping an ER diagram to a relational model is quite precise and are as follows:

### Step 1

For each regular entity you creates a relation with a field for each simple attribute, and a field for each simple attribute in a composite attribute. In this case, there are four relations,

where the relation for *Book* gets 11 fields, the relation for *Author* 4 fields and the relation for *Publisher* 2 fields. Finally, there is the relation *User*, which in principle should have 9 fields, but I've added a surrogate key, so it will get 10 fields. The reason is, that even if you can assume that the mail address is unique and thus can be used as a key it is not suitable, because in database contexts the value of a key in principle cannot be changed, and when people often change email addresses, it is not suitable for the key, and also email addresses are long strings are not especially suitable as keys. The result of the first step is as shown below:

Book

<u>Id</u>	Isbn10	Isbn13	Title	Edition	Year	Pages	Number	Catid	Name
	Text								

Author

<u>Id</u>	Firstname	Lastname	Text
-----------	-----------	----------	------

Publisher

<u>Id</u>	Name
-----------	------

User

<u>Id</u>	Emai	Firstname	Lastname	Address	Zipcode	Town	Tlfnr	Title	Role
-----------	------	-----------	----------	---------	---------	------	-------	-------	------

## Step 2

Then all weak entities are mapped in the same way, and the only difference is that each relation, this time will have an additional field which is the primary key from the regular entity that the weak entity is associated with (additional fields if the primary key is composed). The key in the relation for the weak entity is then the key from the regular entity and the partial key.

In this case, this step results in no changes, as there is no weak entities.

## Step 3

All one-one relations between two entities are mapped. In this case there is none, but the rules are the following.

If the relation is of the form



the relation for B (the side with partial dependence) is extended with a field that is the primary key of A, as well as fields for any attributes associated with the relation. The new field in B is thus a foreign key that refers to A. Is there instead talk about following relation



you must choose one of the sides and place the foreign key there (and possibly attributes). Alternatively, you could create an entirely new relation that then must contain the primary keys from both entities that then are a composite primary key in the new relation, and they will also each be foreign keys respectively to A and B. The new relationship must also contain necessary attributes associated with the relation between A and B. Are there many attributes, it can talk for this alternative solution.

In the particular case of a one-one relation for the relation itself, one will typically implement the relation as a foreign key.

#### Step 4

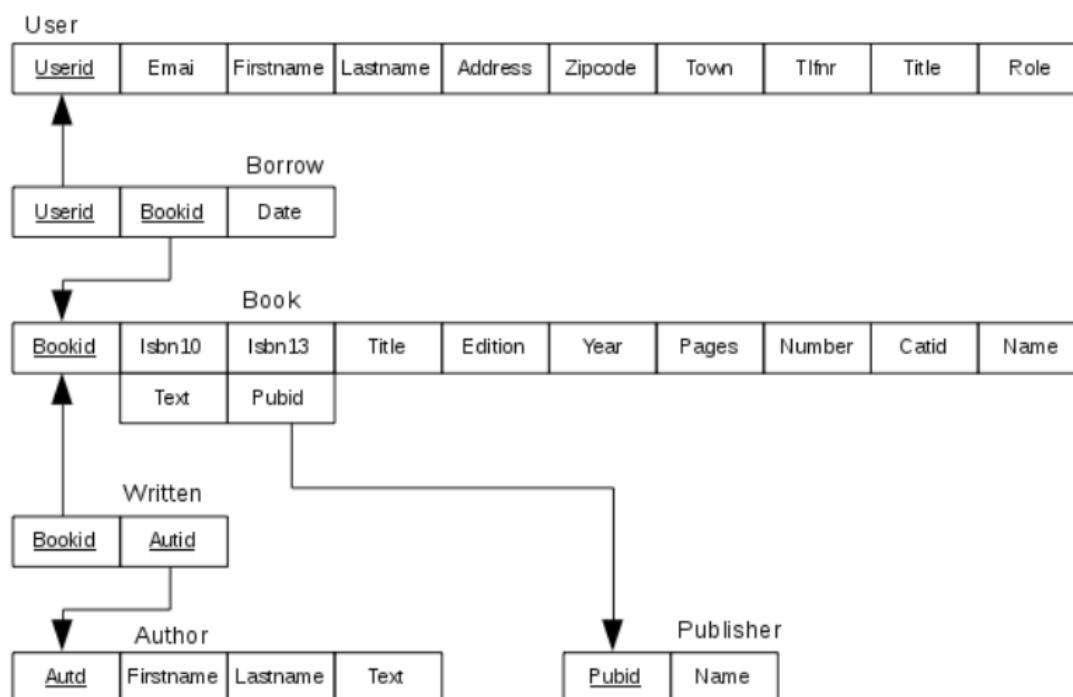
In the case of a one-many relation, it is always implemented as a foreign key at the many side. In this case there is a single one-many relation, and the relation for *Book* must be expanded with an additional field, which is the primary key in the relation *Publisher*, and *Book* therefore has a foreign key to *Publisher*:

Book									
<u>Id</u>	Isbn10	Isbn13	Title	Edition	Year	Pages	Number	Catid	Name
	Text	Pubid							

#### Step 5

Many-many relations between two entities are always mapped as a new relation that includes the two primary keys from the two entities and possible attributes associated with the relation. The two primary keys is a composite primary key in the new relation, and they are each foreign keys to the two entities. The mapping is thus quite in the same manner as described above as an alternative for mapping of a one-one relation. In this example there

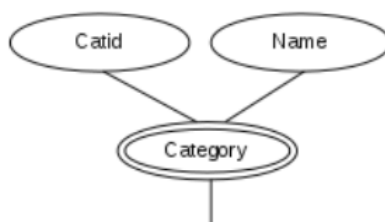
are two many-many relationships, and the diagram below shows the result after step 5 is performed;



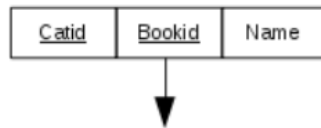
Note that there are two new relations that I have called, respectively *Written* and *Borrow*. The arrows indicate foreign keys. Note that I also have renamed some of the attributes.

## Step 6

The next step concerns the multivalued attributes that are attributes that can have multiple values. In this case, there is no multivalued attribute, but a multivalued attribute is drawing in the following manner:

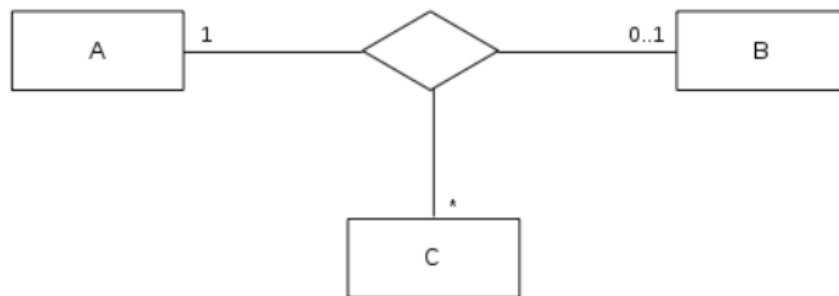


to illustrate that a book could have several categories. If need be, the multivalued attribute must be moved into a new relation, that in addition to the two attributes consists of the primary key from the original relation (*Book*). The new relationship will have a composite primary key, and *Bookid* is foreign key to *Book*.

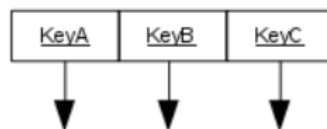


### Step 7

The final rule addresses relations between more than two entities, for example a relationship of the form:



This creates a new relation which contains the primary keys of the three entities and possible attributes associated with the relation, and thus the mapping is done in principle in the same way as with many-many relations:



After this the mapping the database model is in principle finished, and the outcome was in this case 6 relations, all of which have a primary key and contains no multivalued attributes. You could then create a similar database, which would then have 6 tables. However, there is a quality assessment of the results that have been reached, and we talk about that the design should be *normalized*. This is done by ensuring that the design meets several normal forms, and usually that database must be on the third normal form. There are more, but in practice you usually stops with the third normal form.

## 8.3 NORMALIZATION

We say that a relational model is on first normal form if it consists of relations which has no multivalued attributes, and if all relations have a primary key. If you have completed the above mapping, it will automatically be satisfied and the above model is then on first normal form. You can also think of it in that way, that you can create a database from a relational model that is on first normal form.

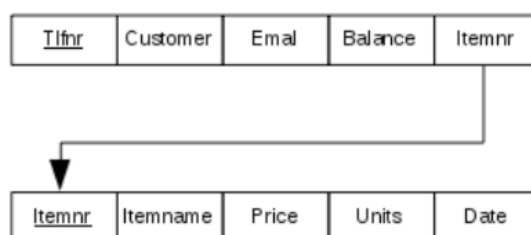


A relational model is on second normal form if it is on first normal form and, if it for any relation applies that it does not contain attributes, which is determined by a part of a composite primary key. The fact that a relation is not on second normal form says that it contains information on more concepts, and therefore should be divided into two relations where the attributes that are determined by a part of the primary key, are moved into a new relation together with the controlling part of the primary key, and it is as a primary key in the new relation. In the original relation, the controlling part of the primary key is a foreign key to the new relation. If a relationship does not have a composite primary key, it is per definition on second normal form. In the current example, there are only two relations which have a composite primary key, namely, *Written* and *Borrow*. Here, the first, has only the primary key and is therefore on second normal form, and the latter has only one additional attribute, which is determined by all of the primary key and is therefore also on the second normal form.

To illustrate the principle, the following relation is perceived as part of a model for a sales system, which for customers shows what products they have bought:

<u>Tlfnr</u>	Date	Customer	Emal	Balance	<u>Itemnr</u>	Itemname	Price	Units
--------------	------	----------	------	---------	---------------	----------	-------	-------

This means that a certain product sales is identified by the customer's phone number and the part number (it is a composite primary key). For each sale is stored the date, the customer name and email address and the customer balance, product name, unit price and number of units. This relation is not on second normal form, since some of the attributes is determined by the field *phone*, while others are determined by the field *itemnr*, and the relationship must be divided into two relations as follows:



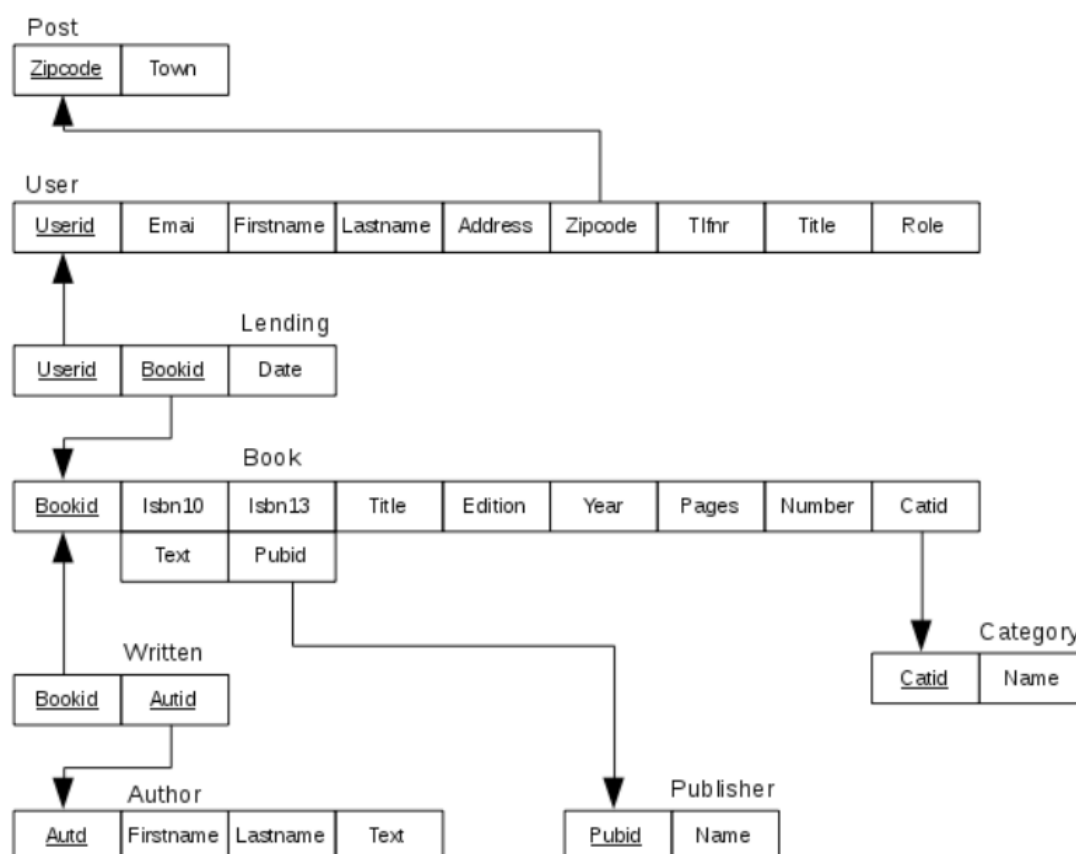
The goal of the second normal form is to ensure that the database tables do not contain data concerning more things, since it can make it more difficult to understand database content.

A relational model is on third normal form, if it is on second normal form and, if for any relations applies that it does not contain attributes, which is determined by other fields than the key. It expresses that there for a piece of information is not stored more than necessary, and if so must be divided into two relations where the attributes are determined by something

other than the key, is moved into a new relation together with the determining attribute that it is as a primary key in the new relationship. The determining attribute remains in the original relation where it then is a foreign key to the new relation. If a relation is not on third normal form we can say that there is a transitive dependency.

Looking at the current example, it is clear that relations *Publisher*, *Author*, *Written* and *Borrow* all are on the third normal form. However, looking at the relation *User* it has a transitive dependency because the city name is determined by the zip code. There must be created a new relation that includes the zip code and the city name, and where the zip code is the primary key. At the same time the city name is removed from the relation *User* and the attribute *zipcode* is a foreign key to the new relation. If you consider the relation *Book*, it corresponding has a transitive dependency because the category name is determined by category id. There must therefore be created a new relation with the attributes *catId* and *CategoryName* and *catId* is the primary key. The name attribute is removed from *Book*, while *catId* being foreign key to the new relation.

After normalization is the relational model of the database as follows:



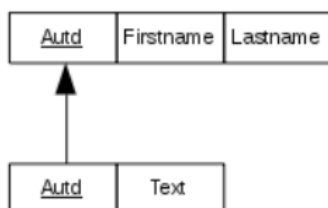
and it consists therefore of 8 relations.

The aim of third normal form is to eliminate redundancy, which means that the same information is recorded at multiple places. For example category name must not be stored for any books, but to save it in its own table, a book must then have a reference (a foreign key) to the name. When redundancy is unfortunate it is because that stored the same information in several places, it can mean unnecessary space consumption, but the main reason is that it becomes more difficult to maintain the database, as it may mean that an information has to be changed in several places. If, for example, imagine that you want to change the name of a category (perhaps because it's spelled wrong), then it would mean if the name was not in its own table that it would be necessary to change all books for that category while with the database on the third normal form you only need to change at one place.

## 8.4 OTHER DATABASE IMPROVEMENTS

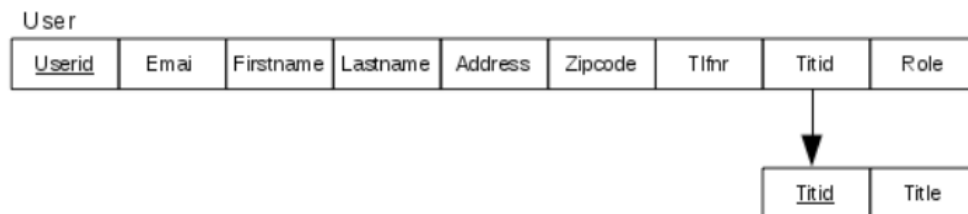
If you look at the normalization process, it will typically entail more tables in the database, and it is in principle nothing wrong with that but it may mean that in requests to databases are required with many JOIN operations that are actually complex to perform by the database management system. One can say that a fully normalized database meets maintaining the database, and thus its integrity, but not necessarily satisfy queries. Therefore, we speak also about de-normalization where one goes the other way and turn tables together, even if it means that the database then contains redundancy. The reason for de-normalization can be databases, which is rarely updated and you want to optimize for the sake of queries.

If you have a database normalized to third normal form you have in principle a good database design, but there are other considerations to make. For example are NULL values a problem - especially for JOIN operations. If you have an attribute that is typically NULL, you should consider moving it to its own table. If you, as an example, take the relation *Author* it has an attribute *Text*, which is a description of an author. One must assume that this attribute will typically be NULL, because you'll rarely describe an author. You can then move it out to its own relation with the primary key of the *Author* as key. The new relation thus has two attributes, where *Autid* is both primary key and foreign key to the table *Author*:



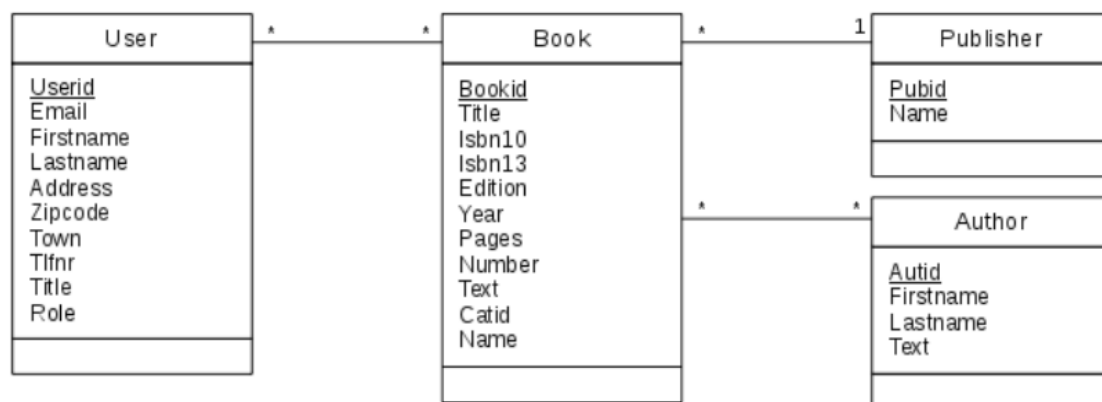
Now, you should not generally move all attributes that can be NULL to their own table, because it can get the number of tables to explode and thus complicate the database unnecessarily, but if you have an attribute as *Text*, that very rarely has a value, you should consider to do it.

As another example, one can have an attribute, where the value is a text, and where there will be many rows have the same value for that attribute. Where necessary, we can consider moving the attribute to its own table. The reason is that a text takes up a lot and there is a risk that the same value is spelled differently. As an example, you may consider the relation *User* that has a *Title* attribute that indicates a user's title. It could, for instance be Student, Teacher etc. Here you could then consider the following design, where that attribute is moved to its own table with a surrogate key. The attribute is in the original relation replaced by this key. The relational model will then be extended accordingly.



## 8.5 THE USE OF A CLASS DIAGRAM

The basis of the above approach to database modeling is an ER diagram, but you might as well start with a class diagram:

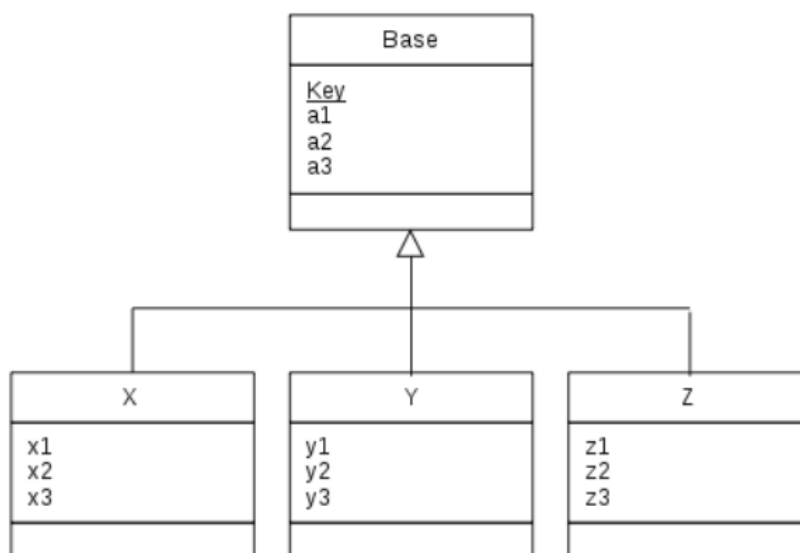


The difference is indeed just that the Entities are drawn as classes with attributes and the relations are shown as associations. With a diagram like the above, it is clear that the 7 mapping rules above can be used directly, and to end up with exactly the same result.

Whether you use an ER diagram or a class diagram has something to do with attitudes. I find the ER diagram particularly suitable, especially in collaboration with others and have to model a database, and especially I think it's a great tool if you are more and have a blackboard available. Conversely, the class diagram is easier to draw, and a large ER diagram quickly become confusing. Therefore, I often use in the initial modeling an ER diagram without attributes that alone shows the entities and relations. The attributes come first in action in step 1 in the mapping.

The construction of the database is a design activity, and the starting point will typically be a design of the model layer, and then you perhaps already has a class diagram that can be adjusted as a basis for the database, but what do you do if this model contains inheritance? The relational model does not support inheritance, and therefore there is a need for a method that can map a class hierarchy of relationships in a relational model.

In fact, there is a so-called EER diagram (for enhanced ER diagram) which supports modeling specializations. The diagram has very complex drawing rules and hard to read, and I never use this tool, but if you are a big supporter of the ER diagram it can be forced worth learning the diagram to know - also because the mapping of specializations can be done in several ways, what the EER diagram has syntax for. Assume as an example that you have the following classes:



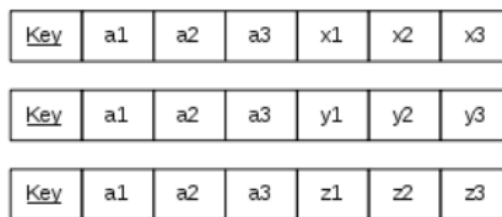
where there of course does not need to be three derived classes, and where each class can have both fewer and more attributes. I will also show 4 options for mapping this structure of a relational model.

One possibility is to create a relation for each of the four classes (entities):



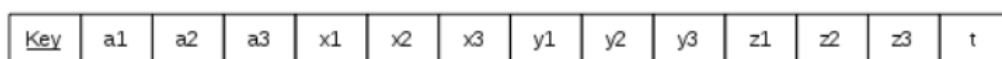
That is to create a relation with the base class attributes and its primary key. There also are created a relation for each of the derived classes, wherein each relation contains the derived class attributes and the key from the base class. The primary key of the relations for the derived classes is the primary key from the base class, and it is also a foreign key to the base class relation. You can say that it is a general method that can always be used regardless of how the specialization may be.

As an alternative, you can create a relation for each derived class, which then must contains all attributes from the base class, including the primary key and attributes from the derived class:



This solution is only interesting if an object is always either an *X*, *Y* or *Z*, but cannot be a base object. The solution is thus of interest in the situation where the base is an *abstract* class.

As another example, is a mapping where all the attributes together are in a single relation:



This solution requires an additional attribute - referred to here as *t* for type - which indicates the type of object in question. It is a design that has the disadvantage that it leads to many NULL values (where, for example it is an *X* object, then all the *y* and *z* attributes must be NULL) and the solution therefore has greatest interest, if the derived classes have few attributes.

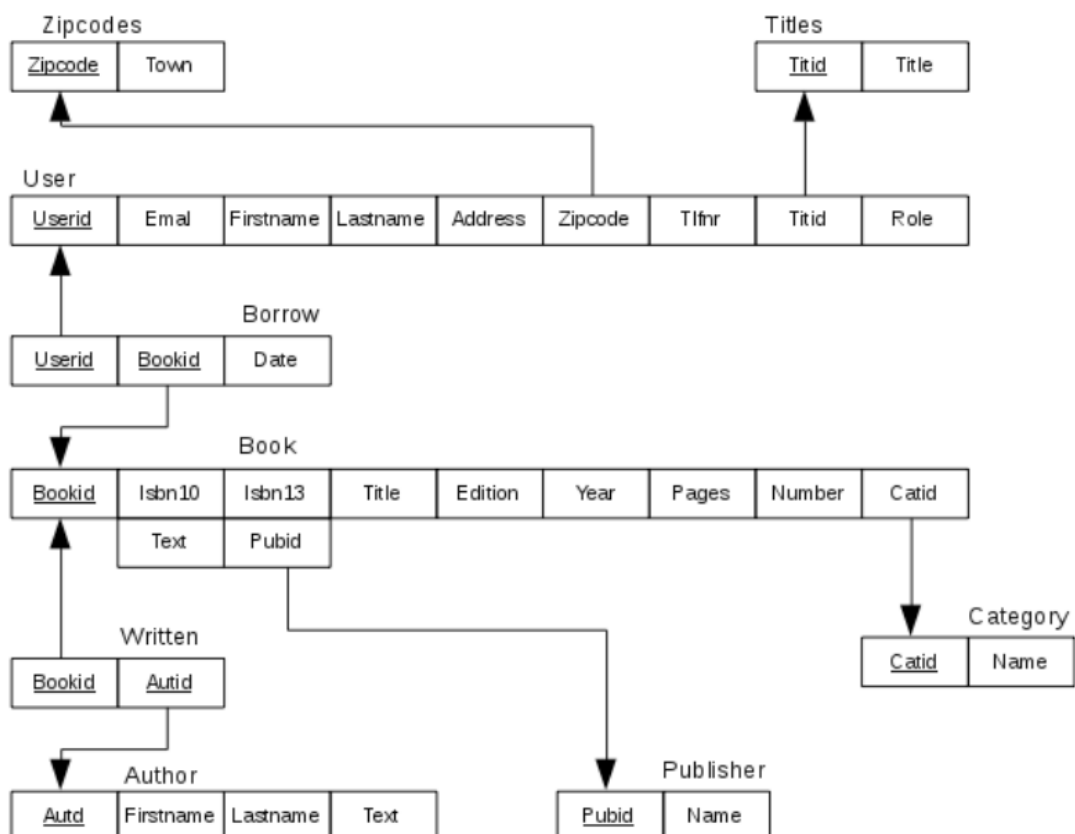
When designing databases it can actually happen that an object can be several things, as for example both an X, Y and Z. If you have such a design, consider the following mapping, which is a variant of the above:

<u>Key</u>	a1	a2	a3	x1	x2	x3	y1	y2	y3	z1	z2	z3	t1	t2	t3
------------	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----

Here the final three attributes are boolean indicating whether an object can be thought of as an X, Y and Z.

## 8.6 CREATE THE DATABASE

After you have modeled a database by the above procedure and have drawn up a normalized relational model, the model should be converted into a database. The model shows which tables must be, the attributes that are primary keys and the foreign keys, and I would assume that the finished model is the following:



Back there are a few things that need to be addressed, primarily concerning each attribute.

First, there is the data types, where for each attribute you must selected a data type. It requires knowledge of the problem area and which values the individual fields should contain. In particular, you must consider the attributes for text fields, where you must specify a size. Moreover, you must consider what constrains to be defined for the individual data fields, and this is particularly

1. surrogate keys
2. where the fields may contain *NULL*
3. possible default values
4. cascading for foreign keys

Finally, there is the naming of both the tables and attributes. In principle, it is clear from the mapping, but often I will in the model for technical reasons, choose short names, but for the sake of SQL it can often be sensible with a little more attention to names. Some recommends the following practices

1. select relatively short but telling names to tables
2. use as name for a relational table (many-many) the names of the two tables referenced separated by an underscore
3. use the table name (in singular) followed by an underscore as a prefix for all attributes
4. use as name for a foreign key the name of the primary key of the table referenced

It can lead to quite long names, but it ensures the uniqueness of the names in the SQL expression, and thus expression that is easier to read, so the principle can be quite reasonable.

Finally is shown a script that creates the database to the library from the above design and naming conventions:



```
use palibrary;

drop table if exists books_authors;
drop table if exists books_users;
drop table if exists books;
drop table if exists publishers;
drop table if exists categories;
drop table if exists authors;
drop table if exists users;
drop table if exists titles;
drop table if exists zipcodes;

create table publishers (
    publisher_id int auto_increment not null,      # surrogate key for a
    publisher                                           # the publisher's name
    publisher_name varchar(50) not null,
    primary key (publisher_id)
);

create table categories (
    category_id int auto_increment not null,      # surrogate key for a
    category                                           # the category's name
    category_name varchar(50) not null,
    primary key (category_id)
);

create table authors (
    author_id int auto_increment not null,          # surrogat key for an
    author_firstname varchar(50),                    # the author's first
                                                    name
    author_lastname varchar(30) not null,            # the author's last
                                                    name
    author_text varchar(200),                        # additional
                                                    documentation
    primary key (author_id)
);

create table books (
    book_id int auto_increment not null,            # surrogat key for a
                                                    book
    book_isbn13 char(17),                            # ISBN with 13 digits
```

```

book_isbn10 char(13),           # ISBN with 10 digits
book_title varchar(255) not null, # the book's title
book_edition int,              # the book's edition
book_year int,                 # the book's release
                                year
book_pages int,                # number of pages in
                                the book
book_copies int,               # number of copies of
                                this book
category_id int,               # foreign key to
                                categories
publisher_id int,              # foreign key to
                                publishers
book_text text,                # description of this
                                book

foreign key (category_id) references categories (category_id),
foreign key (publisher_id) references publishers (publisher_id),
primary key (book_id)
);

create table books_authors (
    book_id int not null,        # foreign key to books
    author_id int not null,      # foreign key to
                                authors

    foreign key (book_id) references books (book_id) on delete cascade,
    foreign key (author_id) references authors (author_id) on delete
    cascade,
    primary key (book_id, author_id)
);

create table zipcodes (
    zipcodes_code char(4) not null, # zipcode
    zipcodes_name varchar(30) not null, # name of the town
    primary key (post_code)
);

create table titles (
    title_id int auto_increment not null, # surrogate key for a
                                title
    title_name varchar(50) not null,      # the title's name
    primary key (title_id)
);

```

```

create table users (
    user_id int auto_increment not null,          # surrogate key to
                                                    user
    user_email varchar(100) not null,             # the user's email
                                                    address
    user_passwd varchar(150) not null,            # the user's password
                                                    ( encrypted)
    user_firstname varchar(50) not null,          # the user's first name
    user_lastname varchar(30) not null,           # the user's last name
    user_address varchar(50) not null,            # the user's address
    zipcodes_code char(4) not null,               # the user's zipcode
                                                    (foreign key)
    user_phone varchar(20),                       # the user's phone
                                                    number
    title_id int,                                 # foreign key to
                                                    titles
    user_role int default 3,                      # the users role (user
                                                    authority)
    foreign key (post_code) references post (zipcodes_code),
    foreign key (title_id) references titles (title_id),
    primary key (user_id)
);

create table books_users (
    book_id int not null,                        # foreign key to books
    user_id int not null,                        # foreign key to users
    books_users_date date not null,              # when the book is
                                                    lent
    foreign key (book_id) references books (book_id) on delete cascade,
    foreign key (user_id) references users (user_id) on delete cascade,
    primary key (book_id, user_id)
);

```

## 9 MORE ON DATABASES

This chapter discusses a few more database concepts that were not accommodated in the previous book:

1. Transactions
2. Concurrency
3. Other database products

For all three topics there is a short presentation and for specific projects it may be necessary to find further information elsewhere.

### 9.1 TRANSACTIONS

In this section I will look at transactions. A transaction consists of several database operations to be performed, but in such a way that they are all executed correctly or none of the operations are performed at all. When performing multiple operations, the first operations may be correctly executed while a subsequent operation fails, and if this occurs, the first operations must be rolled back and the database left as if nothing has happened. Only if all operations are performed properly, can you confirm and the changes are visible in the database. I will show below how to implement transactions in ADO.NET, and I will, as example, look at the book database.

If you want to delete a book, you must first delete the book, but also delete the rows in the table *Written* that pertain to the book corresponding to that *Written* being a relationship table between *Book* and *Author*. One must be sure that deleting in both tables, as otherwise there is inconsistency in the tables, and deleting a book, can therefore be perceived as a transaction consisting of two operations. However, you can get the database management system to do the task, because in the table *Written*, *Written\_isbn* is the foreign key to *Book\_isbn* in *Book*, and if it is cascading delete, the database system will automatically delete dependent rows in *Written*. For the sake of the syntax of a transaction, I will use the example anyway. The following method has, as a parameter, the isbn to be deleted:

```
static bool Remove(string isbn)
{
    using (SqlConnection connection = new SqlConnection(..))
    {
        connection.Open();
        using (SqlTransaction transaction = connection.BeginTransaction())
        {
            try
            {
                using (SqlCommand command1 = new SqlCommand(
                    "DELETE FROM Written WHERE Written_isbn = @Isbn",
                    connection))
                {
                    command1.Parameters.Add(CreateParam("@Isbn", isbn, SqlDbType.NVarChar));
                    command1.Transaction = transaction;
                    using (SqlCommand command2 = new SqlCommand(
                        "DELETE FROM Book WHERE Book_isbn = @Isbn", connection))
                    {
                        command2.Parameters.Add(
                            CreateParam("@Isbn", isbn, SqlDbType.NVarChar));
                        command2.Transaction = transaction;
                        command1.ExecuteNonQuery();
                        command2.ExecuteNonQuery();
                        transaction.Commit();
                        return true;
                    }
                }
            }
            catch
            {
                transaction.Rollback();
            }
        }
    }
    return false;
}
```

Initially, the two database operations to be performed are defined:

1. an operation that deletes all the book references in the table *Written*
2. an operation that deletes the book

A transaction has the type *SqlTransaction*, and an object of this type is created by the method *BeginTransaction()* on the connection object. As the next step, the two operations must be linked to the transaction with a property, and then the operations are performed in the usual way. If it goes well, and that is no exception is raised, the transaction ends with a *Commit()*, which means that you acknowledge that it has been executed correctly and the database is then updated. If an exception occurs, the transaction must be rolled back, which is done with the method *Rollback()*.

That's basically what there is to say about transactions. As another example, I will look at a method that creates a book when the book is to be created based on

- isbn
- title
- released which is an integer
- edition which is an integer
- number of pages which is an integer
- the name of the publisher
- the name of the category
- one or more authors

In this case, the method does not have to validate the input, but the book must be created as a transaction that, in the worst case, may mean writing in all 5 tables:

- if the publisher does not already exist, a new publisher must be created
- if the category does not already exist, a new category must be created
- if an author does not already exist, a new author must be created

In any case, the operation must write in the *Book* table and in the table *Written*, and if you want everything to be either done or not done, you must perform all operations as a transaction. First, I have defined the following simple type, which represents an author:

```
class Author
{
    public string Firstname { get; set; }
    public string Lastname { get; set; }
}
```

Next the method can be written as:

```

static bool Create(string isbn, string title, int year, int edition,
int pages,
    string text, string publisher, string category, params Author[]
    authors)
{
    using (SqlConnection connection = new SqlConnection(..))
    {
        connection.Open();
        using (SqlTransaction transaction = connection.BeginTransaction())
        {
            try
            {
                int pub_id = GetPublisher(connection, transaction, publisher);
                if (pub_id < 0) pub_id =
                    InsertPublisher(connection, transaction, publisher);
                int cat_id = GetCategory(connection, transaction, category);
                if (cat_id < 0) cat_id = InsertCategory(connection,
                    transaction, category);
                InsertBook(connection, transaction, isbn, title, year, edition,
                    pages,
                    text, pub_id, cat_id);
                foreach (Author a in authors)
                {
                    int aut_id = GetAuthor(connection, transaction, a.Firstname,
                        a.Lastname);
                    if (aut_id < 0) aut_id =
                        InsertAuthor(connection, transaction, a.Firstname,
                            a.Lastname);
                    InsertWritten(connection, transaction, isbn, aut_id);
                }
                transaction.Commit();
                return true;
            }
            catch
            {
                transaction.Rollback();
            }
        }
    }
    return false;
}

```

The method creates a connection to the database and starts a transaction. Next, a number of other methods are called which perform various database operations. I do not want to show the code for these methods here, but each operation is linked to the transaction. If

one of the operations fails, an exception is raised and the catch handler performs a rollback and the method returns false. If, on the other hand, all operations are performed without errors, the book is considered to be properly created and a commit is executed, after which the database is updated.

### Disconnected ADO and transactions

Transactions can be used in conjunction with a data adapter, and this is actually even simpler than is the case in the above example. Below is an example, which adds three rows to the zip code table, but such that the three INSERT commands are executed as a transaction, and so that two of the rows have the same primary key, causing the transaction to fail:

```
static void Test2()
{
    SqlConnection connection = new SqlConnection("...");
    SqlTransaction transaction = null;
    try
    {
        connection.Open();
        SqlDataAdapter adapter =
            new SqlDataAdapter("SELECT * FROM Zipcodes", connection);
        DataTable table = new DataTable();
        adapter.Fill(table);
        transaction = connection.BeginTransaction();
        adapter.InsertCommand = CreateInsertCommand(connection,
            transaction);
        DataRow row = table.NewRow();
        row[0] = "8811";
        row[1] = "A small town";
        table.Rows.Add(row);
        row = table.NewRow();
        row[0] = "9911";
        row[1] = "A smaller town";
        table.Rows.Add(row);
        row = table.NewRow();
        row[0] = "9911";
        row[1] = "A big town";
        table.Rows.Add(row);
        adapter.Update(table);
        transaction.Commit();
        Console.WriteLine("OK");
    }
}
```



```
catch (Exception ex)
{
    transaction.Rollback();
    Console.WriteLine(ex.Message);
}
finally
{
    connection.Close();
}
}
static SqlCommand CreateInsertCommand(SqlConnection connection,
    SqlTransaction transaction)
{
    SqlCommand command = new SqlCommand(
        "INSERT INTO Zipcodes (Code, City) VALUES (@Code, @City)",
        connection);
    command.Parameters.Add("@Code", SqlDbType.VarChar, 4, "Code");
    command.Parameters.Add("@City", SqlDbType.VarChar, 30, "City");
    command.Transaction = transaction;
    return command;
}
```

Actually, there's not much new. The transaction is created as in the previous example, and the only new thing is that you need to define an update command to associate the transaction with the command. The fact that there are two rows with the same primary key is discovered when you try to update the database, where you get an exception that results in a rollback.

In this case, the transaction is actually not necessary. If you execute a *FillSchema()* on the adapter, you get information about the primary key, and you will get the error already when you insert a new row in the table in memory. Furthermore, it is rare that it is necessary to deal with transactions related to an adapter if the necessary schema information is extracted, but as shown in this example it is possible.

### Transactions and the entity data model

With an entity data model, the transaction concept is actually out of date, as the framework takes care of everything similar to *SaveChanges()* being executed as a transaction. Consider the following example, which use the ZIP code table with an entity data model. The example inserts three new rows into the table. When *SaveChanges()* is executed, the updates are ignored and none of the three changes are written to the database.

```
static void Test3()
{
    ContactsEntities entities = new ContactsEntities();
    try
    {
        entities.Zipcodes.Add(new Zipcodes { Code = "8811", City = "A
        small town" });
        entities.Zipcodes.Add(new Zipcodes { Code = "9911", City = "A
        smaller town" });
        entities.Zipcodes.Add(new Zipcodes { Code = "9911", City = "A big
        town" });
        entities.SaveChanges();
    }
    catch (Exception ex)
    {
        Console.WriteLine(ex.Message);
    }
}
```

## 9.2 CONCURRENCY

If an element (for example, a row) in a database needs to be changed, only a single program can do it, and while the change is on, others cannot be allowed to use the element. In practice, it is something that the database management system handles by locking it when a program needs to update an item. If a program wants to update an item and a lock has to be set but the item is locked due to another program, the first program must wait until the second is finished and the lock is released.

Of course, if there is only one program using the database, this is not something that you need to think about as the database management system manages it all, but if you have a situation where several programs use the same database, it is something that the program must deal with, as there is a high likelihood of concurrency issues where two or more programs will update the same database element. For example, if using disconnected access, chances are great:

- you use an adapter to fill a data set
- you manipulate the dataset locally and during that time another program can update the database
- you use the adapter to update the database and the other program's updates are lost

Thus, it can be stated that when multiple users try to change data at the same time, it is necessary to manage the order in which the changes are made. Otherwise, you may run the risk of one user's updates overwriting another user's updates, and thus the other user's updates being lost without the user being aware of it. In general, ADO.NET has three ways to handle concurrency control:

Pessimistic control where a row is locked and unavailable to others from it is read by the adapter until it is updated again. If you use that kind of control, there are no concurrency problems, but conversely there is a high risk that others will not be able to use the database.

Optimistic control where a row is locked and then is not available to others during the time it is updated. During the update, the row is compared to the contents of the database to see if it has changed after it has been read. If you try to update a row that has been changed, you get an error.

Last winner where a row is locked and unavailable to others only during the time it is updated, but no comparison is made to the contents of the database and thus the row is written with the possibility of a change made by another being overwritten.

A little differently, the three ways can also be explained as follows:

1. Pessimistic control is where I try to get everything I need and then others have to wait until I'm done.
2. Optimistic control is where I read data, execute my changes and hope that no others have changed in the meantime.
3. Last winner is where I save my changes and otherwise I don't care about everyone else and their work.

Typically, the reason for using pessimistic control is that you know that there is a high probability that there will be more people who will use the same rows, simply because the cost of putting a lock is much less than the cost of a rollback due to of a simultaneous conflict. Another reason for pessimistic control is that one cannot in any way allow another to change a row that is currently in use, for example in connection with a booking or equivalent.

However, pessimistic control cannot be used in connection with disconnected database access, since the connection is only open while data is being read, and again when the database is updated, and therefore no locks that apply during the intervening period can be used, nor would it be appropriate as this would prevent others from updating the database for long periods.

With optimistic control, locks are created to access data when there is an open connection to the database. It prevents others from updating the database at the same time. Data is always available except for the moment an update is performed. When an update is performed, the original version of a modified row is compared to the current row in the database. If the two are different, there is a concurrency error and the update is not completed. It is then up to the user program to decide what to do next.

At the last winner, the original content is not checked with the current value in the database, but the update is performed without notice. The result of this strategy is that lost updates may occur.

Basically, ADO.NET uses optimistic concurrency control, including for the sake of disconnected database access. Therefore, it is up to the program to implement logic for solving concurrency problems, and this can basically happen in two ways.

One solution is that the table to be updated has an additional column containing a timestamp. The value of such a column is updated by the database management system each time the contents of a row are changed. If this value is read and stored along with the rest of the row, as part of the update in a WHERE clause, it can test whether it matches the value in the database. If this is not the case, you get an exception, and the program can then do something and possibly make a rollback.

Another solution is that when you want to write a solution back to the database, you can then compare the original version of a row with the contents of the database, and if there is a deviation, there is a concurrency problem and you will get an exception. The advantage of this solution is that you do not need a timestamp column, but conversely, the SQL UPDATE statement becomes somewhat more complex - especially if it has to be written manually. If Visual Studio auto generate a database access such as for an entity data model, this solution is used for concurrency checking.

As an example on how to use concurrency control I will show a program called *Concurrency* which uses the database *Contacts* and the table *Addresses*. The *Main()* method is:

```

public static void Main(string[] args)
{
    SqlConnection connection = new SqlConnection(..);
    SqlDataAdapter adapter =
        new SqlDataAdapter("SELECT * FROM Addresses", connection);
    DataSet ds = new DataSet();
    adapter.Fill(ds);
    adapter.FillSchema(ds, SchemaType.Mapped);
    new SqlCommandBuilder(adapter);
    Show(ds.Tables[0]);
    Update(ds.Tables[0]);
    Test(connection, adapter, ds);
}

```

that uses an adapter to retrieve all persons from the *Addresses* table and prints the content on the screen using the method *Show ()*. It is a simple method that does nothing but iterates through a *DataTable* and I will not show the method here. The next statement calls a method *Update()*, where you can make updates to the table:

der sætter en adapter til *Addresses* tabellen og printer indholdet på skærmen med metoden *Show()*. Det er en simpel metode, som ikke gør andet end at gennemløbe en *DataTable*, og jeg vil ikke vise metoden her. Den næste sætning kalder en metode *Update()*, hvor man kan foretage opdateringer på tabellen:

```

private static void Update(DataTable table)
{
    string phone = Input("Enter phone number (Enter to Exit)");
    while (phone.Length > 0)
    {
        DataRow row = table.Rows.Find(phone);
        if (row != null)
        {
            row["Address"] = Input("Address");
            row["Zipcode"] = Input("Zip code");
            row["Title"] = Input("Job position");
        }
        else Console.WriteLine("Phone number not found...");
        phone = Input("Enter phone number (Enter to Exit)");
    }
}

```

The method works as follows: One must enter a phone number. If you enter blank, it means that the method stops. Otherwise, for a given phone number (primary key) you can update respectively the address, the postal code and job title. The most important thing to note is that all changes are made to memory and that the adapter has not yet updated the database. Thus, there is a source of a concurrency problem since the table can be changed by a program other than this program. That situation is easy to simulate. Before completing the above method by pressing Enter, you can modify the same rows by opening SQL Server Management Studio, which corresponds exactly to the database being changed by another program and thus a concurrency problem may have been created.

The last statement in Main () calls a method Test (), and from here you can (by simple comments) test different test methods. The first test method associates an update method with the adapter and then updates the database:

```
public static void Test1(SqlConnection connection, SqlDataAdapter
adapter,
    DataSet ds)
{
    adapter.UpdateCommand = CreateUpdateCommand1(connection);
    try
    {
        adapter.Update(ds);
    }
    catch (DBConcurrencyException ex)
    {
        Console.WriteLine("Another user has modified the database");
        DataRow[] rows = new DataRow[ex.RowCount];
        ex.CopyToRows(rows);
        foreach (DataRow row in rows) Console.WriteLine(row["Phone"]);
    }
    catch (Exception ex)
    {
        Console.WriteLine(ex.Message);
    }
}

private static SqlCommand CreateUpdateCommand1(SqlConnection
connection)
{
    SqlCommand cmd = new SqlCommand("UPDATE Addresses SET Address = @
Adr,
        Zipcode = @Zip, Title = @Job WHERE Phone = @Pnr AND Modified = @
Mod",
```

```
connection);  
cmd.Parameters.Add("@Adr", SqlDbType.VarChar, 50, "Address");  
cmd.Parameters.Add("@Zip", SqlDbType.VarChar, 4, "Zipcode");  
cmd.Parameters.Add("@Job", SqlDbType.VarChar, 50, "Title");  
cmd.Parameters.Add("@Pnr", SqlDbType.VarChar, 8, "Phone");  
cmd.Parameters.Add("@Mod", SqlDbType.Timestamp, 8, "Modified");  
return cmd;  
}
```

Regarding the update method, the last SQL parameter is the most important, and it is the one that tests (assuming the table has a *Timestamp* column) whether a row has changed after the adapter retrieves data. If so, you get a *DBConcurrencyException*, and it's up to the program to decide what to do next. In this case, the error handling consists of the program printing the phone number of the rows that have not been updated. It should be noted that changing multiple rows only updating the rows which results in a concurrency problem fails. The rest will be updated properly. However, by default, there is a row that cannot be updated due to a concurrency problem, then the update process is interrupted and all subsequent operations are not performed.

As mentioned above, you can test how it works by updating one or more rows, but before you finish the *Update()* method, you can open Management Studio for SQL Server and modify one of the same rows. When the adapter writes back to the database, an error will occur.

One can make the error handling a little better as shown in the next example. First, the following list is defined

```
private static List<string> errorList = new List<string>();
```

to be used for phone numbers for rows that are not updated properly. The test method is as follows:

```

private static void Test2(SqlConnection connection, SqlDataAdapter
adapter,
    DataSet ds)
{
    adapter.ContinueUpdateOnError = true;
    adapter.RowUpdated += new SqlRowUpdatedEventHandler(adapter_
RowUpdated);
    adapter.UpdateCommand = CreateUpdateCommand1(connection);
    try
    {
        errorList.Clear();
        adapter.Update(ds);
    }
    catch (Exception ex)
    {
        Console.WriteLine(ex.Message);
    }
    foreach (string phone in errorList)
        Console.WriteLine(phone + " could not be updated ");
}

```

First, you tell the adapter that in the case of an error (a row that cannot be updated) it should continue with any other rows. Next, an event handler is defined that is called each time a row is updated:

```

private static void adapter_RowUpdated(object sender,
SqlRowUpdatedEventArgs e)
{
    if (e.Status == UpdateStatus.ErrorsOccurred)
        errorList.Add(e.Row["Phone"].ToString());
}

```

If an error occurs for a particular row, the phone number is added to the list. The update method is the same as in the first example, and the result is that once the adapter has updated the database, keys are printed on the rows that are not updated.

The above way works fine, but it is based on the fact that the database table has a *Timestamp* column, which is not always possible. Below I will show another way to achieve the same, but the disadvantage is that you have to write a little more. The test method is almost identical to the first one, and the difference is that another update method is used:



```

private static SqlCommand CreateUpdateCommand2(SqlConnection
connection)
{
    SqlCommand cmd = new SqlCommand("UPDATE Addresses SET Address = @
    Adr,
        Zipcode = @Zip, Title = @Job WHERE Phone = @Pnr AND Address = @
        orgAdr
    AND Zipcode = @orgZip AND Title = @orgJob", connection);
    cmd.Parameters.Add("@Pnr", SqlDbType.NChar, 8, "Phone");
    AddParameter(cmd, "Adr", SqlDbType.NVarChar, 50, "Address");
    AddParameter(cmd, "Zip", SqlDbType.NChar, 4, "Zipcode");
    AddParameter(cmd, "Job", SqlDbType.NVarChar, 50, "Title");
    return cmd;
}
private static void AddParameter(SqlCommand cmd, string name,
SqlDbType type,
    int size, string column)
{
    SqlParameter param1 = new SqlParameter("@ " + name, type, size,
    column);
    SqlParameter param2 = new SqlParameter("@org " + name, type, size,
    column);
    param1.SourceVersion = DataRowVersion.Current;
    param2.SourceVersion = DataRowVersion.Original;
    cmd.Parameters.Add(param1);
    cmd.Parameters.Add(param2);
}

```

For each field that needs to be updated, two versions of the data element are assigned respectively the current value and the original value. In particular, note how these values are mapped to the command as parameters. Pay special attention to how to name the current and original value. The SQL update command itself is similar to the previous command, but in the WHERE section it now tests against the original values and there is no longer any *Modified* column. This concurrent check method requires code to be written, but if you allow Visual Studio to create to concurrent check code, it is the method used. Of course, it can also be combined with a handler, and the syntax is the same as above.

### 9.3 OTHER DATABASE PRODUCTS

In the previous book about databases I in all examples have used SQL Server Express. It is one of more database products from Microsoft. The product is free and can be downloaded

and installed on the user's computer. It is a full database server with all the features you would expect from a database server, but the intension is development or to test applications on the local machine. If you need a running database server to support your company you have to go another way and by and install one of Microsoft's other database products. For the programmer, however, there is no big difference and primarily consists in connecting to the database using another connection string.

There are also database products from other than Microsoft, but if your job is write a .NET application running on a Windows machine it is natural to think on Microsoft SQL Server. Other possibilities is Oracle, but I will also mention MySQL, which is a very used database server, and it also exists in a free version. I will not show how to use MySQL, but you just have to download and install the product, and all what in these books are said about databases also applies to MySQL, and again it is just the question to use the correct connection string.

This is the case anyway, but something else is so practice, and here one must recognize that there can be significant differences in different implementations of SQL. In particular, SQL Server's implementation differs in several places from standard SQL. However, most common SQL statements (as used in these books) are supported with the same syntax, but be aware that if you change database product you can encounter variations, especially as regards the use of functions.

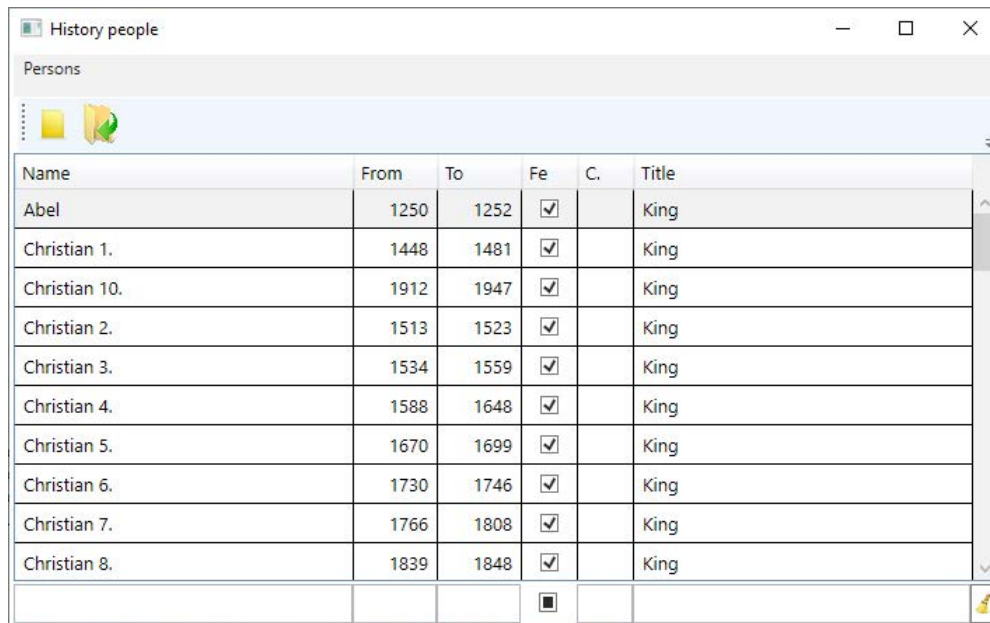
When you write an application that is installed and run on a single machine and the application needs a database to maintain data, it may be a little over killed to use a database server as a service running at the actual machine. For this reason there are file based databases which are not so much more than a class library with methods that can store data in a file, but using the same SQL syntax that is known from actual database servers. You do not get quite the same services as is the case with actual database servers, nor you get the same performance, but as seen from the programmer there is not quite the difference, and again it is to a large extent limited to another connection string. On the other hand, the application does not require a running database server, which makes it considerably easier to install the program on the user's machine.

There is more of this kind of database libraries, and one of the most popular is called SQLite, which I will present below.

## 9.4 HISTORY PEOPLE

In the previous book I write a program *HistoryPeople* which maintains a SQL Server database of historical persons. I will write another version of this program where the only difference is that this version instead should use a SQLite database.

I start with a copy of the project from the previous book, and when I run the program I get the following window:



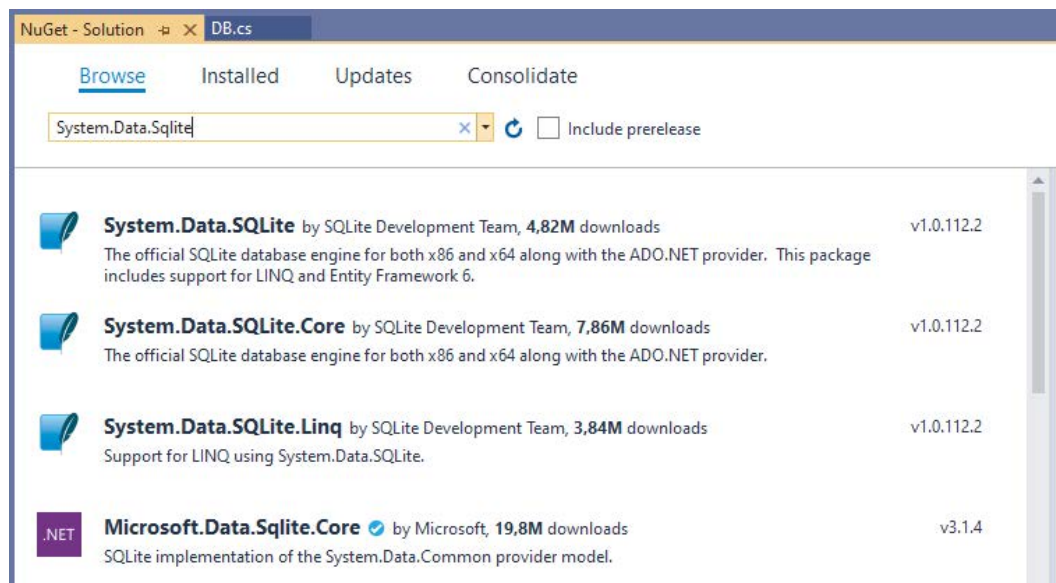
Name	From	To	Fe	C.	Title
Abel	1250	1252	<input checked="" type="checkbox"/>		King
Christian 1.	1448	1481	<input checked="" type="checkbox"/>		King
Christian 10.	1912	1947	<input checked="" type="checkbox"/>		King
Christian 2.	1513	1523	<input checked="" type="checkbox"/>		King
Christian 3.	1534	1559	<input checked="" type="checkbox"/>		King
Christian 4.	1588	1648	<input checked="" type="checkbox"/>		King
Christian 5.	1670	1699	<input checked="" type="checkbox"/>		King
Christian 6.	1730	1746	<input checked="" type="checkbox"/>		King
Christian 7.	1766	1808	<input checked="" type="checkbox"/>		King
Christian 8.	1839	1848	<input checked="" type="checkbox"/>		King
			<input type="checkbox"/>		

which in a *DataGrid* shows an overview for the persons in the database. The tool bar has two buttons used to create a new person or modify the selected person. To use a SQLite database the only thing I must do is to change the class *DB* in the data access layer as all code concerning the database is in this class.

SQLite is a class library (actual more), but the library is not included in the .NET framework and must be downloaded and added to the project. When you develop applications it is quite common that you need to use 3rd party libraries. These must be downloaded in the form of packages and must then be available for the current project. Visual Studio has a tool called *NuGet* that you can sometimes use and for SQLite you can. If you from the menu select

*Tools | NuGet Package Manager | Manage NuGet Packages for Solution...*

you get the following window where you can search for software packages:



The namespace for SQLite is called *System.Data.SQLite* and when I search for this namespace I get more packages for SQLite and I select the first. When I click *Install* the package is downloaded in a folder in your solution folder and Visual Studio inserts references to the necessary class libraries. Then SQLite is ready for use and you can use the library in the project.

In this case I in the class *DB* add the following statment

```
using System.Data.SQLite
```

and then I am ready to modify the class. The database file should be saved in the user's home directory and the class's constructor is changed:

```
public DB()
{
    filename =
        System.Environment.GetFolderPath(Environment.SpecialFolder.
        UserProfile) +
        "\\AppData\\Roaming\\History";
    if (!Directory.Exists(filename)) Directory.CreateDirectory(filename);
    filename += "\\People.sqlite";
    if (!File.Exists(filename)) Reload();
}
```

Note how to find the name of the user's home directory. The constructor tests if there under *AppData* is a directory *History* and if not creates this directory. The database file is placed here and is called *People.sqlite*. If the file does not exist the constructor calls the method *Reload()* to create the file and the database and initialise the database with persons from the XML document if the project folder *Data*:

```
public void Reload()
{
    try
    {
        List<Person> persons = LoadPersons();
        SQLiteConnection.CreateFile(filename);
        using (SQLiteConnection connection = GetConnection())
        {
            connection.Open();
            CreateTable(connection);
            using (SQLiteTransaction transaction = connection.
                BeginTransaction())
            {
                try
                {
                    foreach (Person pers in persons)
                    {
                        using (SQLiteCommand command = new SQLiteCommand("...",
                            connection))
                        {
                            command.Transaction = transaction;
                            command.Parameters.AddWithValue("@Name", pers.Name);
                            command.Parameters.AddWithValue("@From", pers.From);
                            command.Parameters.AddWithValue("@To", pers.To);
                            command.Parameters.AddWithValue("@Gender", pers.Gender);
                            command.Parameters.AddWithValue("@Title", pers.Title);
                            command.ExecuteNonQuery();
                        }
                    }
                }
                transaction.Commit();
            }
        }
        catch
        {
            transaction.Rollback();
            throw new Exception();
        }
    }
    if (PersonsLoaded != null)
        PersonsLoaded(this, new PaEventArgs<int>(persons.Count));
}
```

```
catch (Exception ex)
{
    throw new Exception("Error when reloading database");
}
```

The method starts to create a list with *Person* objects by reading and parsing the XML document, and it happens in exactly the same way as before. The the method create the file and the the database and create a connection for the database:

```
private SQLiteConnection GetConnection()
{
    return new SQLiteConnection("Data Source=" + filename +
        ";Version=3");
}
```

and note that the connection string is quite simple. When the connection is open the method calls a method *CreateTable()* which create the database table. It is a simple method that performs one SQL statement which add a table to the database:

```
private void CreateTable(SQLiteConnection connection)
{
    using (SQLiteCommand command = new SQLiteCommand("CREATE TABLE
    People
        (Hist_Id INTEGER PRIMARY KEY AUTOINCREMENT, Hist_Name VARCHAR(50)
        NOT NULL,
        Hist_From INTEGER, Hist_To INTEGER, Hist_Gender CHAR, Hist_Title
        VARCHAR(100),
        Hist_Country CHAR(2), Hist_Text TEXT)", connection))
    {
        command.ExecuteNonQuery();
    }
}
```

In principle, the method executes the SQL script used in the previous book to create the table. The a database with one tabel is created, and the method *Reload()* loops over all *Person* objects in the list and inserts rows in the table. You should note that it happens as before with the same SQL expression, just are the parameters for the expression defined in another way.

Also the other methods must be changed, but I will not show the code. When studying the code you should note that these are only minor changes and that the SQL statements are generally the same.

After these modifications of the class *DB* the program can run again, but now it uses a SQLite database instead of a SQL Server database.

## PROBLEM 1: THE WORLD

In the previous book in problem 2 you have written a program called *The World*. The program shows and maintains information about countries stored in a database with two tables:

1. a table with names for this world's continents
2. a table with some information about this world's countries

Maybe you should read the problem again to remember what the program is doing and the design of the database.

In this problem you must write the same program again with only one difference, that the program this time should use a SQLite database. To write the program you can follow the guidelines below:

1. Take a copy of the project from the book C# 6.
2. Open the project and use NuGet to add packages for SQLite.
3. The class *DB* has an empty constructor. You must add code such the constructor test if the database exists, and if not the constructor must create the database and insert a row in the table *Continent* for each continent. The constructor must then open a file browser where the user can select the data file with data for countries and initialize the table *Country* with data about countries.
4. You must then modify the methods in the class *DB* such they all uses the SQLite database.
5. If you have changed the datatype of the column *Area* from *Decimal* to *Real* you also have to change the type in the model class *Country* to *double* (and maybe also in a *Converter*).

Then the program should run again, but now using a SQLite database.

## 10 FINAL EXAMPLE: MYWINES

The program is an ordinary PC application that should maintain information about a private wine cellar, which is not entirely realistic. Should such a program be written in practice, you would probably write an app to a mobile phone. It should be ignored, and the purpose of the project is to use some of the substance that is treated in this and the previous book.

Compared with the programs that I have shown in the previous books, it is a relatively large program, and it is also a program that can be used in practice, if one has a wine cellar and feel a need to register the wine purchase and consumption. The program consists of many classes (about 90), and there are more than 25 windows (dialog boxes). Now it is all not as violent as it sounds. Most of the dialog boxes are simple, and the project architecture is Model-View-ViewModel.

When you have to write a program which, as in this example will consist of many dialog boxes, you can then try to reduce the code and the number of dialog boxes by parameter control the dialog boxes and apply them to multiple functions depending on the parameters that are transferred to the constructors. If you do that, you get less code and fewer classes and thus, in principle, a program that is easier to maintain, but however more complex classes, which can be difficult to understand, because you have dialog boxes that perform several functions. Conversely, a dialog box for every function mean that the number of classes becomes very large, which in turn may mean that you have to change in many places to maintain the program. It is a choice, and in the current solution I have tried to some extent to parameter control dialog boxes without letting it go beyond clarity.

You are encouraged to spend the time reading the following description of how the program is developed, and also to study the final result and including primary the program's code and to test the program. The following is a description of how the program was written and in this context what changes were made along the way, among other things to indicate that in practice, changes to a program and its requires always occurs during development. The description also includes comments on matters that should be given special attention when writing a program from scratch.

### 10.1 THE TASK

Many private wine lovers has a substantially wine storage - perhaps several hundred bottles or more. It requires control, including to ensure that the wines is not too old, but also for the sake of historical data with information about prices, ratings, etc. These historical



data are vital when buying the same wine again, but also to learn from experience and in general as documentation of purchase and consumption. There is thus not only a need to keep track of the current stock, but at least as much a need to store information about wines that are drunk and removed from the wine cellar.

The wines are acquired from different Danish suppliers - including grocery stores, but there may also be stock entries either in the form of direct import (that is from holidays or trips to wine countries) or gifts. The wines are purchased at greatly varying prices, and especially in connection with gifts, the price can be 0 (or lack of). It is common in the business, the granting of large discounts (and, arguably, even unrealistically large similar to that the list prices is not real). The price, and thus the discount granted is often determined by whether you buy single bottles or boxes (of 6 or 12 bottles).

The task is to write a program called *MyWines* that should be used to manage a private wine cellar. The program should be written in accordance with the following requirements / wishes.

In general, it should be possible to record the following information about a wine:

Information about the supplier (where the wine is purchased or obtained):

- The supplier's phone number
- The supplier's name (company name or other)
- The supplier's address
- The supplier's zip code
- The supplier's city
- The supplier's email address

Which supplier information that actual should be recorded may vary, because a wine can be a gift or be imported directly from from a vacation or in a similar way.

Information about the wine (that is facts about the product):

- The wine's name
- The producer's name
- The wine's type (white, red, rose, sparkling, dessert, port)
- The country where the wine is produced
- The country's code
- Wine district (for example Alsace, Barolo and son on)
- The grapes used for the production of the wine and in what quantities

- Classification if there is a classification (for example Chianti Classico Riserva DOCG)
- The wine's vintage
- Alcohol percent
- Size of the bottle / packing (in cl)

Information about the purchase:

- The date when the wine is purchased
- Expiry date (date when the wine latest should be drunk)
- The wine's price without discount
- Discount price
- The number of bottles purchased
- Quality (D = daily wine, H = house wine, Q = quality wine, C = cellar wine, X = cult wine)

The stock levels:

- Stock level (the current number of bottles)
- Revised quality (D, H, Q, C, X)
- Rating - an assessment consists of a date and a value from 0 to 100
- A description of the wine as a text

Every time there is used a bottle from the store, you have to register, which wine it is, how many bottles are taking and when.

For both the supplier and the wine itself you cannot expect that all the information is present. The same goes for how long the wine can be stored, discount and classification when a new wine must be added to the store, and in general you should be aware that sometimes you only have the knowledge that you can get from the label on the bottle.

The program should basically have the following features:

1. Register / create new wines (purchase)
2. Updates (consumption from the store and classifications)
3. Search functions

Because the work of records information about wines can be a great (there may be many details), you should aim for a solution where you have to enter as little as possible, and where it is easy to get the data available, which is already in the system. On the whole, you should priority that the program is easy to use and it is important that the program is robust and react sensibly by improper operation.

The program must have a search function where you can easily search wines from several criteria. You should aim for enhanced flexibility when searching, but conversely, the function should not be too complex. In connection with the search for wines, you should get:

- an overview of the total wine purchases within a given period and if necessary with the possibility of demarcation on country or by other criteria
- an overview of the total consumption within a given period and if necessary with the possibility of demarcation on country or by other criteria
- an overview of a wine's rating assessments over time (points)

It is also a desire that the program has a feature where you quickly can get a list of the wines that have reached the expiration date, and possibly has exceeded the expiration date, but also wines that should be drunk within the next time.

## 10.2 DEVELOPMENT PLANE

As mentioned many times, the development of any program starts with an analysis aimed at determining the requirements of the program, at least to the extent that one can address the development of key parts of the program. It is important to quickly get something done that you can present to the future users, but before you can get started you have to start with a development plan so that you have established some reference points that you can use to manage.

In this case the program will be developed through the following iterations:

1. Analysis
2. Database design
3. The model layer
4. The *MainWindow*
5. The dimension tables
6. The wine table
7. The search functions
8. The last things

The end of each iteration defines a reference point where the program is in a stable state. Initially, not all iterations will lead to new states that show progress to users, but one should strive to do so. In this case, the result of iteration 2 and 3 will not lead to a version of the program that, seen from the future users adds something new.

### 10.3 ANALYSIS

The program is basically a database program to maintain data on wines. There are very few uncertainties associated with the development and the analysis therefore includes only:

1. A data dictionary
2. A function list
3. A prototype

#### A data dictionary

The program must maintain information about 5 entities:

1. Suppliers
2. Wines
3. Purchases
4. Consumptions
5. Ratings

A supplier is typically an address of a store and often a web store. A cross in the third column means the value is required:

Suppliers			
Name	Type		Description
Phone	String, 20	x	The phone number
Name	String, 100	x	The name of the supplier
Address	String, 100		The address of the supplier
Zip code	String, 10		The zip code
Town	String, 50		The town for this zip code
Mail	String, 50		The supplier's email address
Text	String		An arbitrary description

A supplier can be anything, for example gifts, own import and so on.

A wine describes a particular wine:

Wines			
Name	Type		Description
Name	String, 100	x	The wine's name, is not unique
Producer	String, 100		The producer of the wine
Type	String, 20		The wine type (red, white, rose, port and so on)
Country	String, 50		The country where the wine is produced
District	String, 50		District within the country
Class	String, 50		If an official classification exists
Grape			Grape used for this wine, there can be several grapes
Year	Integer		The production year
From	Integer		The year when the wine should be drunk at the earliest
To	Integer		The year in which the wine should last be drunk
Size			Name of the packing, for example bottle, magnum and so on
Volume	Decimal		Alcohol percent
Aged	String, 100		How the wine is aged
Text	String		An arbitrary description

For wine types are used

- Red wine
- White wine
- Rose wine
- Sparkling
- Port
- Sherry
- Dessert

but it must be possible to add more types.

A grape is defined by a name and a number, which indicates the percentage of the grape contained in this wine.

A size consists of a name and volume indicated in centiliters. The following sizes are standard:

- Pony	18.75
- Split	37.5
- Standard	75
- Magnum	150
- Jeroboam	300
- Rehoboam	450
- Methusaleh	600
- Salmanazar	900
- Balthazar	1200
- Nebuchadnezzar	1500
- Melchior	1800
- Solomon	2000
- Sovereign	2500
- Primat	2700
- Melchizedek	3000
- Bag in box	250
- Bag in box	300
- Bag in box	500

but it must be possible to add more sizes.

A purchase occurs when there is access enter because when buying wine, the user receives gifts or other:

Purchases			
Name	Type		Description
Date	Date	x	The date for the purchase
Number	Integer	x	The number of units
Price	Decimal		The unit price

It has been decided that the price of the wine (if known) is always the price at which the wine is purchased and thus always the discount price. This means that any list price is considered uninteresting. For quality uses for the moment the following, but it must be possible to add more:

- D = daily wine
- H = house wine
- Q = quality wine
- C = cellar wine
- X = cult wine

A consumption must be registered every time a bottle is used and no longer exists in the cellar:

Consumptions			
Name	Type		Description
Date	Date	x	The date for the consumption
Number	Integer	x	The number of units

It must be possible to continuously assess the wines on a 100 percent scale. The same wine can be evaluated several times and for a particular wine there is thus a continuous assessment of the wine's changes.

Ratings			
Name	Type		Description
Date	Date	x	The date for the purchase
Rate	Integer	x	Rating from 0 - 100

## Functions

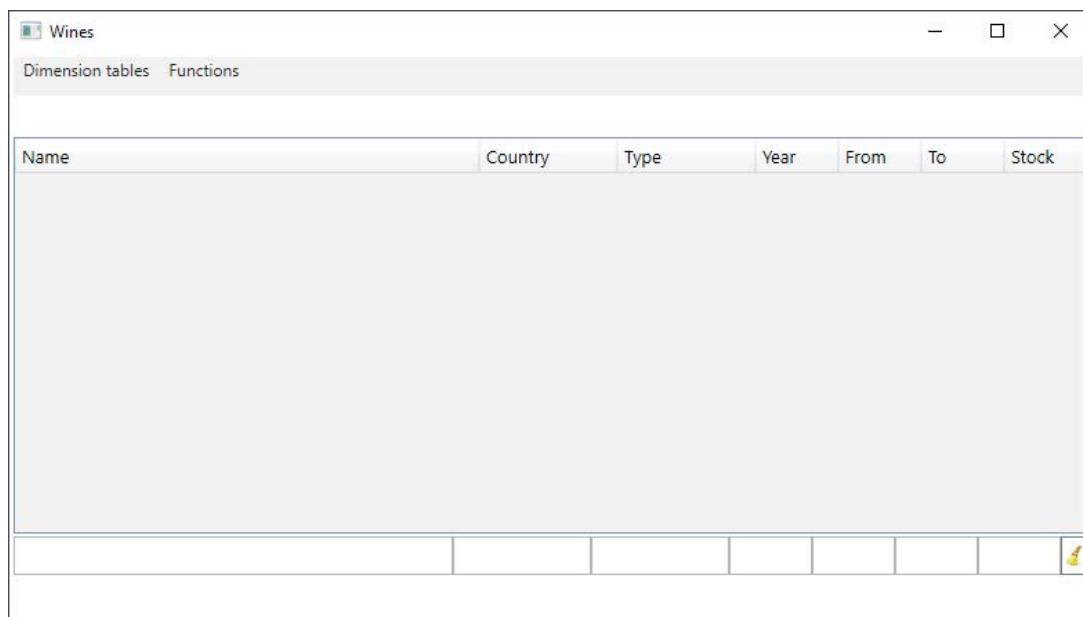
The program's main functions are:

1. *MainWindow*, which in a grid shows an overview on all the wines in the cellar with positive stock.
2. Maintaining the database and especially tables like the wine table depend on as (also called dimension tables).
3. Registration of purchases.
4. Consumption registration.
5. Registration of assessments (points).

6. Search.
7. Warning about wines to drink.

## Prototype

To create a prototype I have created a Visual Studio project called *MyWines*. It is a WPF application and for the moment the project only contains the *MainWindow*, and when you run the program the following window opens:



The window should in a *DataGrid* show an overview over all wines in the stock, and for each column is defined a filter. The window has a menu bar where the first menu has all the functions to maintain the database, while other functions are in the other menu. As so it is a classic design for a database application.

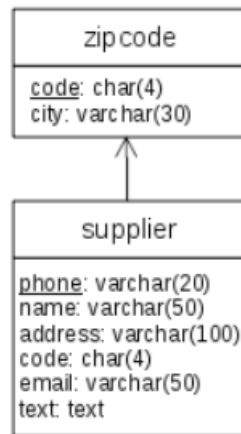
Of course, a prototype as shown here is trivial, but the development of the prototype means:

1. You have something that you can show to the future users and that can form the basis for a discussion of whether you have understood the task correctly.
2. You have a Visual Studio project which is ready for future development.
3. In Visual Studio it is quick and simple to develop a prototype as shown here and the work is not wasted, as the continued development will typically be based on the prototype.



## 10.4 DATABASE DESIGN

I will start the development with design of the database. If I look at the above description of the task, the database should contain information about suppliers, and the analysis of the description results in two tables:



Basically, it is an address book, and here you will usually place the zip code and the city name in a table for zip codes, and then let *code* be a foreign key in the table *supplier*. The information about the supplier has been expanded with a new value, where it is possible to register a description of the supplier. It has been decided to use the supplier's telephone number as the primary key. If the user creates suppliers without a telephone number (for example, for gifts or similar), the user may choose pseudo numbers, but such suppliers are perceived as exceptions and there will be only a few of them.

The primary database table is a table *wine*, which should contain information about a particular wine, thus a wine to be included in the cellar. This table would have among others the following columns:

wineid	int	# surrogate key
name	varchar(50),	# the wine's name
year	int?,	# the wine's production year
year1	int?,	# the year when the wine can be drunk
year2	int?	# the year when the wine latest must be drunk
percent	double,	# alcohol percent
storing	varchar(100),	# storing on barrel / tank / bottle
amount	int,	# current amount
text	text,	# description of the wine

and here it is also agreed that it should be possible to register a description of a wine. A wine has to be identified by a surrogate key, and it is agreed that two wines are considered different, if they do not have the same production year. Furthermore, for each wine it must be possible to define

- supplier
- packing (bottle, bib, other)
- producer
- production country
- district
- classification
- wine type
- wine category

Here are suppliers already described and the table wine must have a foreign key to the *supplier*. The 7 other information are in principle text, but it has been decided to register the individual information in their own table. The argument is partly that several wines must be registered with the same value, and second, that it can be difficult to ensure that the same text is spelled the same way each time, and finally you have for packing and district to register an additional information which is respectively the packing's volume and the district's country. For the type of a wine and wine categories the description of the task defines a natural primary key (one letter), but for the other 5 I has to choose a surrogate key. The design should therefore be extended with 7 other but simple tables (these tables are sometimes called dimension tables) and the table wine must have additional 8 foreign keys.

There should also be a dimension table for grapes, where a grape must be stored as a name and a surrogate key. However a wine can be produced on several grapes and as so there must be a many-to-many relation between grapes and wines:

Cuve
Wineid: Int Grapeid: Int Percent: int

where *Wineid* and *Grapeid* are a composed primary key and are foreign keys to the tables for wines and grapes.

There is also a need for a table to purchases:

Purchase
Wineid: int Date: DateTime Number: int Price: double

where a purchase should be identified by a surrogate key and a the table must have a foreign key to *wine*. When the table has a surrogate key it is because it should be possible to buy the same wine several times the same day.

Similarly, there must be a table with information on consumption of wines:

Consumption
Wineid: int Date: DateTime Number: int

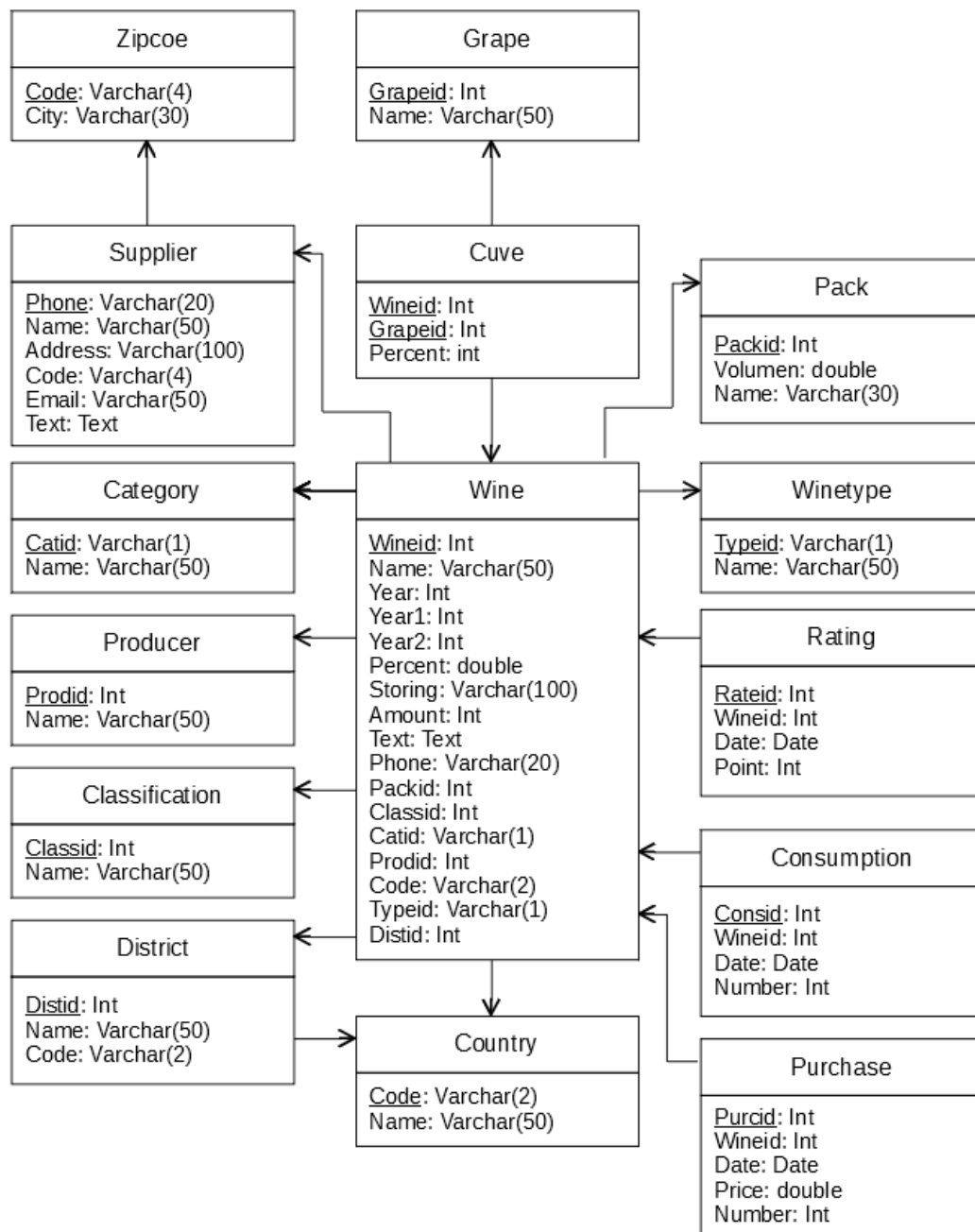
when rows in this table also must be identified be a surrogate key corresponding to the possibility of consuming the same wine several times on the same day.

Finally, there must be a table to record the user's ratings by the 100 points scale:

Rating
Wineid: int Date: DateTime Point: int

A database design as above requires several decisions which can easily prove to be inappropriate at a later time and, if necessary, the design must of course be changed. It should be emphasized that changes to the database design can later result in significant changes to the program and code written, and thus it pays to be extremely careful about the design and follow the guidelines described in the previous book. In the present case, I did not do so directly (the database has several tables but is in fact relatively simple), but the following diagram shows a model for the database where an implicit normalization has been made. However, it is important to note that for just a slightly extensive database, it is difficult to end up with a design where it is not necessary to adjust later, but it pays to be careful in this place and consider the choices that are made.

After these considerations, the design of the database can be illustrated as:



The database can then be defined by the following script:

```
CREATE TABLE Zipcode (  
    Code VARCHAR(4) PRIMARY KEY,  
    City Varchar(20) NOT NULL);  
  
CREATE TABLE Supplier (  
    Phone VARCHAR(20) PRIMARY KEY,  
    Name VARCHAR(50) NOT NULL,  
    Address VARCHAR(100),  
    Code VARCHAR(4),  
    Email VARCHAR(50),  
    Text TEXT,  
    FOREIGN KEY (Code) REFERENCES Zipcode(Code) ON DELETE SET NULL);  
  
CREATE TABLE Category (  
    Catid VARCHAR(1) PRIMARY KEY,  
    Name VARCHAR(50) NOT NULL);  
  
CREATE TABLE Producer (  
    ProdId INTEGER PRIMARY KEY IDENTITY(1,1),  
    Name VARCHAR(50) NOT NULL);  
  
CREATE TABLE Classification (  
    ClassId INTEGER PRIMARY KEY IDENTITY(1,1),  
    Name VARCHAR(50) NOT NULL);  
  
CREATE TABLE Country (  
    Code VARCHAR(2) PRIMARY KEY,  
    Name VARCHAR(50) NOT NULL);  
  
CREATE TABLE District (  
    DistId INTEGER PRIMARY KEY IDENTITY(1,1),  
    Name VARCHAR(50) NOT NULL,  
    Code VARCHAR(2),  
    FOREIGN KEY (Code) REFERENCES Country(Code) ON DELETE SET NULL);  
  
CREATE TABLE Pack (  
    PackId INTEGER PRIMARY KEY IDENTITY(1,1),  
    Name VARCHAR(30) NOT NULL,  
    Volumen REAL NOT NULL);
```

```
CREATE TABLE Winetype (  
    Typeid VARCHAR(1) PRIMARY KEY,  
    Name VARCHAR(50) NOT NULL);  
  
CREATE TABLE Grape (  
    Grapeid INTEGER PRIMARY KEY IDENTITY(1,1),  
    Name VARCHAR(50) NOT NULL);  
  
CREATE TABLE Wine (  
    Wineid INTEGER PRIMARY KEY IDENTITY(1,1),  
    Name VARCHAR(50) NOT NULL,  
    Year INTEGER,  
    Year1 INTEGER,  
    Year2 INTEGER,  
    Percent REAL,  
    Storing VARCHAR(100),  
    Amount INTEGER NOT NULL,  
    Text TEXT,  
    Phone VARCHAR(20),  
    Packid INTEGER,  
    Classid INTEGER,  
    Catid VARCHAR(1),  
    Prodid INTEGER,  
    Code VARCHAR(2),  
    Typeid VARCHAR(1),  
    Distid INTEGER,  
    FOREIGN KEY (Phone) REFERENCES Supplier(Phone) ON DELETE SET NULL,  
    FOREIGN KEY (Packid) REFERENCES Pack(Packid) ON DELETE SET NULL,  
    FOREIGN KEY (Classid) REFERENCES Classification(Classid) ON DELETE  
    SET NULL,  
    FOREIGN KEY (Catid) REFERENCES Category(Catid) ON DELETE SET NULL,  
    FOREIGN KEY (Prodid) REFERENCES Producer(Prodid) ON DELETE SET NULL,  
    FOREIGN KEY (Code) REFERENCES Country(Code) ON DELETE SET NULL,  
    FOREIGN KEY (Typeid) REFERENCES Winetype(Typeid) ON DELETE SET NULL,  
    FOREIGN KEY (Distid) REFERENCES District(Distid) ON DELETE SET  
    NULL);  
  
CREATE TABLE Cuve (  
    Wineid INTEGER,  
    Grapeid INTEGER,  
    Percent INTEGER,
```

```
PRIMARY KEY (Wineid, Grapeid),
FOREIGN KEY (Wineid) REFERENCES Wine(Wineid) ON DELETE CASCADE,
FOREIGN KEY (Grapeid) REFERENCES Grape(Grapeid) ON DELETE CASCADE);

CREATE TABLE Rating (
  Rateid INTEGER PRIMARY KEY IDENTITY(1,1),
  Wineid INTEGER NOT NULL,
  Date DATE NOT NULL,
  Point INTEGER NOT NULL,
  FOREIGN KEY (Wineid) REFERENCES Wine(Wineid) ON DELETE CASCADE);

CREATE TABLE Consumption (
  Consid INTEGER PRIMARY KEY IDENTITY(1,1),
  Wineid INTEGER NOT NULL,
  Date DATE NOT NULL,
  Number INTEGER NOT NULL,
  FOREIGN KEY (Wineid) REFERENCES Wine(Wineid) ON DELETE CASCADE);

CREATE TABLE Purchase (
  Purcid INTEGER PRIMARY KEY IDENTITY(1,1),
  Wineid INTEGER NOT NULL,
  Date DATE NOT NULL,
  Price REAL,
  Number INTEGER NOT NULL,
  FOREIGN KEY (Wineid) REFERENCES Wine(Wineid) ON DELETE CASCADE);
```

In this case, the script really should not be used, but I would still recommend writing such a script and I find that the work of writing the script is well done. The script is written in Sql Server Management Studio and that means I get a syntax check of the script. The most important thing, however, is that the script documents the database, including data types, primary keys, and which columns may contain null values. Another important thing is the documentation of foreign keys.

The program is a personal program, and as database product it is decided to use SQLite. This means that in order to install and use the program it is not necessary first to ensure that there is a running database server and create a database on the server. As so I will finish the design with a program which creates the database.

The database should consists of 15 tables. To create the database I have create a console application project called *CreateDB*. I will not show the code here, but the program creates the 15 tables when the database is created in the user's home directory:

```
static void Main(string[] args)
{
    filename = System.Environment.GetFolderPath(Environment.SpecialFolder.
        UserProfile) + "\\AppData\\Roaming\\MyWines";
    if (!Directory.Exists(filename)) Directory.CreateDirectory(filename);
    filename += "\\MyWines.sqlite";
    CreateDatabase();
    CreateTables();
}
```

Perhaps it should later be possible, as an option for the program, to place the database somewhere else.

The program also inserts data in four tables:

- *Zipcode*
- *Winetype*
- *Category*
- *Pack*

Finally, the program creates a single supplier, a single country and three wines. This data should not be inserted into the database, but it is included here to have some data in the database while developing the program.

You should note that to create this program I have used the database script from the previous iteration.

## 10.5 THE MODEL LAYER

I am now ready to start on the development of the program and have created a copy of the project from the analysis called *MyWines0*. I will begin with the model layer and write the first model classes, when a model class should model a row in a database table. In this iteration I will add model classes to the *Models* folder. So far, I will concentrate on the table *wine* and the tables which it refers, and I will write the following model classes:

- *Zipcode*
- *Supplier*
- *Producer*
- *Category*
- *Classification*



- *Winetype*
- *Grape*
- *Country*
- *District*
- *Pack*
- *Wine*
- *Cuve*

Here is the last a helper class for *Wine* and represents the grapes used for the current wine. Except the last they are all simple and vary only in terms of the fields they contains. As an example is shown the class *District* that is modeling the table *district*:

```
using System.Collections.Generic;

namespace MyWines.Models
{
    public class District
    {
        public District()
        {
            Wines = new List<Wine>();
        }

        public int Id { get; set; }
        public string Name { get; set; }
        public Country Country { get; set; }
        public List<Wine> Wines { get; private set; }

        public override bool Equals(object obj)
        {
            if (obj == null) return false;
            if (obj is District)
            {
                District other = (District)obj;
                return Id == other.Id;
            }
            return false;
        }

        public override int GetHashCode()
        {
            return Id;
        }
    }
}
```

The class is simple since it only contains properties for the columns in the database table, but note that the class overrides *Equals()* so that two objects are equal, if the corresponding rows in the database have the same primary key. You should note the property for the column *Code* has the type *Country* and as so references a *Country* object as *Code* is a foreign key to the table *Country*. This property may be *null* since it is not a requirement that a row in the *District* table refers to a country. Also note that the class has a list with *Wine* objects which are references for these wines that has a foreign key for this district. In this way a *District* object has a collection with references for all wines which references this *District*.

The class *wine* is basically designed in the same way, but it takes up a lot more, as the table *wine* has many columns. The class is:

```
using System.Collections.Generic;

namespace MyWines.Models
{
    public class Wine
    {
        public Wine()
        {
            Cuves = new List<Cuve>();
            Ratings = new List<Rating>();
            Purchases = new List<Purchase>();
            Consumptions = new List<Consumption>();
        }

        public int Id { get; set; }
        public string Name { get; set; }
        public int? Year { get; set; }
        public int? Time { get; set; }
        public double? Percent { get; set; }
        public string Storing { get; set; }
        public int Amount { get; set; }
        public string Text { get; set; }
        public Pack Pack { get; set; }
        public Supplier Supplier { get; set; }
        public Producer Producer { get; set; }
        public District District { get; set; }
        public Country Country { get; set; }
        public Classification Classification { get; set; }
        public Category Category { get; set; }
        public Winetype Winetype { get; set; }
        public List<Cuve> Cuves { get; private set; }
        public List<Rating> Ratings { get; private set; }
        public List<Purchase> Purchases { get; private set; }
        public List<Consumption> Consumptions { get; private set; }
```

```
public override bool Equals(object obj)
{
    if (obj == null) return false;
    if (obj is Wine)
    {
        Wine other = (Wine)obj;
        return Id == other.Id;
    }
    return false;
}

public override int GetHashCode()
{
    return Id;
}
}
```

Here you should note how the columns for the foreign keys are defined as a reference to an object of that model type. That is, if you, for example, consider the supplier, it is not a *String* (for the primary key *phone*), but a reference to a *Supplier* object, when the table *Wine* has a foreign key to the table *Supplier*. The class also defines four collections which contains objects of the types:

- *Cuves*
- *Purchases*
- *Consumptions*
- *Ratings*

These collections represents tables in the database, which all has a foreign key to the table *Wine* and the collections contains an object for each reference. As an example contains the list *Purchases* an object of the type *Purchase* for each purchase of this wine.

With these classes ready I have expanded the project with a *DataAccess* folder with only one class called *DB*. The class is defined as a singleton and defines a collection for objects representing the rows in the database tables (except for the table *Cuve*, since this table is a many-to-many relationship between the tables *Wine* and *Grape*):

```
public List<Category> Categories { get; private set; }
public List<Classification> Classifications { get; private set; }
public List<Consumption> Consumptions { get; private set; }
public List<Country> Countries { get; private set; }
public List<District> Districts { get; private set; }
public List<Grape> Grapes { get; private set; }
public List<Pack> Packs { get; private set; }
public List<Producer> Producers { get; private set; }
public List<Purchase> Purchases { get; private set; }
public List<Rating> Ratings { get; private set; }
public List<Supplier> Suppliers { get; private set; }
public List<Wine> Wines { get; private set; }
public List<Winetype> Winetypes { get; private set; }
public List<Zipcode> Zipcodes { get; private set; }
```

When the private constructor is performed these collections are initialized from reading the tables in the database, and the collections are then an object oriented model of the database. For each table there are three methods for the three database operations INSERT, UPDATE and DELETE, that is 42 operations in all (there is no operation for the table *Cuve* as it represents a many-to-many relation between the two tables *Wine* and *Grape*). These operations are basically simple, but they must also maintain the above object oriented model, and if a database operations fails there is a risk that the model does not reflect the current state of the database. Most of the operations are performed as a transaction, and if a database operation fails the transaction is rolled back and the method raise an exception. Is the database operation is performed without error the model is updated and the method performs a commit of the transaction.

The class *BD* has also other operations and and especially a method that creates the database. The code for this method is copied from the program *CreateDB*. This method is intended for a later extension of the program, where it should be possible to determine where the database should be stored and also as an option to perform a reset of the database, if desired.

The *DB* class is a comprehensive class with many methods, but you should note that in reality I have written my own entity model. There may not be many good reasons for that, but it gives you an idea of what an entity model is and what code are auto generated if you use an entity model.

As you read the code you should note that I have used LINQ somewhere, which is first introduced in the next book. The code is quite simple to understand and when you see the code you might not even think about that it is LINQ, as I have only used extension methods for the collection classes alone.

## A change of the database design

The design of the database compared to the previous iteration has changed somewhere. The table *Wine* has a column *Amount* whose contents indicate the inventory of the wine. This value must be updated every time there is a purchase or a consumption and the value is in principle just a summary of the records in the two tables *Purchase* and *Consumption*. This solution is unfortunate if it is also should possible to change the value of already registered purchases and consumptions, including deleting already registered purchases and consumptions.

The problem is that it is difficult to ensure that the value of *Amount* is accurate or reflects the current transactions and it is therefore decided to completely remove this column from the table *Wine*. The object *Wine* still has a property *Amount*, which is then initialized when the contents of the database are loaded and updated every time there is a purchase or consumption.

## Test

Before I continues with the next iteration, the class *DB* must be tested to be sure it works without errors. The main problem is to test that the memory model of the database is properly maintained. To test the class *DB* I have added a unit test project with one test class. The class has 11 test methods, and test starts to create a new database initialized as in the program *CreateDB*.

It is not simple to write these test methods, but it is worth the work and the test has actually found several errors. However, the most important thing is that in future changes to the *DB* class, the test can be immediately repeated and possibly expanded with new test methods.

## 10.6 THE MAINWINDOW

I will then write the program's *MainWindow*, and this iteration also starts creating a copy of the project from the previous iteration. The copy is called *MyWines1*. It is a short iteration, where the job alone is to implement the window from the prototype and thus to show the wines in the window.

After performed the unit test in the previous iteration there are no wines in the database and I has to execute program *CreateDB* again, but before it must be updated as the column *Amount* in the table *Wine* is removed.

To implement the window for this iteration I have create a folder *ViewModels* and to this folder copied the two classes

1. *RelayCommand*
2. *ViewModelBase*

from previous projects. I have then added two classes:

1. *ViewModelWine* as view-model for the class *Wine*
2. *ViewModelMain* as a view-model for *MainWindow*

These two classes are in principle identical to similar classes from early projects, and back there is only to bind elements of the XML code for the user interface to the view-model and to instantiate an object of the model in code behind. Both classes will be changed in later iterations. Then the program can run again and show the three wines in the database, and the window also has a filter:



The window has a tool bar for the most common functions:

1. Create wine
2. Show / modify wine
3. Purchase wine
4. Consume wine
5. Rating of wine
6. Search

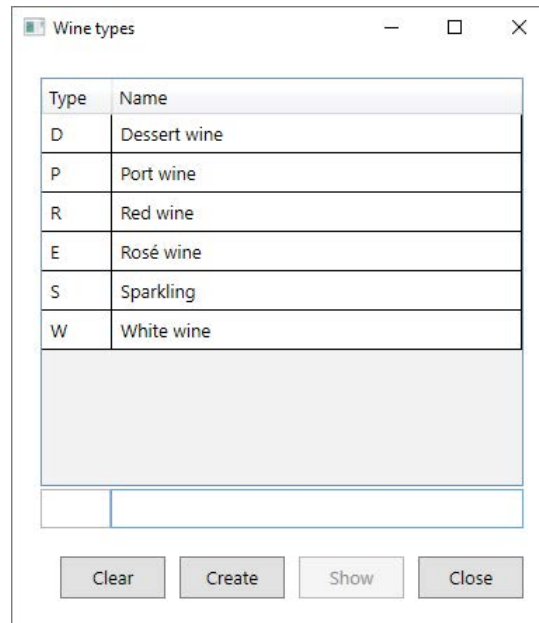
The menu has the following functions (where 6 functions are the same as the functions in the tool bar):

- The wines properties
  1. Create wine
  2. Maintenance selected wine
  3. Maintenance of suppliers
  4. Maintenance of producers
  5. Maintenance of countries
  6. Maintenance of districts
  7. Maintenance of classifications
  8. Maintenance of grapes
  9. Maintenance of wine types
  10. Maintenance of wine categories
  11. Maintenance of bottles / packings
  12. Maintenance of zip codes
- Other functions
  1. Advanced Search
  2. Purchase
  3. Consumption
  4. Rating wine

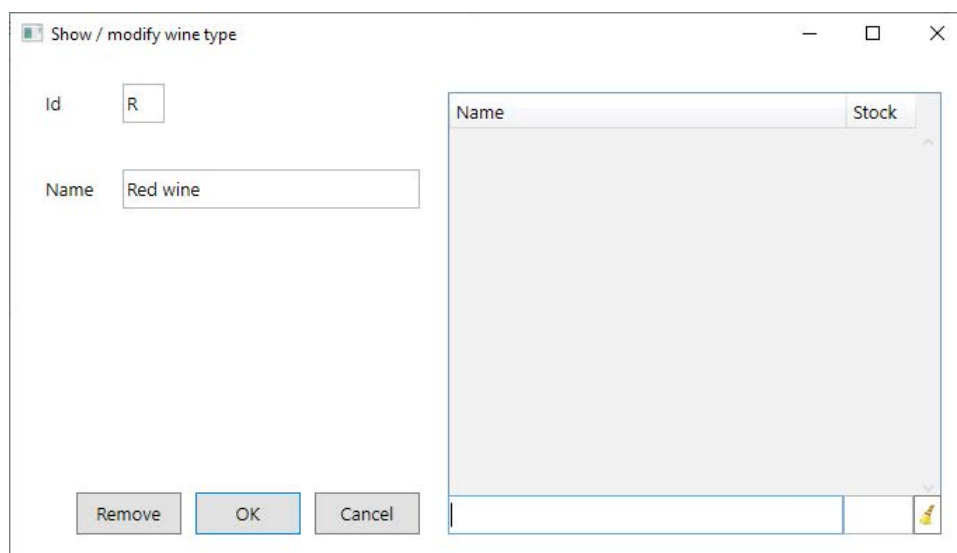
The result of this iteration is modest and is primarily an improvement of the prototype, but also an extension of the prototype so that the program uses the model. Even so, the user is clearly moving forward and gives the user a good idea of what the finished program will look like.

## 10.7 THE DIMENSION TABLES

The iteration starts to create a copy of the project from the previous iteration. The copy is called *MyWines2*. In this iteration I implements maintenance the dimension tables, and thus all functions under the menu *Dimension tables*, except for the first two. All functions works in principle the same way, and I will as an example look at *Maintenance of wine types*. If you click the menu item, the program opens the following dialog box:



The dialog box shows a *DataGrid* with all wine types and such the content of the table *Winetype* in the database and that is exactly the content of the collection *Winetypes* in the model. At the bottom is defined a filter to the table. The buttons are used to clear the filter and create a new wine type. The button that is disabled id used to open another dialog box where you can edit the selected wine type (if a row in the *DataGrid* is selected), and the last button close the window. You can also open the edit window by double-clicking on a line in the table, you get a dialog box where you can edit (and delete) a wine type, for example:

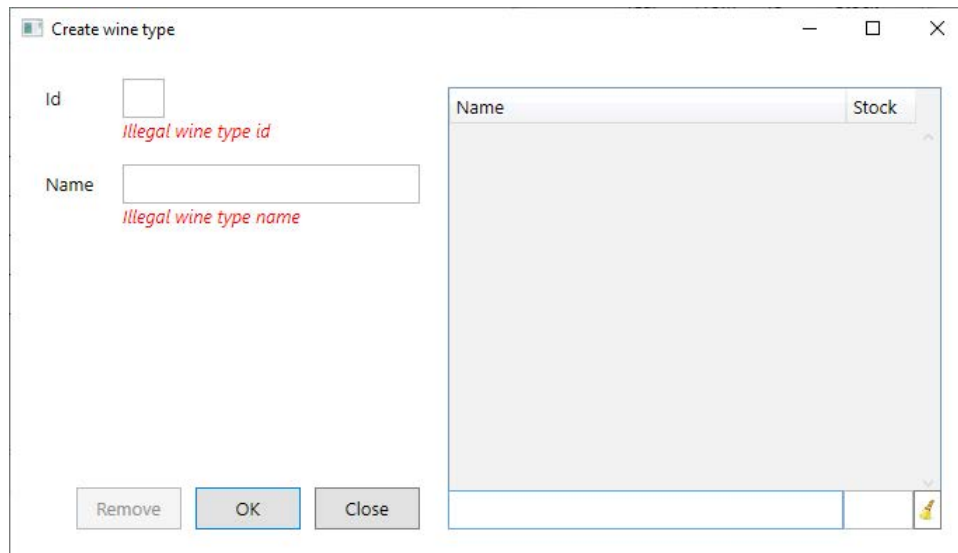


The field with the *Id* is read-only, but you can change the name, and you can remove the wine type. The *DataGrid* shows a table with all wines of this type, in this case all red wines and number of units at stock. The *DataGrid* has a filter, and if you double-click on a wine



in the grid the program opens the window for that wine (this window is the program's most complex window and will only be implemented in the next iteration).

If you in the window showing all wine types click on the button *Create* the program opens the same window as above:



The difference is that you now must enter the *Id* (and the name as a wine type must have a name) and you cannot remove the wine type (it is not created). When you have entered a wine type and click *OK* the window will be blank and remain open so you can enter the next wine type.

To implement this function the project is expanded with 5 new classes, three classes in the *ViewModels* layer and two classes in the *Views* layer:

1. *WinetypeModel*
2. *ViewModelWinetypes*
3. *ViewModelWinetype*
4. *WinetypesWindow*
5. *WinetypeWindow*

The first is an encapsulation of the model class *Winetype* so that the user interface can bind to the class. This means that the class inherits the class *ViewModelBase* and implements the interface *IDataErrorInfo*. It is also this class which has methods to perform the three database operations INSERT, UPDATE and DELETE for objects of the type *Winetype*.

The next class is a *ViewModel* for the first of the above two windows. The primary task for this class is to implement the filter and the commands for the buttons.

The third class is the *ViewModel* for the other of the above two windows, and it must in the same way implements the filter for the *DataGrid* and the commands. The class also define a property of the type *WinetypeModel* which is used in XML to bind components to the model.

The last two classes are view classes and defines the two windows. In this case both windows are simple and there is not much to note. The only thing is double-click on a row in a *DataGrid*. It is possible to bind this operation to a command in the *ViewModel*, but I have used the more direct way and from code behind call a method in the *ViewModel* from an event handler for double-click. It is not completely in line with the MVVM pattern since I put program logic in code behind, but it is a simple and very direct solution. A pattern is good, but conversely, using the pattern it must not complicate the solution and result in code that is difficult to understand. The problem here is that there is no direct property for double-click.

This iteration should implements 9 other functions (in all 10 function in the first menu). All the other functions are implemented in the same way, and that means I have to write 5 classes for each of the next 9 function. The functions all looks like each other and the difference is to maintain different objects with a few differences with respect to the properties of the objects. To implement the functions you can at least go two ways:

1. You can try to parameterize the classes so the windows and to some extent also the *ViewModel* classes can be used to maintain several types of objects. This means that the program will have fewer classes and thus you will need to write less code, and in principle the program will be easier to maintain. On the other hand, the code becomes more complex and less transparent, which can make it difficult for others to maintain the code in the future.
2. You can write 5 classes for each function. It's quite simple as it are simple classes, but there is many classes to write and as so a lot of code to write. However, the classes are similar to each other so much that you can go far with copy / paste, but that does not change that in future there will be many classes to maintain, but classes that are easy to understand.

I have chosen the last solution and the result is that I have written another 45 classes, but to a large extent by copying the code from the classes for maintenance wine types.

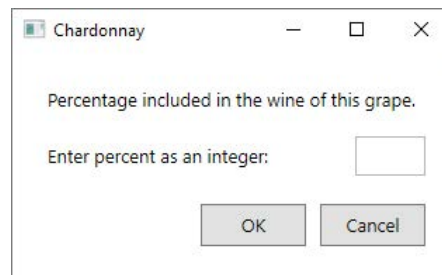
Everyone who writes programs uses copy / paste. Typically, you should write a code that resembles existing code, and then copy the existing code and correct it. However, it is not risk free, and not only that, there is an extremely high likelihood that you will not correct the code everywhere, and that there are simply places where you overlook statements that need to be changed. Everyone who has programmed for just a little longer has tried it and has similarly spent an incredibly long time locating errors due to copy / paste. The errors that the compiler finds are not the problem, but they are runtime errors that are either expressed by the program stopping with an exception or the program giving a wrong result. I'm not saying that you should not copy / paste, just be careful, because the chance of mistakes is great. It will be understood after a few times spending half a day and maybe some of the night to find that kind of mistakes.

## 10.8 MAINTENANCE OF WINES

The iteration starts with a copy of the project from the previous iteration and the copy is called *MyWines3*. In this iteration the program should be extended with a function used to maintain data for a wine and then create a new wine and update existing wines. This means that the iteration should implements the last two menu items in the first menu.

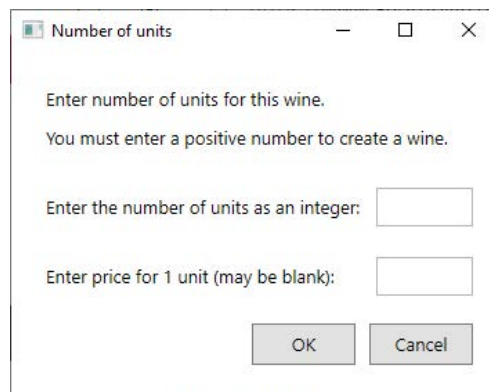
If you select the menu item *Create wine* the program opens the following window:

The window is divided by two *GridSplitter* components. The left column has input fields for the information, that can be entered (the field to enter the number of units is always disabled), while the middle column has combo boxes to select a value from a dimension table. For each combo box is two buttons. The first (the grey) is used to clear the combo box for a selection while the other (the green) is used to open the dialog box to create a new value. If you, for example want to select a country and the country is not created, you can create the country without using the menu item for countries. Below the first two columns are a *TextBox* to enter a description of the wine. The last column is used for the grapes which this wine uses. The column has two *DataGrid* components where the lowest has a filter. The lowest *DataGrid* contains the names for all grapes, while the top one contains the name and the percentage (if given) for this grape in the wine. To add a grape to a wine you double click on the name in the lowest *DataGrid*, or you can click the button *Create grape* if the grape is not already created. In both cases the program opens the following dialog box to enter the percentage for the grape:



You do not have to enter anything as it is not a requirement to state a percentage for the grape. If you have added a grape to a wine, and you regret you can remove the grape by double click the row in the top *DataGrid*.

You cannot create a wine without adding an amount. The field for amount is disabled, and when you click *OK* to create a wine you get the following window:



where you must enter the number of units and maybe also a price. If you click *OK* the wine is created, and at the same time the program create a purchase for the entered number of units and the entered price (if you have entered a price). This operation requires a change to the class *DB* where another version of the method *InsertWine()* has been added, which, in addition to a *Wine* object, also a *Purchase* object as a parameter.

The window *WineWindow* is a relatively complex window and the window's XML is very filling. The same window is used to maintain a wine:

To maintain a wine you can select a wine in the *DataGrid* and click the menu item in the first menu, or you can double click a row in the *DataGrid*. The difference is that the window this time is initialized with values for the selected wine and the four buttons lowest in the window are selected.

If you for example clicks on *Purchase* you get the following dialog box to register a new purchase of this wine:

Date	Number	Price
2020-06-29	24	4375,00

You must enter a value for number, but the value for price may be empty as you do not have to give a price. The *DataGrid* shows the purchases of the wine, and if you double-click on a line you can edit the purchase:

The buttons *Consume* and *Rating* opens corresponding dialog boxes.

## 10.9 THE SEARCH FUNCTIONS

Then there is the four functions in the last menu. The copy of the project from the previous iteration is called *MyWines4*. The four functions are very similar and opens dialog boxes with almost the same design. When you select a function you should enter some search criteria and the function finds objects representing wines, purchases, consumptions or ratings. The first opens a dialog box where the user can enter several criteria to search for wines. If a field is empty (has not been selected a search criterion), it is ignored in the search. If you enter a value in the fields *Name*, *Storing* and *Description* it means that a row in the database to match must contain the value in the actual column. For a criteria where you can enter

two values, it means a from and to value, and if you only enter one value it means either from or to. If you select a value in a combo box a row in the *Wine* table must have that value in the actual column.

The image shows a Windows-style dialog box titled "Advanced search". It contains two columns of search criteria. The left column includes: "Wine name" (a single text box), "Year" (two text boxes separated by a hyphen), "Drink from" (two text boxes separated by a hyphen), "Drink to" (two text boxes separated by a hyphen), "Alcohol %" (two text boxes separated by a hyphen), "Stock" (two text boxes separated by a hyphen), "Storing" (a single text box), and "Text" (a single text box). The right column includes: "Wine types", "Grapes", "Country", "District", "Supplier", "Packing", "Classific.", "Category", and "Producer", each with a dropdown menu and a small circular icon to its right. At the bottom of the dialog are three buttons: "Clear", "Search", and "Cancel".

When you click *Search MainWindow* is updated with the wines that matches the search criteria. To implement the function the above window (*SearchWindow*) must be created, and the ViewModel is called *ViewModelSearch*. It is a very simple class, but inherits a class *ViewModelSearches* which contains everything needed. This base class is also used for three other search functions.

If you for example select *Purchases* in the menu you get the following window to select search criteria for purchases:

**Purchases**

Date:  -       Wine types:

Number:  -       Grapes:

Price:  -       Country:

Wine:       District:

Supplier:

Packing:

Classific.:

Category:

Producer:

When you click *Search* you get a window showing all purchases matching the search criteria:

**Purchases**

Wine name	Year	From	To	Date	Number	Price
Chateau Lamartine	2011	2016	2020	2020-06-29	12	75,00
Benedictus de Vatican	2017	2018	2022	2020-06-29	24	43,75

If you double-click a row in the table with the search result, the program opens the dialog box *WineWindow* for that wine, and you can maintain the wine in the same way as described in the previous iteration.

The two other search functions open corresponding dialog boxes, and I will not show these dialog boxes here.



## 10.10 THE LAST THINGS

Back there are four things:

1. look and feel
2. moving the database to another directory
3. adding a copy function to *WineWindow*
4. a code review and refactoring
5. the class DB
6. test
7. write an installation script

Also this iteration starts to make a copy of the project from the previous iteration, the copy is called *MyWines5*.

### Look and feel

It is about how the program should present itself to the user and thus how the user interface should appear, and after the programming is completed, a number of adjustments and improvements are typically needed.

As the first I have added an icon to the program. Then I have written some styles as static resources in *App.xaml*. It is primary styles for a window and a *DataGrid*. This means that the XML for all window classes must be updated and that is all classes in the folder *Views*. In this process I have tested all windows, a test that has led to several small adjustments.

The *MainWindow* is updated. Here is assigned a command to the menu item to modify a wine and there is assigned commands to all buttons in the tool bar. As another improvement, the *DataGrid* component has been changed so that wines that should have been drunk will be shown in red, but wines that are ready to drink will be displayed in green. Of course, this requires the corresponding information to be present for the wine in question. To implement this feature the model class *WineModel* is extended with a new property. There is also added a *Label* component to the window which show a warning, if there is wines that should have been drunk.

When you create a new wine you must enter a positive value for the number of units. In practice, this is not always appropriate and therefore the program has been changed so that in case of cancellation of the dialog box for entering the number of units (and price) the wine will be created but with an empty inventory. It allows to add wines to the database which currently does not have.

## Moving the database

It should be possible to change where to store the database and then also to create a new database and select another one. It is intended as an exception, but it basically allows you to use the program to maintain several wine cellars.

When the program starts the private constructor in the class *DB* loads the database. For that the constructor should use the following algorithm:

1. Test if the users home directory contains a directory *MyWines*, and if not so create the directory.
2. Test if the directory *MyWines* contains a configuration file called *MyWines.config*. If so open the file and read the file name for the database and else set the database name to *MyWines.sqlite* in the user's home directory.
3. Try to load the database. If it fails (maybe because the database is not created), delete the configuration if it exists and create a new database at the default location and load the database.

To connect to another database the following must happen:

1. Use a file browser to select the database and load the database. If it fails cancel the operation.
2. Create a new configuration file which contains the name of the database.

To create a new database you must:

1. Use a file browser to select where to store the database and enter the name.
2. Create the database.
3. Create a new configuration file which contains the name of the database.

In principle, it is quite simple to implement these functions, and nothing much else needs to be done than expanding the menu with three new menu items and adding three *Command* objects to *ViewModelMain*. The class *DB* is prepared for these extensions, but is expanded with a property to set the value of the file name and a method to test where a filename represents a database for this program. To *MainWindow* is added a label which at the bottom of the window shows the name of the database.

## A copy function

You can enter / selection many values for a wine. Two wines are perceived different if they do not have the same vintage and this means that you have to create the same wine for a different year, but where all data are equal except the data for year. To facilitate the work there is to the window *WineWindow* added a button *Copy* which creates a copy of a wine, and where the user has to enter values for year.

## Code review and refactoring

These means to review all classes (there are 112), and it is then a work that takes time. The task is combined with testing of the program as many classes change. For all classes I have special removed not used *using* statements and variables and code, which are not used. At this time in the project, the classes will also typically include code that is commented out. That kind of code also needs to be removed. Finally, one should be aware of code repetition, where the same code occurs in several places, or there may be methods that are almost the same. For the sake of maintenance, this kind of repetition should be removed, but of course the benefit of removing should be considered. You also need to be aware of classes that are almost identical, and in that case, consider whether such classes should be specializations of a common base class.

Refactoring and code review is very important for the sake of future maintenance, but it is also a process where many errors and inconveniences are typically found, and so has this program. These errors and shortcomings must of course be corrected. There is therefore reason to emphasize once again that refactoring and code review are an extremely important (and time-consuming) iteration in connection with system development, and when you sometimes encounter IT solutions that are put to use and have to acknowledge that the solution contains too many errors, then the reason is often this iteration has not been performed or at least by the development organization has not been done with sufficient enthusiasm. But both time and money are certainly well spent.

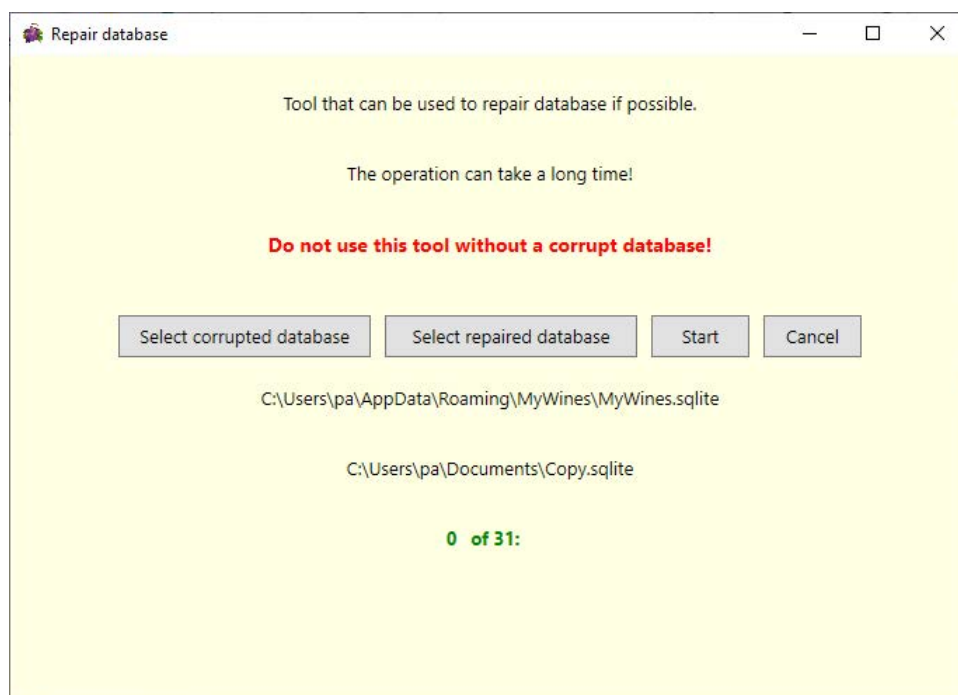
## The class DB

This class is the most important class in this application as object oriented model (ORM) of the database. The database is stored in a file, and if this file is corrupted because the class *DB* does not maintain the database correct, the program will not start. It should not be possible, at least not if the class *DB* does not contain errors, but even if the class should not have errors, something can always happen, and if the program is used for a long time

and you does not have a backup of the database, all information about the wine cellar will be lost. Therefore, there is reason to think about what you can do to ensure that the user does not lose his data.

One possibility is to add a backup function (to the last menu) which makes it easy for the user to back up the database. Such a function is quite simple, but if you implements such a function you should also implement a function to restore the database.

Another possibility is to write a function that can repair a corrupted database. In this case I have added a function which opens the following window:



The function let the user select the database to repair and where to place the repaired copy. It is not simple to write such a function, but in this case the function performs 31 operations:

1. Create a new SQLite database (it is trivial).
2. Create the 15 tables which the database should contains. It is also copy simple as the code can be copied from the class *DB*.
3. Copy the 10 dimension tables which should not cause the major problems. All rows which cannot be copied are ignored. There is one problem as the table *Wine* and the table *Cuve* have a foreign keys for the tables, and when some of the tables has an auto generated surrogate key it is necessary to save pairs of the old key and the new key in a dictionary.

4. Copy the table *Wine* where it also is necessary to save pairs of the old key and the new key in a dictionary. All foreign keys to dimension tables must be assigned the right value if possible. Else they are set to *Null*.
5. Copy the table *Cuve*. Only rows where the foreign keys for the table *Wine* and *Grape* are found (in the dictionaries) are copied.
6. Copy the last three tables: *Purchase*, *Consumption* and *Rating*. Only rows where the foreign key to *Wine* is found are copied. Other rows are ignored.

The result of the function can then be loss of data, and the function is also intended as a tool that can only be used if the database is corrupt and there is no backup.

## Test

After the above the program should be tested before it is ready for use. Although the program in this place should be free of errors, there is still a great risk that something will not work properly. Therefore, a summary test is required, a test that typically must be performed of real users in the right environments and over a period of time.

In this case I have tested the program (as a real user). I have selected the default location create a new database and that directory which is my home directory. I have the performed the following test:

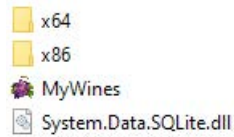
1. Checked the dimension tables *Zipcodes*, *Winetypes*, *Categories* and *Packs* (that is the tables initialized when creating a new database).
2. I have created 2 suppliers, 2 producers, 3 countries, 2 district, 2 classification and 6 grapes.
3. I have created 20 wines, where I have varied which data is entered / selected.
4. Then I have modified 10 of the 20 wines.
5. Create a backup, create a new database and restore the backup.

If I find something which must be changed, the error / inappropriateness is corrected and the effect of the change is tested.

If all goes well, I see the program as tested and ready to use.

### The installation script

To install the program on a computer I use Advanced Installer to create a MSI file. As shown before it is quite simple, but the program uses SQLite and then you must add the needed files to the installer project. First I have build a release version of the project. Then I create a installer project and add the following files:



and the program is ready for install and use on a machine.