Poul Klausen

JAVA 7

About system development

Software Development

bookboon
LEARNING

POUL KLAUSEN

JAVA 7: ABOUT SYSTEM DEVELOPMENT

SOFTWARE DEVELOPMENT

Java 7: About system development: Software Development
1st edition

ISBN 978-87-403-1819-7
Peer review by Ove Thomsen, EA Dania

CONTENTS

FOREWORD

This book is the seventh in a series of books on software development. The programming language is Java, and the language and its syntax and semantic 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 while programming plays only a minor role, and the book does not address anything new with respect to Java. 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 to principles of system development, the book has an appendix that provides a short introduction to UML and an appendix that briefly treats the design of databases. The book assumes knowledge of programming and Java and the development of applications that uses databases corresponding to the content of the first books in this series.

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 Java. 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 and 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 to the many examples and also solves the exercises and problems or possibly just studying the recommended solutions.

All books ends with one or two larger sample programs, which focus primarily is on 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.

Most books also ends with an appendix dealing with a subject that would not be treated in the books. It may be issues on the installation of software or other topics in computer technology, which are not about software development, but where it is necessary to have an introductory knowledge. If the reader already is familiar with the subject, the current appendix can be skipped.

The programming language is, as mentioned Java, and besides the books use the following products:

- NetBeans as IDE for application development
- MySQL to the extent there is a need for a database server (from the book Java 6 onwards)
- GlassFish as a web server and application server (from the book Java 11 onwards)

It is products that are free of charge and free to install, and there is even talk about products, where the installation is progressing all by itself and without major efforts and challenges. In addition, there are on the web detailed installation instructions for all the three products. The products are available on Windows and Linux, and it therefore plays no special role if you use Linux or Windows.

All sample programs are developed and tested on machines running Linux. In fact, it plays no major role, as both Java and other products work in exactly the same way whether the platform is one or the other. Some places will be in the books where you could see that the platform is Linux, and this applies primarily commands that concerning the file system. Otherwise it has no meaning to the reader that the programs are developed on a Linux machine, and they can immediately also run under Windows unless a program refers to the file system where it may be necessary to change the name of a file.

Finally a little about what the books are not. It is not “how to write” or for that matter reference manuals in Java, 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 task is presented, to the program is written, tested and put into operation at the future users. In practice, a system development is implemented as a project that is characterized by a longer period, and the task is typically solved by a project group that do all the work. Both the time schedule and the size of the project group is obviously determined by the task's scope so that the time horizont 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, but to talk about a system development project, it will often be a project that will be performed over several weeks and perhaps even months, and which involves several people.

1Large 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 along with it there are creating new wishes and requirements for the finished system. It is thus impossible from start 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 as 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 that 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 must usually be used for many years. In that time, the program must often be modified as new demands occur, or there is something you want to work 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 play program quality 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 made, and finally may be mentioned that programs should be robust, so they do not fail because of improper use, and also the robustness is part of the program's quality.

All that should emphasize that 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 develpment 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 an example 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 as to give some guidelines to suit the tasks you now 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 assets 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.

In the rest of this book, I will describe the method's activities and what you should do in each of the six 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. However, such development tools also have their entitlements, and in the books Appendix A, I have given a brief description of some tools.

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 folowing 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 writte 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 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 detail, 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 has 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 a 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 subdirectory 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

2.2.1 AMBIGUITIES

The task formulation describes the task overall, but not the details. This will 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 get worked up 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, as 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.

2.2.2 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 resolved, and if one at all is able to solve the problem. Contains the project such challenges, it is important that they come to light as soon as possible. At worst, it may well mean that the project can not 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 can not be priced and may not be time estimated.

Now there may also be less challenges 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 subtasks 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.

2.2.3 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, the 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 that it is something you have to pay for, and maybe paid a license for a particular service.

Now do the opposite party not being an external organization, but it can, for example be customer's business. Then there is is 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 vice versa.

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.

2.2.4 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 has also 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 forward 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 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 it is easy for the developers to develop good prototypes. The work is also not wasted, as it often includes something that still need to be developed at a later date.

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

2.2.5 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 carrie out the analysis, and in this way can the task based on a detailed specification of the requirements 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, get 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.

2.2.6 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 documantation should be used as basis for the rest of the system development project.
- Secondly, the documentation are 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 oven 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 for now, but until then, 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 subfolders 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 subfolder of meeting summaries, a subfolder to other material, etc. The project folder will also contain NetBeans 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 designed with a NetBeans project, a project which I will continue to work on in the programming phase, but so that I will continue to 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 NetBeans 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, including to manage different versions of documents and also 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 work 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's 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.

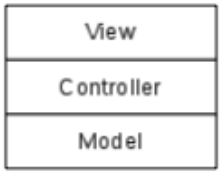
In contrast to the analysis the design activities varies highly dependent on what the task is, and it is difficult in the same way to establish guidelines for the implementation of the design phase, but the following are typical activities:

- 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.

2.3.1 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 final example I will show how the pattern can be applied in practice.

2.3.2 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 defines 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 is 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's a simple process, and it obviously can not 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 – the book's appendix give a short presentation), but I'll 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) and that 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). The diagrams used correctly can provide an overview, and so they do not require knowledge of specific programming languages.

2.3.3 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 is it the 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. 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 are databases 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.

2.3.4 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 should 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, but modern programming languages is far more flexible, and by combining language and plain text in the form of comments, a programming language like Java 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 translated.

2.3.5 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 NetBeans you easily can create windows components, and it is similar easy to add application logic so the user can navigate between 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.

2.3.6 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 NetBeans project and often both. The NetBeans project that contains the project (possibly several projects), I will continue working with during the programming phase.

2.3.7 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, wee 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 can not just change it, without it matters the other classes.

Now it is not such that you can avoid couplings. A program obviously can not 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 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 programs 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 can not 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 NetBeans 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's 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's 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 quality of the code you writes is alright.

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 with a special tool attached to NetBeans. 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 inexpediencies and errors.

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

2.4.1 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 can not 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 requirement specification. 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 todo-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 todo list, and then the program is ready for use.

The test phase is a follow-up to the tests, 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 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 can not 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 phase will really take the program in use and thus hand it to 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 to.
- 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, and I will in chapter 5 with an example shows how the method can be used to develop a small program. 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 proceding with other parts towards the finished program. This means that you work with analysis, design, programming and test 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 phases carried out in a particular order. There is a need for more flexible methods that can be adapted to the given task and situation – something that I will look at later in the other books.

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.

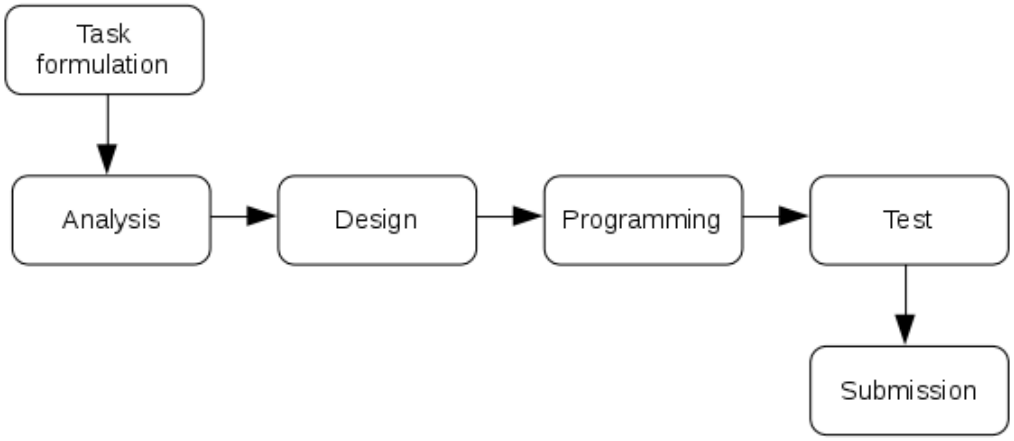
The conclusion is that the waterfall model is very applicable for the development of small programs which have not many uncertainties and the model is used in the next books, but with the important addition that it is adapted by my way to the development of a program.

As mentioned, I later 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 delepment 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 it 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 alternative procedures 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:



4 MVC

Above at the design I have talked about *Model View Controller* as a pattern for the architecture of a GUI program. It is a three-layer architecture, and in the following I will briefly elaborate the architecture, as I will use it in the following example.

Model View Controller – short MVC – is one of the most talked about design patterns as a pattern that recommends how to design a program with a graphical user interface. MVC is a very interesting pattern, because the pattern is concerned with a very frequently occurring problem, and it is a pattern that is available in several forms, in particular because different types of applications require variations of the pattern, but also because the development tools often introduce their own variations.

There are three elements in MVC:

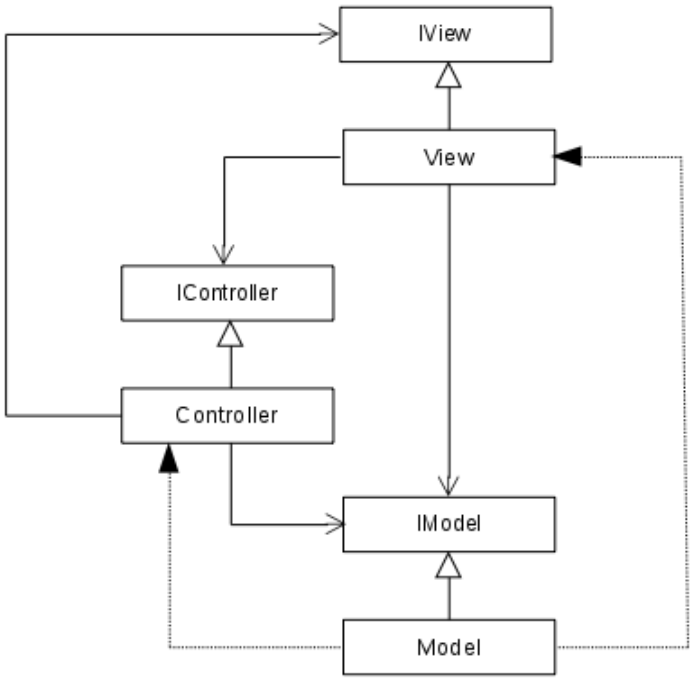
1. *Model* representing the program's data and thus the program's state, and it is also the model, which implements the essential parts of the program logic. What the model do is hidden from both the controller and view, but the model defines (often via an interface) methods such that the controller and the view can read and modify the model's state, and the model can send notfications to observers concerning changes in the state. Typically, the view layer may be observer, but also the controller layer may be observer for the model.
2. *View* representing the model and displays the model's state by means of components in a graphical user interface. The program's view will often use many different components (for example panels, buttons, input fields, etc.), and it's the view layer's responsibility to ensure that these components reflects the model's state.
3. The *controller* processes the user's input and determine what impact it should have for either program's view or model.

In my use of MVC the main program will typically create the view object, which then has the responsibility to establish the controller and the model and possible send the model as a reference to the controller (possibly also a reference to the view itself). The result is that the controller knows the model (and possibly also the wiew, but, when appropriate, typically through an interface), while the view knows both the controller and the model. To get loose couplings between the elements both the view, controller and model are often defined by an interface and MVC can be outlined as shown below.

In the figure, are both the model, view and controller defined by an interface. It is not certain that it will be the case in practice, but the objective is that in this way you can better replace a layer (for example the program's view) without affect the other layers. The dotted lines illustrates that the model can send notifications to the other layers. Listeners are typically the program's view, but it can also be the controller.

The idea is the following:

- When there are an occurrence of an event in the application's view, because the user is doing something, the view will delegate the treatment to the controller (call a method on the controller), which then decides what to do.
- The view can always read the model's state using the methods defined by an interface that the model implements.
- The controller knows the view so it can be notified to update the user interface.
- The controller also knows the model so that the controller can update the model.
- The model knows neither the controller or the view, but it implements the observer pattern such the view (and possibly also the controller) may be registered as observers and receive notifications when the model's state is changed.



The above is a common sight on MVC, which seeks decoupling of the elements using interfaces. Another question is, in turn, where and how the individual parts/components are created. That is, how the individual objects are instantiated and that is to a large extent determined by the current development tools. It will be illustrated with the following example of a GUI program.

Another question is what the individual layers concretely should contains in the form of classes and other types. In general, the view layer may contain a class for each window, and there will, in principle, also be a corresponding controller. Moreover, in all three layers there will also be other classes and defining interfaces. There may also be several layers, and often there will be a layer below the model layer, which has the task to map the object-oriented model to for example a database. This layer is often called a *data access layer*. For large applications, it may also be a possibility to create an overview to subdivide the individual layers, so they consist of several packages.

MVC exists as mentioned in several variants and is a highly used design pattern, because it is sufficiently flexible. This may mean that some of the types and maybe even layers are not found in a concrete solution. There is nothing wrong with that, and you should not, of course complicate a solution needless because a pattern prescribes it.

5 LIBRARY

As a final example, I will write a program that can maintain a simple library of books, for example on an educational institution. In addition to being able to manage the library's content the program must also be able to manage borrows for example to students and staff. It is, therefore, a typical database application. The aim is to test the waterfall method and the MVC pattern with an example that is a typical office application with a graphical user interface.

5.1 TASK FORMULATION

The task is to develop a program for managing a library of books at an educational institution, when books can be borrowed to the institution's staff and students. The program should partly be able to manage the library's holdings of books, including the registration of new books, and also manage borrowed books and especially recall. Since the books are typically textbooks used for teaching/research, and where you often only need small lookups, the application must offer search facilities, where you can also see who, where appropriate, have borrowed a specific book.

The maintenance of the library's contents is performed by an employee who has responsibility for registering new books and delete discontinued titles. Moreover, it is also the librarian's responsible to send reminders to borrowers who have not returned borrowed books.

All on the institution can, in principle, borrow a book, and book borrowing is a self-service, where a person when borrows books may register himself as a borrower of the borrowed books. The library operation is thus based largely on trust.

5.1.1 REMARK TO THE TASK FORMULATION

The task formulation is nothing but a short description of the task. There are many details to be clarified, but it is part of the analysis.

The results of the phase is alone this document, and a project folder named *library*. Preliminary the folder contains only a subdirectory *doc* for this document.

5.2 ANALYSIS

The program is an ordinary PC application running on a machine located in the library such that borrowers can registering information about a borrow and return books. All who are created as a user, can borrow books, and the principle is that if a book leaves the library, it must be registered as borrowed and to whom, and when the book later is brought back to the library, the borrower must record the book as returned.

The analysis includes:

1. The system's data
2. User access
3. The programs functions

5.2.1 THE SYSTEM'S DATA

An institution may in principle have many titles, where a title is an ordinary paper book. Basically, the library contains books with academic content, divided into categories, but the library can in principle also contains non technical books. It is understood that a book should be recorded with the following information:

- ISBN
- Title
- Edition
- Publisher year
- Category
- Publisher
- Authors

- Number of pages
- Number of copies
- A description

ISBN is not used as key as the library includes books that do not have an ISBN. Since there are two formats of the ISBN, one should be able to register both numbers. The rules for ISBN are indicated by:

http://www.isbn.dk/

Both the categories and the publishers are just recorded as a name, and there must not be two categories or publishers with the same name.

Authors are registered with a first name and a last name. It provides a problem as there may well be several authors with the same name. It has therefore been decided to register an author with the following information:

- First name
- Last name
- A description

where the description can be used to register additional information – for example if two authors have the same name.

To borrow books, a person must be registered as a borrower/user. A user is registered with the following information:

- Name
- Address
- Zip code
- Phone number
- Position (for example student)
- Email address

The email address is perceived as key.

When a user borrows a book, the book is recorded together with who the borrower is and the date.

5.2.2 USER ACCESS

For a user to borrow a book, the user must log in by entering the email address and a password. If you as a user comes to the library machine, you can immediately see a list of the library's books, but otherwise you should be able to do anything else than log in. After log in, there must be three levels for the use of the program:

1. Ordinary users (students), which alone can search the books and read all the information about a particular book and including who might have borrowed the book. An ordinary user can also borrow the book (if not all copies of the book are borrowed) and returns the book back.
2. Librarians (stafs) as beyond what an ordinary user can, in addition, can maintain the library. That is create books, publishers, categories, authors and modify these data. Librarians should not be able to delete books (publishers, categories and authors).
3. Administrators that can perform all functions that in addition to the above primarily includes maintenance of users, including sending emails to users regarding recall of borrowed books.

There may well be several administrators, but when running the program the first time the program must automatically create an administrator.

5.2.3 THE PROGRAM'S FUNCTIONS

At programstart: An overview of all the books in the library with the possibility of filtering.

Ordinary users:

- search the library
- see details
- borrow books
- return borrowed books
- see information about the other borrowers of a book
- change password

Librarians:

- create and maintain books
- create and maintain publishers
- create and maintain categories
- create and maintain authors

Administrators:

- delete books, authors, categories, publishers
- create and maintain users
- sending mails regarding recall of books

The result of the above analysis appears from the following requirements specification.

5.2.4 REQUIREMENTS SPECIFICATION

When the program opens, it should display a list of all the library's books, where for each book appears

- title
- publisher year
- publisher
- category

If you are not logged in, you can not do anything other than browsing in the book list.

Functions

The program has the following features, which are organized into four menus:

General

- Log in

Users

- Search
- Change password
- Log out
- View details (opened by double clicking on a book)

Librarians

- Maintenance of publishers (create, modify)
- Maintenance of categoris (create, modify)
- Maintenance of authors (create, modify)
- Create books
- Maintenance a book (opened by double clicking on a book)

Adminitrators

- Delete publishers
- Delete categories
- Delete authors
- Delete books
- Recall of books
- Maintenance of users (create, modify, delete)

The most important feature is *Maintenance a book*, which opens a window that shows all the details about a book. The window is also used to

- Borrow the book (if not all copies are borrowed)
- Return a borrowed book
- See who has borrowed the book, if it is borrowed

Depending on the user role, you can also from this window

- Change all information about the book
- Delete the book

When searching you must be able to search for

- ISBN13, matches the start of the string
- ISBN10, matches the start of the string
- Title, matches the content of the string
- A particular edition
- A particular publisher year
- A particular publisher
- A particular category
- Author, two strings separated by spaces where the last part matches the start of the last name and the first part matches the content of the first name
- Text, matches the content of the description

Maintenance of users must display a list of all users, and you should be able to creates new users and modify information about a particular user. In this context, it must be possible to see what books the user has borrowed.

At recall of books you must be able to select a time interval (between two dates), and you should then get a list of all the books that are borrowed in that period. You have to select the books to be recalled, and then it should be possible to send an email to the borrower.

Users

There are three user roles corresponding to the above three categories. After login, the features available are determined by the user role:

1. ordinary users
2. librarians
3. administrators

Datadictionay

Name	Data type	Description
Book		
Isbn13	String on 17 characters	Not required
Isbn10	String on 13 characters	Not required
Title	String on max 255 characters	
Edition	Integer	Not required, 1–9 both inclusive
Year	Integer	Not required, 4 digets
Pages	Integer	Not required, positive integer
Copies	Integer	Not required, positive integer
Publisher	String on max 50 characters	
Cattegory	String on max 50 characters	Not required
Text	Any text	Not required

Name	Data type	Description
Author		
Firstname	String on max 50 characters	Not required
Lastname	String on max 30 characters	
Text	String on max 200 characters	Not required
User		
Email	String on max 100 characters	
Password	String on max 30 characters	
Firstname	String on max 50 characters	
Lastname	String on max 30 characters	
Address	String on max 50 characters	
Zipcode	String on 4 characters	
Telefon	Streng på max. 20 tegn	Not required
Title	String on max 50 characters	Not required
Role	Integer	1 = admin, 2 = staf or 3 = student

Deployment

The program uses a database and there must be written a script that can create the database.

There should be written script, which can install the program on the users machine.

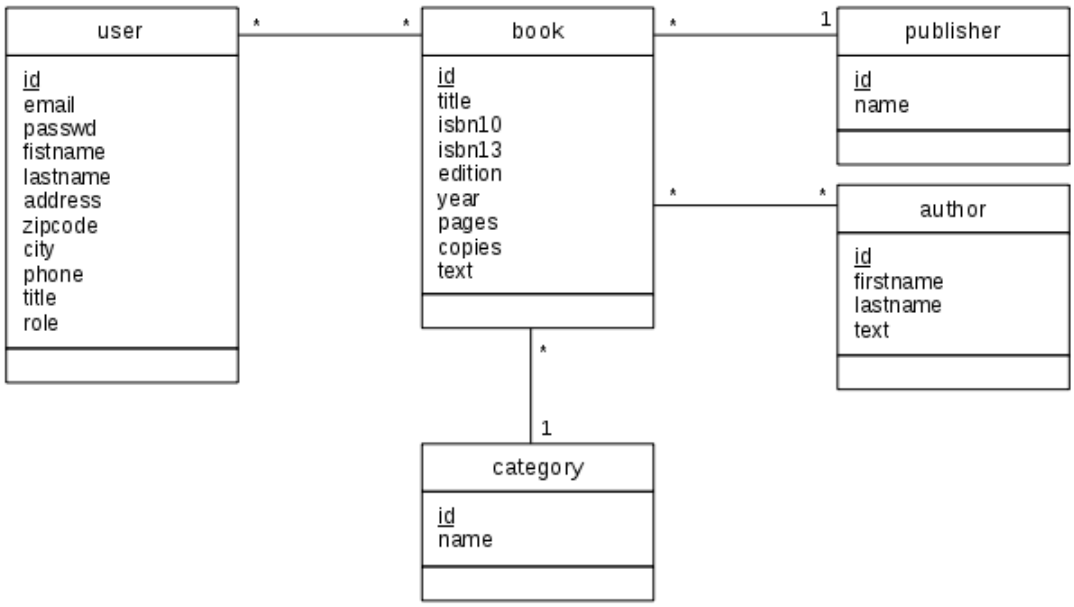
5.3 DESIGN

The program has low complexity, and basically the program must maintain a database and execute queries on the database. The design is performed in the following steps:

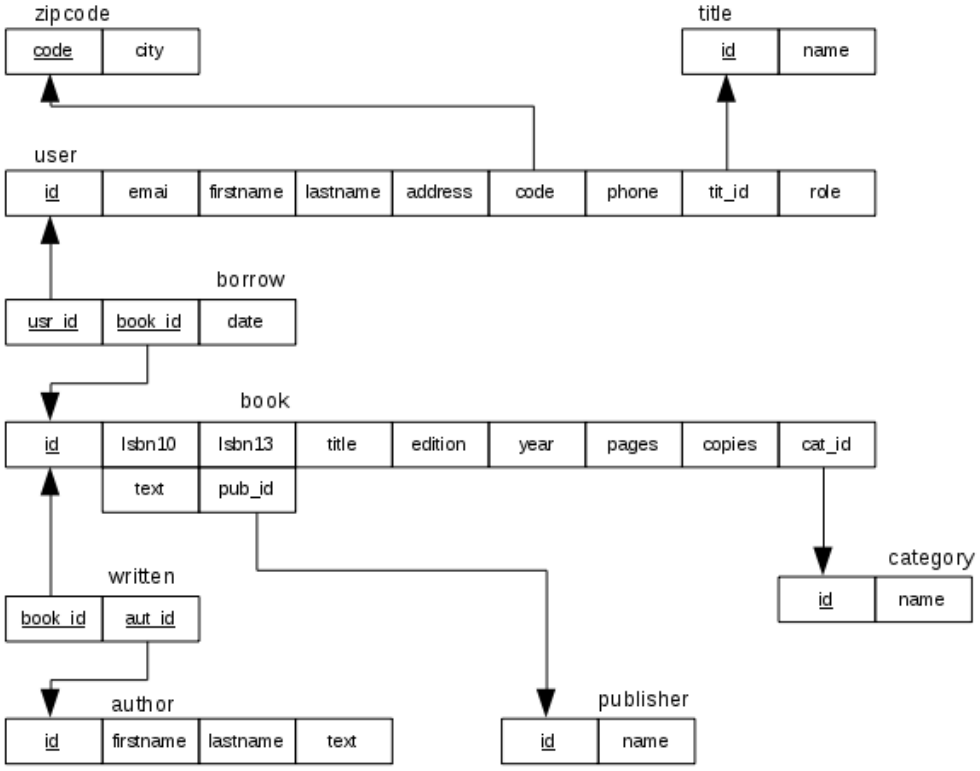
1. database design
2. layout of the user interface
3. design of the architecture
4. design of the model layer
5. design of the dal layer
6. design of the controller layer

5.3.1 DATABASE DESIGN

The database can be illustrated by means of the following class diagram in which the *book* and the *user* are the two main entities. For all entities are chosen a surrogate key. Regarding *category*, *titles* and *author* there are no other obvious keys, and even if a publisher has an official publisher identifier in combination with the country code, it is not suitable as a key, as many publishers consists of several publishers (or series) with the same name. For *book* is selected a surrogate key because not all books has an ISBN. Finally, there is the user, wherein the email address could be used as a key, but since it is a relatively long string, there is also selected a surrogate key.



For the relationship between *user* and *book* is associated an attribute called *date* that indicates the date a book is borrowed. From the general mapping rules to a relational model you get the follwing diagram, where there are defined an entity to a user's title:



The model is fully normalized to third normal form. The script that creates the database is called *Library.sql* and are prepared in accordance with the above design and data dictionary from the analysis. Some names have been changed (primarily foreign keys). The script is shown below as the specific data definitions is important in the programming phase:

```
use sys;

drop database if exists library;
create database library;

use library;

create table title (
 id int auto_increment not null,
 name varchar(50) not null,
 primary key (id)
);

create table publisher (
 id int auto_increment not null,
 name varchar(50) not null,
 primary key (id)
);

create table category (
 id int auto_increment not null,
 name varchar(50) not null,
 primary key (id)
);

create table author (
 id int auto_increment not null,
 firstname varchar(50),
 lastname varchar(30) not null,
 text varchar(200),
 primary key (id)
);

create table book (
 id int auto_increment not null,
 isbn13 char(17),
 isbn10 char(13),
 title varchar(255) not null,
 edition int,
 year int,
```

```
 pages int,
 copies int,
 catid int,
 pubid int,
 text text,
 foreign key (catid) references category (id),
 foreign key (pubid) references publisher(id),
 primary key (id)
);

create table written (
 bookid int not null,
 autid int not null,
 foreign key (bookid) references book (id) on delete cascade,
 foreign key (autid) references author (id) on delete cascade,
 primary key (bookid, autid)
);

create table zipcode (
 code char(4) not null,
 city varchar(30) not null,
 primary key (code)
);

create table user (
 id int auto_increment not null,
 email varchar(100) not null,
 passwd varchar(150) not null,
 firstname varchar(50) not null,
 lastname varchar(30) not null,
 address varchar(50) not null,
 code char(4) not null,
 phone varchar(20),
 titid int not null,
 role int default 3,
 foreign key (code) references zipcode (code),
 foreign key (titid) references title (id),
 primary key (id)
);

create table borrow (
 bookid int not null,
 userid int not null,
 date date not null,
 foreign key (bookid) references book (id) on delete cascade,
 foreign key (userid) references user (id) on delete cascade,
 primary key (bookid, userid)
);
```

```
load data infile '/var/lib/mysql-files/data/zipcodes' into table zipcode

character set UTF8 fields terminated by ';' lines terminated by '\n' (code,
city);
```

The table *zipcode* must be initialized with information about Danish zip codes, which is done in the last command by reading data from a text file. The file is called *zipcodes*. For safety reasons, MySQL generally do not accept the command

LOAD DATA INFILE

unless the file is placed in a sub directory with read access for all in the directory

```
/var/lib/mysql-files
```

After this the script can be executed and the database is created.

There is also written a script called *Books.sql* that creates three books in the database. This script is intended solely as part of the development, such that the database has a content, and the script is executed after the database is created.

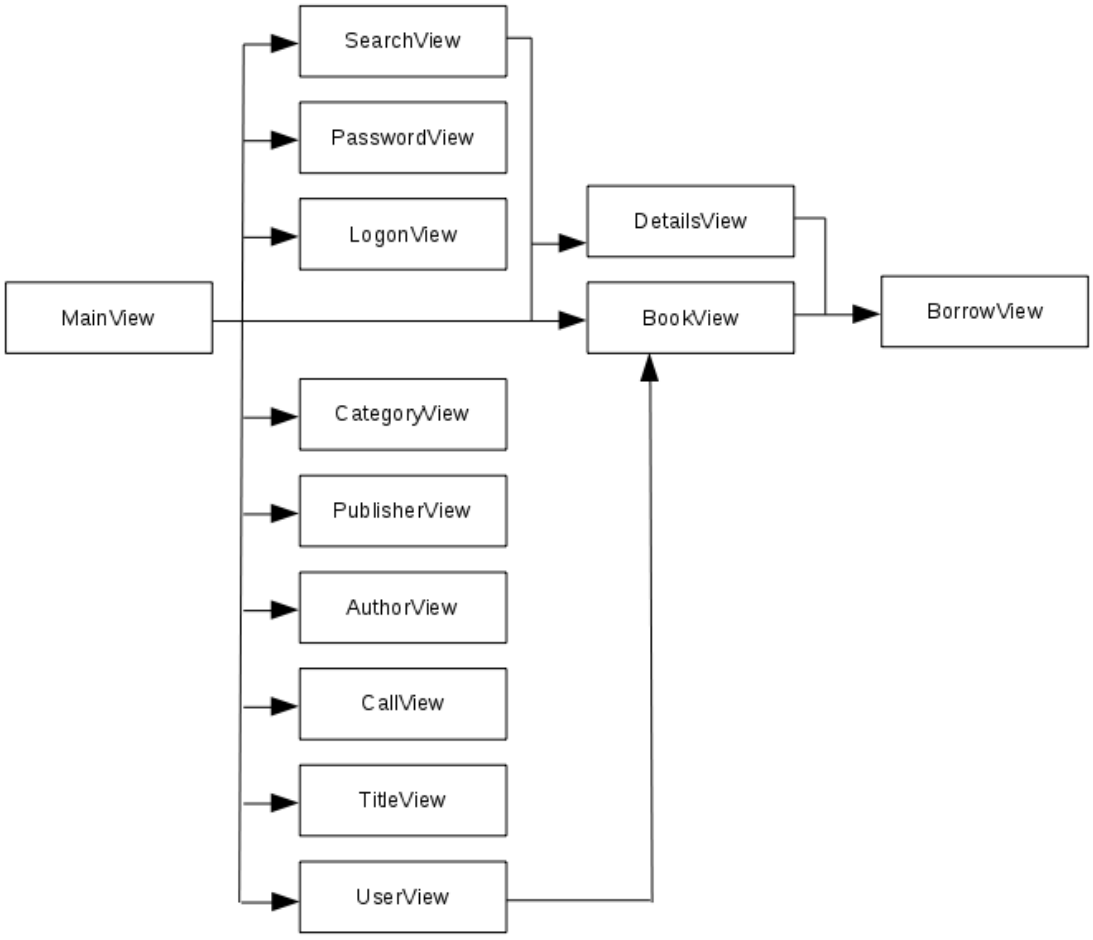
The table *user* has a column called *passwd*, which are intended for the user's password. It has the type VARCHAR and the values are therefore saved as text, but in encrypted form. The password is a calculated message digest, which in principle is an integer, that is stored in the database. In such the type should be BINARY, but I want to convert the number to a hexadecimal string, and then the type must be VARCHAR.

5.3.2 LAYOUT

The program should have a very classic layout where it opens a main window with a list of all books in the library. In addition, the program consists of a menu and several dialog boxes that either are opened from the menu, by clicking of a button or by double-clicking on an item in a *JTable*. The program's main windows are:

- *MainView*, which is the main window with the menu and shows an overview of all books.
- *LogonView*, there is a login dialog box (for entering the user name and password).
- *DetailsView*, that is a dialog box to display the details about a book as well as show who have borrowed the book.
- *BookView*, there is a dialog box to edit a book. The dialog box should also be used both to create and edit books and can only be accessed if you have staf privileges. It is the program's most complex dialog.
- *BorrowView* is a simple dialog box that shows which users have borrowed a particular book.
- *TitleView*, there is a dialog box for maintenance of titles.
- *CategoryView*, which is a dialog box for maintenance of categories.
- *PublisherView*, which is a dialog box for maintenance of publishers.
- *AuthorView*, which is a dialog box for maintenance of authors.
- *PasswordView* to change the user's password.
- *SearchView*, that is a dialog box for advanced search.
- *UserView*, there is a dialog box for maintenance of users. It is also a complex dialog box.
- *RecallView*, which is a dialog box for recall of books. It is an average complex dialog.

The relationship between the windows can be illustrated as follows:



As regards the advanced search, you can search for the following criteria:

1. ISBN-13
2. ISBN-10
3. Title
4. Edition
5. Publisher year
6. Publisher
7. Category
8. Author
9. Description

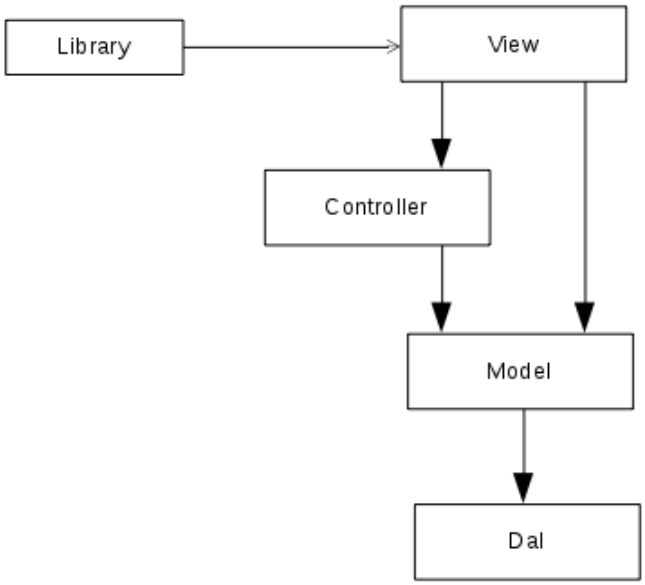
- When searching the ISBN, the number needs to start with the search text.
- When searching the title and description, the text must contain the search text.
- When searching the author, you enter any string, and if there is a space, the last space is separating the first and last name. The first name must contain the search text while last name must start with the search text.
- For the other fields the search value must be equals to the fields value.

With regard to the dialog box *RecallView* it should for a selected period show a list of all the books that are borrowed. Here the administrator can select which books to send an recall notice, and what borrows that has to be removed. The latter corresponds to books that are lost, and the number of copies has to be reduced by 1. If the administrator checked a book for a recall notice the program must sent an email to the borrower.

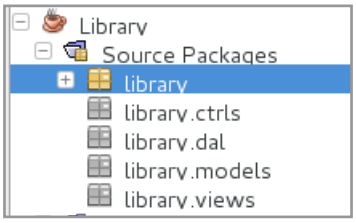
As the above dialog boxes are all relatively simple, there are not developed prototypes.

5.3.3 ARCHITECTURE

The program is written starting from MVC, and uses basically a 4-tier architecture:



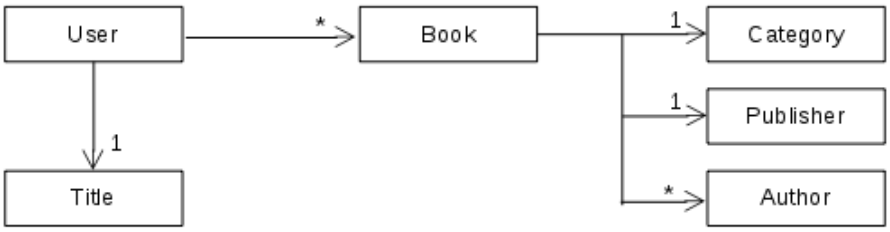
The *main()* method creates *MainView*, which then directly or indirectly instantiates other program objects. A window creates essentially a controller, but for very simple windows where there is no particular data processing the windows directly calls methods in the model layer. A controller must perform an essential task to be included. As a start, I have created a NetBeans project called *Library*, where preliminary are defined the following packages



The model layer (see below), defines an object oriented abstraction over the database. The database operations are not implemented in the model layer, but are implemented in a layer under the model layer (Data Access Layer), which only the model layer knows. The aim is to keep everything regarding the database hidden from the rest of the program and to create an overview, and in order to facilitate maintenance. The disadvantage is that as the model layer must include very thin classes that act as adapters for the dal classes.

5.3.4 DESIGN OF THE MODEL LAYER

The model layer consists primarily of classes that represent the objects, the program must manipulate. The classes can be illustrated in the following diagram:



The classes are all simple and consists primarily of instance variables and get-and set methods. The implementation of the classes methods is deferred to the programming phase. As an example is shown the class *Book*:

```
package library.models;

import java.util.*;

// Represents a book
public class Book
{
 public static final int ISBN13 = 17;
 public static final int ISBN10 = 13;
 public static final int TITLE = 255;
 private int id; // the book's id
 private String isbn13;                          // the book's ISBN-13
 private String isbn10;                          // the book's ISBN-10
 private String title;                           // the book's title
 private Integer edition;                        // the book's edition
 private Integer year;                           // the book's publisher year
 private Integer pages;                          // the book's number og pages
 private Integer copies;                         // number of copies og this book
 private String text;                            // a description of the book
 private Category category;                      // the book's category
 private Publisher publisher;                    // the book's publisher
 private List<Author> authors = new ArrayList(); // the book's authors
}
```

The class *User* has af reference to the class *Book*, where there is a reference for the books that the user has borrowed. When a user borrows a book, the reference should indicate which book and when the book is borrowed, and for that is added the following class to the model layer:

```
package library.models;

import java.util.*;

// Represents a borrow of a book
public class Borrow
{
 private Calendar date; // the date for the borrow
 private Book book; // the book that has been borrowed
}
```

The result is, that there are added 7 model classes to the model layer.

The model layer delegates all database operations to the DAL layer, such that everything regarding the database and SQL are located there. Therefore, the model layer defines an adapter class for each of the 6 model classes, and an example is:

```
package library.models;

import java.util.*;

// Adapter class for User
public class Users
{
 /**
  * Returns all users.
  * @return All users
  */
 public List<User> getAll()
 {
  throw new UnsupportedOperationException();
 }

 /**
  * Returns a user with a particular email address and password.
  * @param email The user's email address
  * @param passwd Message digest of the user's password
  * @return A User object if the user is found or else null
  */
 public static User getUser(String email, String passwd)
 {
  throw new UnsupportedOperationException();
 }

 /**
  * Returns a user with a particular key.
  * @param id The user's primary key
  * @return A User object, if the user is found and else null
  */
 public User getUser(int id)
 {
  throw new UnsupportedOperationException();
 }

 /**
  * Creates a new user.
  * It is assumed that the user object previously is validated.
  * @param user The user to be added to the database
  * @param passwd Message digest of the user's password
  * @return true, if user is added
  */
 public boolean insert(User user, String passwd)
 {
  throw new UnsupportedOperationException();
 }
```

```
/**
 * Updates a user. It is assumed that the user object previously is validated.
 * @param user The user that should be updated
 * @return true, if user is updated
 */
public boolean update(User user)
{
 throw new UnsupportedOperationException();
}

/**
 * Method that is used by a user to change the password
 * (the user's own password).
 * @param user The user
 * @param passwd Message digest of the new password
 * @return true, if the user's password is changed
 */
public boolean update(User user, String passwd)
{
 throw new UnsupportedOperationException();
}

/**
 * Delete an user with a particular key.
 * @param id The user's primary key
 * @return true, if the user is deleted
 */
public boolean delete(int id)
{
 throw new UnsupportedOperationException();
}

/**
 * Tests if there is an user with a particular email.
 * @param email The mail address to search for
 * @return true, if the user is found
 */
public boolean exists(String email)
{
 throw new UnsupportedOperationException();
}

/**
 * Method used by an adminstrator to change an user's password.
 * @param email The user's email address
 * @param passwd Message digest of the new password
```

```
   * @return true, if the user's password is changed
   */
  public boolean changePassword(String email, String passwd)
  {
   throw new UnsupportedOperationException();
  }
}
```

As seen in this class, the encryption of the users password happens in the model layer. The principle is as follows. When creating a new user, you choose a password, which of course is just a string. For this password is calculated a message digest, which is just a bit pattern calculated using a hash algorithm. This bit patttern is converted to a hexadecimal string and is stored in the database along with the other user data. When you again want to read the user's data (for example at logon), you must re-entering the password, calculate a new message digest and check if it matches the one stored in the database. The principle is to use a calculation algorithm that is strong enough so that no one is capable on the basis of the encrypted value in the database to determine or guess the user's password. Java has built-in classes for that sort of calculation which is considered to be sufficiently strong.

5.3.5 DESIGN OF THE DAL LAYER

The DAL layer must define classes to database operations. There must be a class for each adapter in the model layer, and basically the classes will consist of static methods. Since the program is database heavy, a significant portion of the code would be placed in this layer.

A problem to be solved is where to store the username and password to the database server, when these values can not be hard coded. The values should be stored in a configuration file that administrators can configured.

5.3.6 DESIGN OF THE CONTROLLER LAYER

In general, there will be a controller class for each dialog box. Although controller classes in principle is simple, they will contain some code as they need to validate the user input. It is typical controller classes that sends data to the model, if they can validate the data correctly.

It is also the controller layer that sends mail to users when an administrator has to recall books. Java has the necessary classes to send mails.

5.3.7 REMARK TO THE DESIGN

The result of the design is

1. the above report
2. a CSV file containing Danish zip codes
3. a SQL script *Library.sql* that creates the database
4. a SQL script *Books.sql* that creates three books in the database
5. a NetBeans project with the result of the design

5.4 PROGRAMMING

The phase requires that the database is created, and the script that creates the three books is performed.

I will divide the programming in four steps (iterations):

1. The book list
2. Maintenance of books
3. User administration
4. Borrowing and search

The first iteration is relating to the *MainView* which should shows a list of all books in the database. The iteration also defines the overall look and feel.

The second iteration includes the maintenance of books, authors, publishers and categories, that is create, modify and delete.

The third iteration includes the maintenance of users, including logon and implementation of the users' access rights to the program.

Finally, the last iteration implements the users borrow of books and return of books and including the administrator's recall of books with the issue to send mails with recalls to borrowers.

In the following I will, for each iteration briefly explain what is done. The result of each iteration is documented with a NetBeans project.

5.4.1 THE BOOK LIST

As the first is created a copy named *Library1* of the NetBeans project from the design.

Then I have written the model classes *Publisher*, *Category*, *Author* and *Book* finished. They are all simple classes, and the work is mainly to implement the *get-* and *set* methods and override *equals(),* such that comparing of objects have value semantics.

The next step is to write four classes in the dal layer:

1. *BookData*, that should have a static method, that returns alle books in the database
2. *PublisherData*, that should have a static method, that returns a publisher with a particular key
3. *CategoryData*, that should have a static method, that returns a category with a particular key
4. *AuthorData*, that should have static a method, that returns all authors for a particular book

All classes are simple and will be expanded continuously through the project's iterations. Here are the last three methods defined with friendly visibility, because they should not be used outside their package. There is also defined a simple class *Db* that only has one simple metode, that returns a connection to the database. All parameters to the database are hard coded within this class, something to be changed later. The class will also later be expanded with other methods.

After implementing the above classes I have implemented the method *getBooks()* in the model class *Books*.

Then the model can provide the necessary data to the book list and I can address to write the program's *MainView*.

To separate classes in the view layer I have created a sub package:

- *library.views.tables* to helper classes for *JTable* components

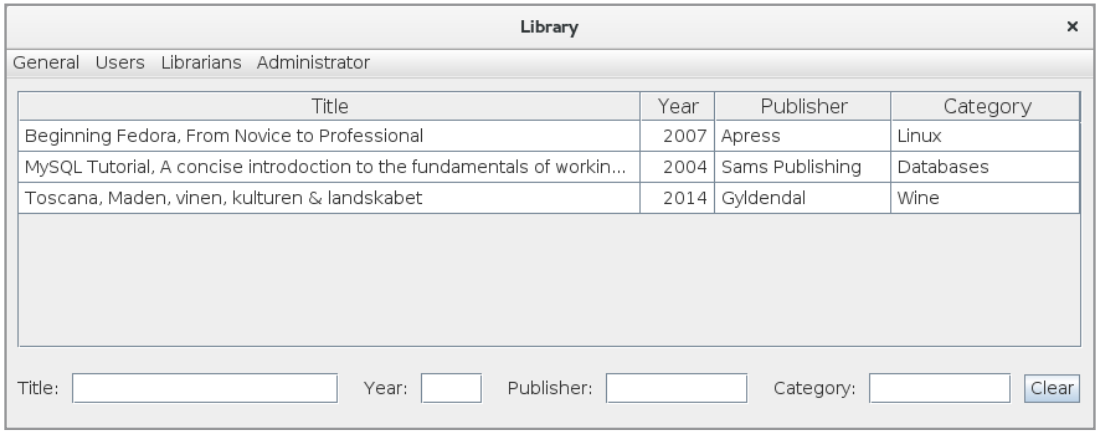
The *MainView* must contains a *JTable*, and I have added the folowing classes to the above package:

- *IntegerRenderer* that is a *CellRenderer* that assign a font and defines that numbers should be right aligned. It also defines that 0 must be shown as blank.
- *StringRenderer* that is a *CellRenderer* that assign a font and a margin.
- *FilterListerner* that is a simple interface that defines a single method used to define a row filter for a *JTable*
- *Filter* that is a simple class that implements *DocumentListener*, and is used to simplify the definition of a row filter for a *JTable*
- *BooksTable* that is a model for the *JTable*

To the package *library.views* are added the following classes:

- *Options*, that is a simple class that defines the fonts, that the program should use
- *GUI*, that is a class with statical help methods to build the user interface
- *MainView*, that is the program's main view

After these classes the *main()* metod can open the *MainView*, and the result is:



The menu is created with the necessary menu items, but none of the menu items has a function.

You will notice that instead of the class *GUI* I could have used class library *PaLib*, and in fact there is no particular reason that I have not done it.

After this the first iteration is completed and the program can be shown to the future users as a verification that the task is understood.

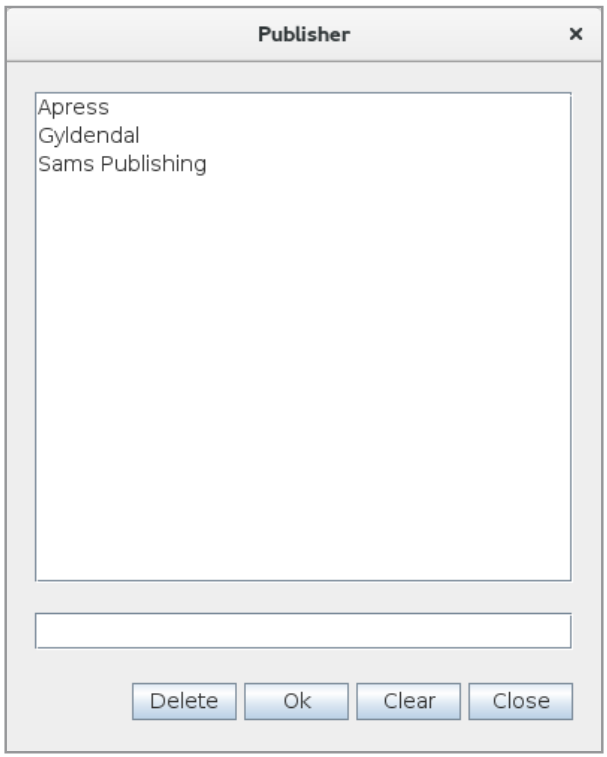
5.4.2 MAINTENANCE OF BOOKS

As the first is created a copy named *Library2* of the NetBeans project from the previous iteration.

In this iteration I should implement the functions under the menu *Librarians*, and the function to edit a book when the user double click a book in the *JTable,* and that is the following functions

1. Maintenance of publishers
2. Maintenance of categories
3. Maintenance of authors
4. Create book
5. Edit book

I'll start with maintenance of publishers that is selected from the menu and opens the following dialog:



that in a list box displays a list of all the publishers. If you double-click a publisher in the list box, you have chosen the publisher and the name is inserted in the input field. The meaning of the buttons are as follows:

- *Close* close the window.
- *Clear* clear the input field and sets the selection of a publisher to *null.*
- *OK* creates or updates a publisher. If there is not already selected a publisher a new publisher is created and otherwise the selected publisher's name is updated.
- *Delete* deletes the selected publisher. Preliminary the button is active, but after the next iteration it will only be active if the user is administrator.

To implement the function I have written the following code:

1. The class *PublisherData* is expanded with the necessary methods (5 methods) to maintain the *publisher* table in the database. To simplify the class, the class *Db* is expanded with some helper methods, that will be used in other DAL classes.
2. The methods in the adapter class *Publishers* are implemented, such they call the corresponding methods in the class *PublisherData*.

3. To the package *library.ctrls* is added a class *PublisherController*, which is a controller for the above dialog. If a method in the controller can not be executed properly, the method returns false, and the view can then read an error message.
4. There is added a new package *library.views.listeners*, and to this package is added an *interface BookListener*, which defines two methods that can be used to send notifications if the contents of the database is changed.
5. The class GUI is modified and updated with two new methods.
6. The dialog box is defined by the class *PublisherView* in the package *library.views*. The dialog box's constructor has a parameter of the type *BookListener* so that it can send notifications to the main window on the changes of publishers.
7. The class *BooksTable*, which is the data model for the main window's *JTable,* is updated with two methods, which can fire an event if the model is updated.
8. The *MainView* is modified so it implements the interface *BookListener*, and the menu opens the dialog box *PublisherView*.

After the function is implemented, the following test is performed:

1. create two publishers
2. close the dialog box
3. modify one of the new publishers
4. delete the other publisher
5. close the dialog box
6. delete the last test publisher

As the next step I will implement maintenance of categories. The result is a dialog box, which in principle is identical to the above, but it is only necessary to write the following code:

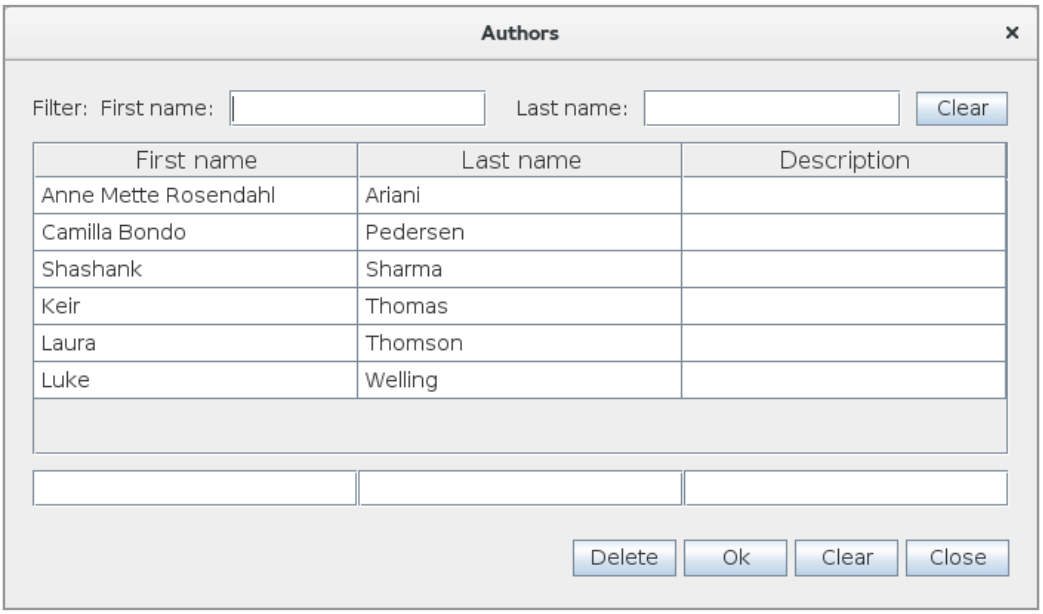
1. The class *CategoryData* must be expanded with the necessary methods.
2. The methods in the adapter class *Categories* should be implemented.
3. To the package *library.ctrls* is added a class *CategoryController*, which is a controller for the dialog.
4. The dialog box is defined by a class *CategoryView* in the package *library.views*.
5. The *MainView* is modified so the menu opens the dialog box *CategoryView*.

The dialog is tested in the same manner as above.

Next step is to implement the function *Maintenance of authors*. The function is similar to the above, but as there may be many authors, the authors are this time presented by a *JTable* (see below), and at the top are fields for a filter. The code is similar to the above, but this time must be added a class to *library.views.tables* that defines a data model for the *JTable* component.

The function is tested as follows:

1. create an author alone with a last name
2. create an author with a first name and a last name
3. create an author with a first name, a last name and a description
4. close the window
5. open the window again and test the filters
6. modify all the three fields for two of the new authors
7. delete one of the new authors
8. close the window
9. open the window again
10. delete the last two new authors



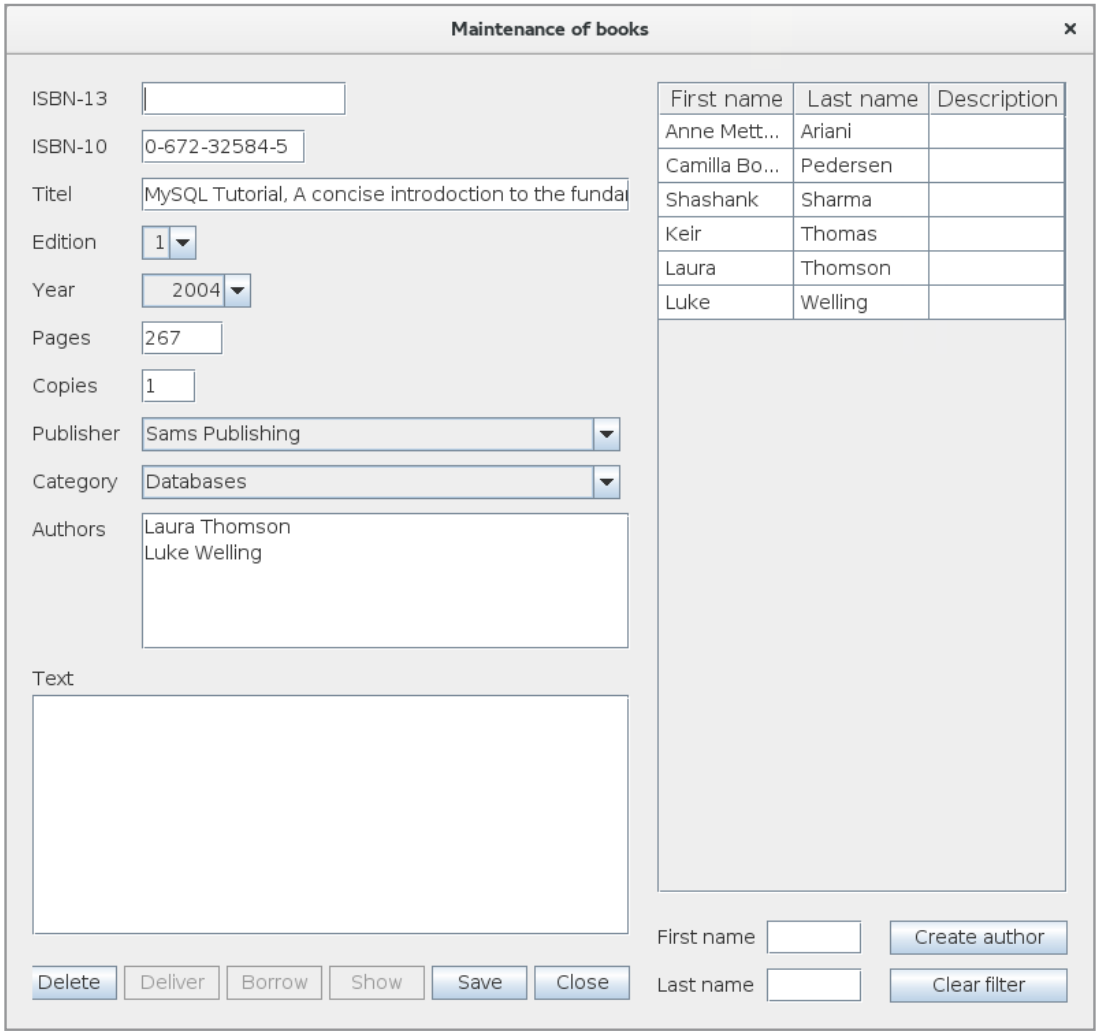
As the last function should I implement the function *Create book*, which is the most complex of all dialog boxes (see below). This is the same dialog box that is used to both create a book and edit a book where the last function is performed by double-clicking on a book in the book list.

The dialog box can be explained as follows. On the left side are the fields for a book, while the right side shows a list of all authors. The fields on the left side should be self-explanatory except the field for authors, that is a list box. If you want to associate an author of a book, you should double-click on the author in the table on the right. If you wish to remove an author from the book, you should double-click on the author in the list box.

The meaning of the buttons on the left side are (from left):

- Allows you to create a publisher if the publisher has not been created and is available in the combo box.
- Allows you to create a category if the category has not been created and is available in the combo box.
- Delete the book (the button will subsequently only be active if the user has the right to delete the book, and if it is not the function *Create book*).
- Return a book – is first implemented in iteration 4.
- Borrow a book – is first implemented in iteration 4.
- Show who has borrowed a book – is first implemented in iteration 4.
- Save the book.
- Close the dialog.

As regards the right side the two input fields are for a filter, while the buttons respectively remove the filter and allows you to create a new author.



To write the code, I have done the following:

- The class *BookData* has been extended with the necessary methods (4 methods) that can maintain the database with respect to books. Since most of these methods must maintain multiple tables, they are written as transactions.
- The adapter class *Books* is correspondingly updated.
- There is added a package *library.util*, which currently only has a single class called *Tools*. The class contains only static methods that are used to validate ISBN, email addresses and more.
- There is added a controller class *BookController*, whose primary task is to validate user entries/choices.

- Since it must be possible to create both categories, publishers and authors from the above dialog there are added three simple dialog boxes to *library.views*:

  - *CreateCategory* that is a simple dialog box used to create a catagory.
  - *CreatePublisher* that is a simple dialog box used to create a publisher.
  - *CreateAuthor* that is a simple dialog box used to create an author.

  Since these are very simple dialog boxes, there is not defined any controller for these windows.

- There is added more interfaces for the project. The package *bibliotek.views.listeners* is expanded with the following interfaces:
  - *CategoryListener*, defining one single method. The class *BookView* implements this interface and the class *CreateCategory* uses the method to send a notification when there is created a new category.
  - *PublisherListener*, defining one single method. The class *BookView* implements this interface and the class *CreatePublisher* uses the method to send a notification when there is created a new publisher.

  - *AuthorListener*, defining one single method. The class *BookView* implements this interface and the class *CreateAuthor* uses the method to send a notification when there is created a new author.
  - *EditListener* also defines a single method, and is implemented by *BookView*. The method is called when another dialog box modifies the contents of a book, so *BookView* can keeps track that the content is changed.
  - *EditTextListener* is a class that implements the *DocumentListener* and works the same way as above and set the content of a window of the type *BookView* as changed, when one of the window's input fields are changed.
  - *EditListListener* is a class that implements *ItemListener* and works the same way as above and set the content of a window of the type *BookView* as changed, when one of the window's combo boxes change selection.
  - *EditDataListener* is a class that implements *ListDataListener* and works the same way as above and set the content of a window of the type *BookView* as changed, when the content of the window's list box with authors is changed.
- Then there is the dialog box, *BookView*, which is a very comprehensive dialog. It will be modified in the next two iterations.
- Finally, *MainView* is updated and attached an event handler for double-click in the table. Double-clicking on a row in the table (a book) *MainView* opens *BookView*, but with data for that book inserted. You then have the option to edit the book and possible delete it.

The function is tested in the following manner:

1. create a book (ISBN 10, title)
2. create a book (title)
3. close the window
4. create a book (ISBN 13, title, edition, year, existing publisher, non existting category, 3 non existing authors)
5. close the window
6. edit the first of the above books (edition, year, pages, non existing publisher, non existing category, one non existing author)
7. edit the second of above books (edition, pages, copies must be 2, non existing publisher, existing category, one non existing author, description)
8. edit the third book (pages, description)
9. create a book (title, description, authors)
10. close the window
11. double-click on the book last created
12. test filter for authors
13. delete the book
14. create 10 random books

5.4.3 USER ADMINISTRATION

In this iteration are implemented the maintenance of user administration, such that each function can only be performed under the four user roles

1. all
2. students
3. librarians
4. administrators

The iteration also includes changing the password.

As a first step, I have created a copy of the project from the previous iteration, and the copy is called *Library3*.

Next I have expanded the model layer and the dal layer:

1. The classes *Title* and *User* in the package *library.models* are implemented.
2. There are added two classes to the dal layer: *TitleData* and *UserData*. The first is similar to the other classes in this package, but the latter has some more methods.

3. Two adapter classes *Titles* and *Users* are added to the package *library.models*.
4. There is added a class *Zipcodes* to *library.dal* representing the zip codes. The class is written as a singleton, where the private constructor initializes a list with the zip codes.
5. There is added a method to the class *Users* that tests whether a zip code exists. This method should not be in this class (due to low cohesion, but in its own class), but for reasons of simplicity, it is located in *Users*.

Regarding the database, it was decided that a title named *Administrator* must denote a super user that should not be deleted or edited. The class *TitleData* tests for that.

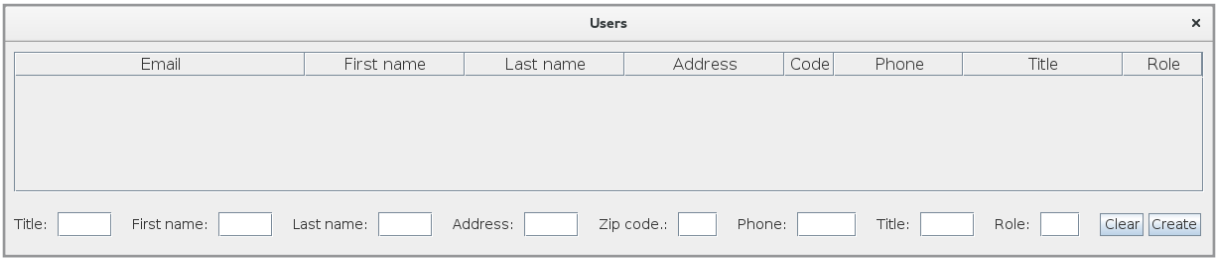
To maintain the title's is added a new menu item in the menu *Administrator* in *MainView*. The function is basically identical with the functions *Maintenance Categories* and *Maintenance Publishers* and opens a dialog box identical to those functions. There is has added two new classes:

- *TitleController* (in *library.ctrls*)
- *TitleView* (in *library.vievs*)

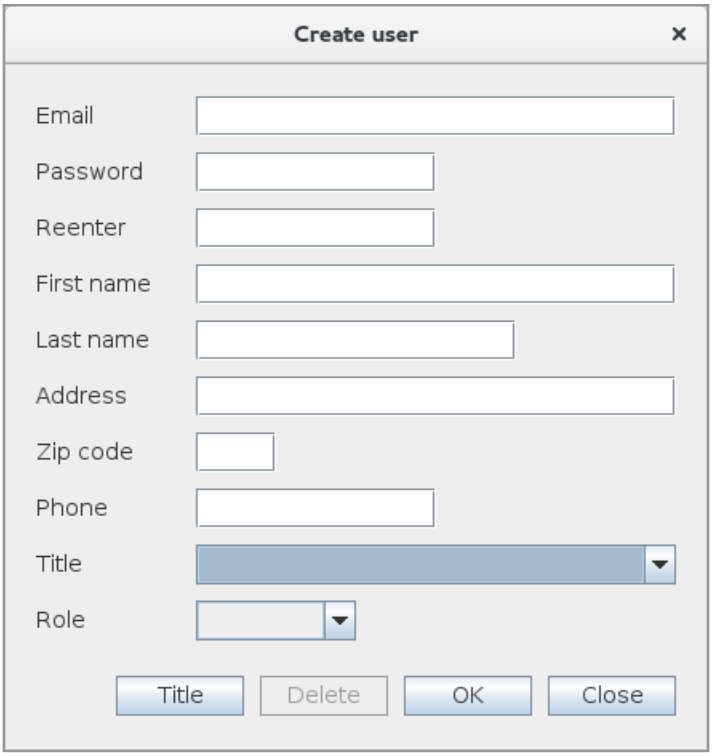
The function is tested in the following manner:

1. create a title
2. create another title
3. create a third title
4. close the window
5. open the window again
6. modify the three titles
7. close the window
8. open the window again
9. delete the three titles
10. close the window
11. open the window again
12. create two titles with the names *Teacher* og *Student*

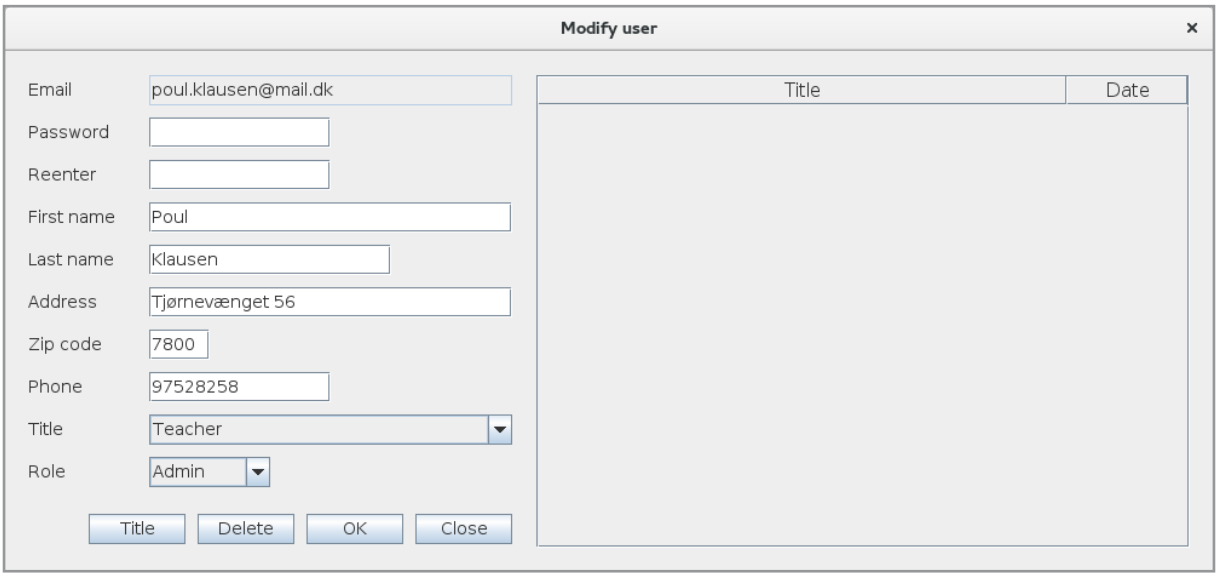
The next step is to implement the function *Maintenance of users*, which is a relatively complex function. The function must open the following window:



that in a *JTable* shows an overview of all users in the database. In addition to the *JTable* is at the bottom filter fields, and the *Clear* button removes a possible filter. The *Create* button opens a dialog box where you can create a user:



If you have created a user and double-click a line in the table with all users, you opens the same dialog (see below), but this time there is shown a table on the right. The table is not used yet, but it should show an overview of the books that this user has borrowed. The table is a *JTable* and it gets only a content in the last iteration.



When you create a user, you select a user role

1. Admin
2. Staf
3. Student

These roles are used for user access rights that are implemented below.

The implementation of *Maintenance of users* adds several new types. The package *library.views.tables* is expanded with two classes:

- *UserTable* which is a data model for the *JTable* showing the overview of all users.
- *BorrowTable*, that in the same way is a data model for the *JTable* shown above and are applied to the books, a user has borrowed. This class is not finished and the objects that table should show is not yet defined.

There is added an interface named *UserListener* to *library.views.listeners* that defines two methods, indicating that a user is created, modified or deleted. The interface is implemented by *UsersView*.

There is added a controller class for *UserView* to *library.ctrls*. The class is called *UserController*.

Finally is added two view classes:

- *UsersView*, that shows the overview over all users.
- *UserView*, that is used to edit a user.

The following is done to test the function:

1. create three users
2. close the window
3. open the window again
4. modify each of the three users
5. close the window
6. open the window again
7. delete the three users
8. create a user as Admin
9. create a user as Staf
10. create a user as Student

When the program is opened the first time, there is in principle no users and no titles. It has been decided that there must always exsists a super user who is known by the user role 0, and entitled *Administrator*. When the program starts, it tests whether this user and title are available, and if not, the program will automatically open a dialog box so you can create this super user, and the title *Administrator.* Both this title and the super user must not be deleted or changed.

There have been updates to the classes *UserData*, *Users* and *UserController*. Furthermore, to the package *library.views* are added the following classes

- *AdminView* which is almost the same dialog as *UserView* but so you can not select a title and a role.
- *SuperView*, which is a simple dialog box for entering the super user's password.

Furthermore, there is added an interface named *LogonListner* as *MainView* implements. It defines a single method that is called from *SuperView* when you have entered a correct password.

Back in this iteration is the implementation of the user access rights. It includes the following:

1. There is added a class *CurrentUser* to *library.models*. The class is implemented as a singleton and it represents the current user. After the program is started, there is no user – no user is loged in.
2. There is added a new controller, called *LogonController* that is controller for the dialog box below.
3. There is added a dialog *LogonView* to logon. The class can be found in *library.views*.
4. An additional controller called *PasswordController* is added as controller for the following dialog box.
5. There are added yet a view to *library.views*. The dialog box is called *PasswordView* and is used by the current user to change password.
6. Finally is added a dialog box *DetailsView* to library.views, which opens if a *Student* double click on a book in the overview. The reason is that a user with *Student* access only should be able to se the book's details, but not to change data. A student can borrow the book, but this is first implemented in the next iteration.

I have also changed the class *MainView*, so more menu items are active, but so that there is only access to the individual menu items depending on the user access rights. In the same way that are changed some dialog boxes, so some of the buttons are inactive, depending on the current user.

The result of the above is tested as follows:

1. when the program opens, there should only be access to *MainView* with the overview over all books (the user should be able to apply the filters) and the menu *General*
2. log in as student – there should now further be access to the menu *Users*
3. double-click on a book to test the dialog box *DetailsView*
4. change the password for the current user

5. log out
6. test that there is only access to the menu *General* and you not can open the dialog box *DetailsView*
7. log in as the same user again (with the new password)
8. log in (without first logging out) as staf – there should now further be access to the menu *Librarians*
9. test all functions under the menu *Librarians* are working as they should and the *Delete* buttons are inactive
10. log out and test that there is only access to the menu *General*
11. log in as admin – there should now further be access to the menu *Administrator*
12. test that the functions under the menu *Administrator* (except the first) works as they should
13. test all functions under the menu *Librarians* works as they should and the *Delete* buttons are active (all buttons used to borrow books should not work)
14. log out
15. log in as the super user
16. log out

5.4.4 BORROWING AND SEARCH

In the last iteration I lack to implement the borrowing of books and advanced search. Furthermore, the problem with the database parameters must resolved so that they are move into to a configuration file.

As in the other iterations, I have created a copy of the project from the previous iteration, and the copy is called *Library4*.

I will start implementing borrowing of books, so users can borrow and return books. In principle, it is simple enough, as there is to some extent made preparations, but the classes must be corrected in several places.

1. The class Borrow is changed, and there is added a field *user*, representing the user, that borrows a book, and the class's methods are implemented.
2. There is also defined a class *BorrowData* in the data access layer that encapsulates the database table *borrow* and also a corresponding adapter class in the model layer, called *Borrows*.
3. To show all borrows of af book there is added a view class *BorrowedView* showing which users have borrowed a particular book. Because this class applies a *JTable*, there is also added a data model to *library.views.tables* called *BorrowedTable*.
4. The class *DetailsView* is changed so that the three buttons relating to borrowing books now are active, and such a student can borrow and return books.
5. The class *BookView* is changed accordingly so that the three buttons for borrowing books are active.
6. The class *UserView* is changed so that the table that shows the books that a user has borrowed, now has a content. The class *BorrowTable* has been changed accordingly. If you double-click on a book in the *JTable* for borrowed books, you get the opportunity to cancel the borrow (the user is adminstrator).

The following test requires that there is a book with at least 2 copies.

1. log in as a student
2. open *DetailsView* for a book and borrow the book
3. close *DetailsView* and open it again – it should not be possible to borrow the book again
4. close *DetailsView*
5. log in as staf
6. open *BookView* (for the same book)

7. check who has borrowed the book – *BorrowedView*
8. borrow the book
9. close *BookView*
10. log in as adminstrator
11. open *Maintaining users*
12. open *UserView* for the staf user
13. cancel the borrow of the book
14. log in as student again
15. check who has borrowed the book – there should be one that is the current user
16. returns the book

The next step is to implement recall of books from the menu *Administrator*.

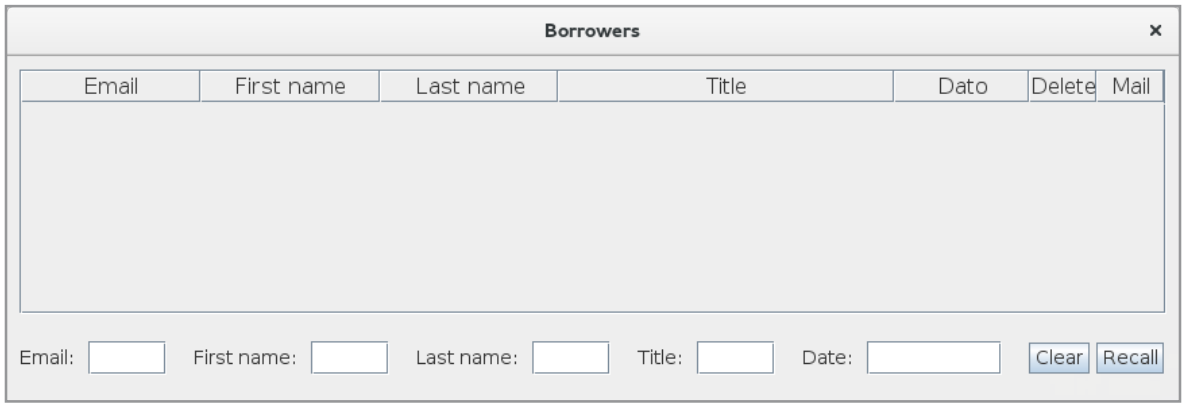
For an application can send mails, there must on the machine be installed an extension to Java, an API called *JavaMail.* You can download a jar file (there is only one) from Oracle and place it in the right place. That is all.

To the package *library.util* is adding a class called *Mail*, that uses the above API to send mails. After that all the technical stuff is in place with respect to send mails. So far, all the parameters related to mail are hard coded, but it must of course be changed later.

There is added a simple dialog box called *MailView*, where you with two dates can enter the period of recall notices. This is the dialog box that opens from the menu. There is also added a corresponding controller, called *MailController*.

The dialog box *MailsView* consists of a *JTable* showing all borrows within the selected period (see below). Basically, the window is a JTable with an associated filter. If you double-click on the title for a borrow you opens the book's *DetailsView*, and if you double-click on one of the other columns you opens a similar dialog box with information about the user. The dialog box is called *BorrowerView*. For *MailsView* is thus added the following classes:

- *MailsTabel*, that is the data model for the *JTable* component
- *BorrowerView*, that is the dialog box with user information
- *MailsController*, that is controller for the class *MailsView*. It is this class that sends mails by using the class *Mail* in *library.util*.



The table has rightmost two columns, where to put check marks. In the first column you can marks the borrows that simply must be canceled, while in the other column, are used select the borrows to be called home.

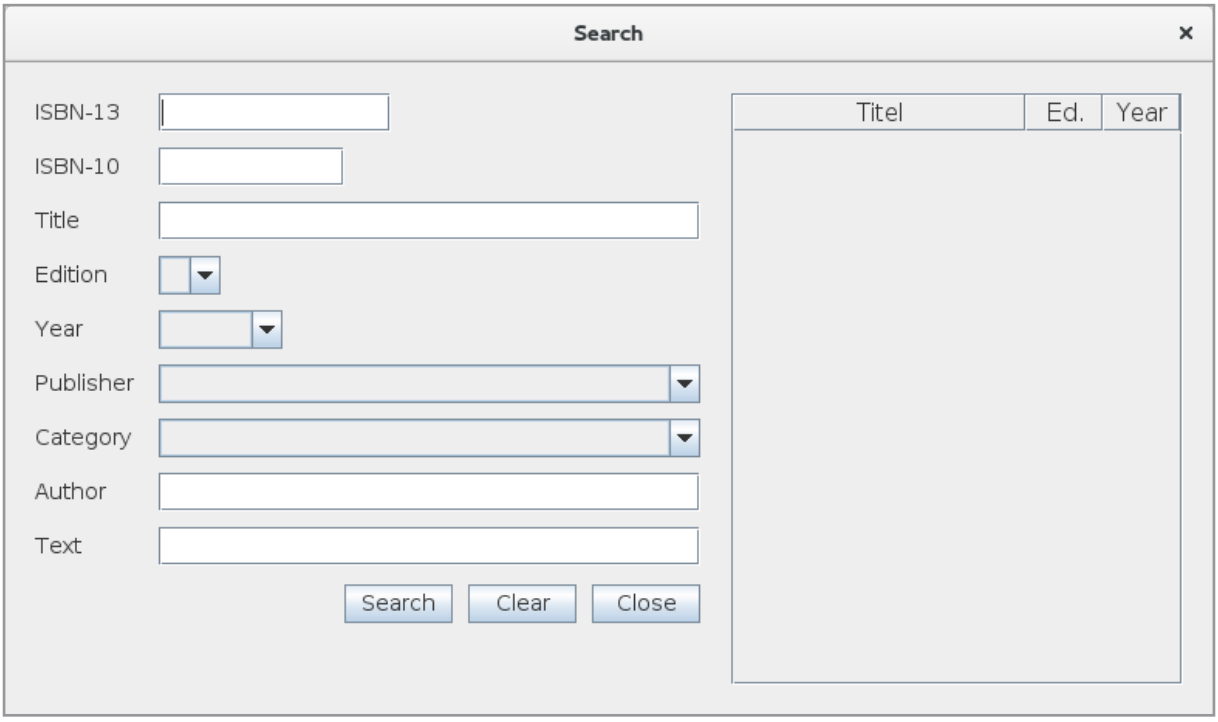
To test the functionality I have done the following:

1. create a user with a mail address that I can use (have access to)
2. create another user with a mail address that I can use (have access to)
3. borrow a book to the first user

4. borrow a book to the second user
5. borrow a book to a third user
6. log in as adminstrator
7. select the function *Recall*
8. do not enter anything for period – all borrows are selected
9. double click om the last name for one of the users to se information about the borrower
10. double click on a title to se the details about the book
11. mark recal for the two first users and delete for the third user
12. click *Recall* and test that the functions are performed correct

What remains is to implement the search function. The function displays a window as shown below, where you can enter/select the *search criteria*. When you click *Search*, the books that match are displayed in the table to the right, and if you double-click on a line in the table either *DetailView* or the *BookView* opens depending on which user you are.

In principle, it is simple to implement the function, and the only complex is writing the SQL expression to be applied to a concrete search. This is based on the search criteria in the class *BookData* and the criteria are prepared in class *Books*. The dialog is implemented by means of two classes: *SearchView* (in *library.views*) and *SearchTable* (in *library.views.tables*). To test the function, you can not do other than to perform multiple searches. It is difficult to test the function adequately in this place, because the data may be too small.



Finally, there is added another feature to the menu *Administrator* for maintenance of the super user.

Parameters for both the database and the mail server is hard coded, what obviously does not hold. These values should be moved to a configuration file. It's called *library.config* and is preliminary in my home directory and have the following form:

```
dbhost:***
dbport:***
dbdata:***
dbuser:***
dbcode:***
mlhost:***
mlport:***
mluser:***
mlcode:***
```

where *** should be replaced of current values. The meaning is as follows:

- *dbhost* is the name of the database server
- *dbport* is the port, that the database server use
- *dbdata* is the name of database
- *dbuser* is the name of database user
- *dbcode* is the password for the database server
- *mlhost* is the name of the mail server (smtp server)
- *mlport* is the port, that the mail server use
- *mluser* is the mail server's email address
- *mlcode* is the password for the mail server

The project has been expanded with a class *Servers* in the package *library.util*. The class is programmed as a singleton, where the private constructor reads the above configuration file, and the class has methods that returns a connection to each of the two servers. Next, the two classes *Db* and *Mail* are modified where the specific server information is gone and so they instead use the class *Servers*.

When the configuraion file not necessarily should be located in my home directory (but for example in */etc/opt*), is added the following statement to *main()*:

```
if (args != null && args.length > 0) library.util.Servers.setPath(args[0]);
```

This means that you by a parameter can specify where the configuration file can be found.

Test of this change is simple:

1. create a configuration file and place it in your home directory
2. test if the program can run
3. move the configuration file to another directory
4. start the program from a terminal and specify the configuration file as a parameter and test if the program still can run
5. test where appropriate to send a mail for a recall

5.4.5 A LAST ITERATION

Now, the program is finished in principle. Although there still is not talking about a large program, it consists nevertheless of 84 types in the form of interfaces and classes. No matter how careful you have been in the program development, there is a large chanche that some of the code is improper, that there is code repetitions and something that could be written in a better way. It is therefore worthwhile to complete the development of a program with a code review, where you also should comment the code (to the extent it is not already done) and remove inconveniences (and of course direct errors). It is a big work, not the world's most interesting work, but the work is certainly well spent. Especially should you pay attention to variables or methods that are not used, and that kind should be removed.

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

The code review starts, like the other iterations to create a copy of the project, and I created a copy named *Library5*.

As a result of the code review, I have among other things, changed the class *BookData*. I have improved the class by optimizing the SQL SELECT statements, such they JOIN more tables. The result is that a part of the work in this way is been moved to the database server that is inherently more efficient. Moreover, I have reviewed all classes for the user interface again, where I have made typical cleanup and controlled that texts are correct.

When developing a program as the program *Library*, there will be many classes with common features and for example dialog boxes that are very similar. Here it is natural to copy the code already written and then modify it (and also common copy and paste). It is of course nothing wrong with that, but you should be aware that it is one of the very major sources of errors, simply because you do not get corrected in all places. Therefore, a warning: Be careful when you copy code.

5.5 TEST

The program has been tested as follows:

1. all borrows are deleted (the function *Recall*)
2. all books which makes no sense are deleted
3. all authors that are not relevant are deleted
4. all categories that are not relevant are deleted
5. all publishers that are not relevant are deleted
6. all users (but not the super user) are deleted
7. all titles (but not Administrator) are deleted

Then I closed the program and manually used MySQL Workbench to delete the super user and the title *Administrator* and ensure that the tables *title*, *user*, and *borrow* are all empty.

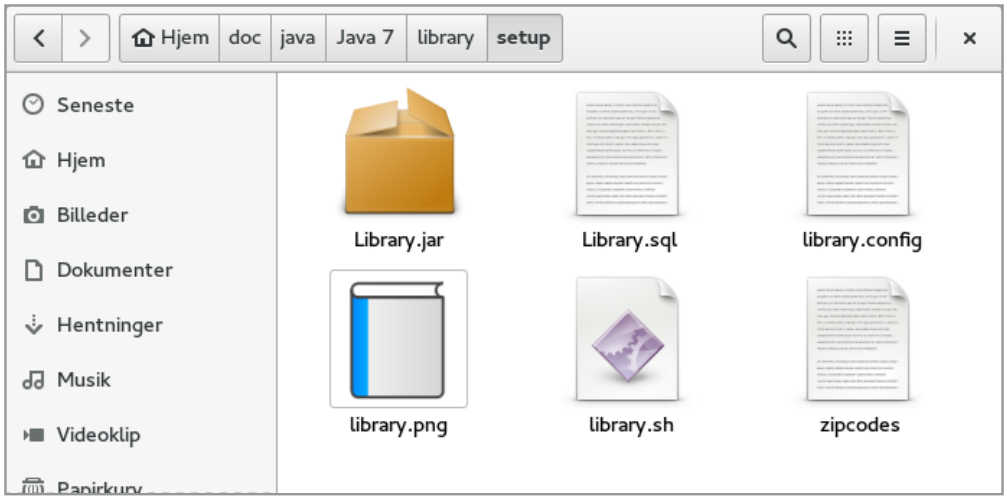
The I have:

1. started the program and created the super user
2. loged in as super user
3. created three users (with real mail adsresses) that has different user roles (Admin, Staf, Student) and related appropriate titles.
4. loged in as Staf
5. created 20 (real) books such that more books are created party for latter to be updated, and so that the function is performed in multiple sessions
6. loged in as Student and borrowed 5 books
7. loged in as Staf and borrowed 5 books, where 2 books are the same books borrowed to the Student user (some books must have several copies)
8. loged in as Admin and borrowed 5 books, where one book is the same book also borrowed to the Student, and one book is the same book borrowed both to the Student and the Staf user.
9. tested that everything regarding borrowing look right
10. performed recall of all the borrowed books

When this test is performed properly is back to test the program with many books and many users. This test can only be performed with the customer and in the right environment. This means that the customer must provide correct data, both books and users. This data may, for example, take the form of a CSV file, and you can then write a small program, which from this file can update the database. Then you can perform the above test again, so you are sure that the test is performed satisfactorily when the database has a large data volume.

5.6 DELIVERY

The last phase is to prepare the program for delivery. It takes an icon and the development of an installation script. The necessary files are assembled in a directory with the following content:



I will not show the installation script here. To install the program you must:

1. copy the file *zicodes* to */var/lib/mysql-files/data*
2. open *MySql Workbench* and perform the script *library.sql*
3. update the configuration file *library.config* with the correct parameters
4. open a terminal and change currrent directory to the *setup* folder
5. perform the command: *sudo ./library.sh*

The result should be, that you get an icon on the desktop, ad you can start the program.

APPENDIX A

I have repeatedly mentioned UML as diagrams that can be used in system development. I am not a great supporter of UML, but must also admits that the diagrams also has there uses. Therefore, this appendix provides an overview of the main rules and diagram syntax.

As described above, you in the analysis and primarily in design has to build a model of a program. This means that you must decide how the program should be written and including, for example, which classes the program should consist of. To that you can use diagrams that can illustrate the program's architecture and key components, and the advantage is partly that you can work visually with the program's structure, where you with a diagram can draw a model that would otherwise have to be explained with many words, and the model not dependents on the programming language's many details.

My earlier suggested that I am not so great a supporter of the diagrams, is because they are difficult to draw (it takes a long time), and they are time-consuming to maintain, if the design changes during the development process. Another problem is that the diagrams must have value, and they must be understood by everyone and it is therefore necessary to know the rules, and these rules tend to be complex and difficult to learn and remember. The first problem can be solved by using specific case tools that support the development of such UML diagrams, while the second problem is merely a matter of learning the diagram's syntax.

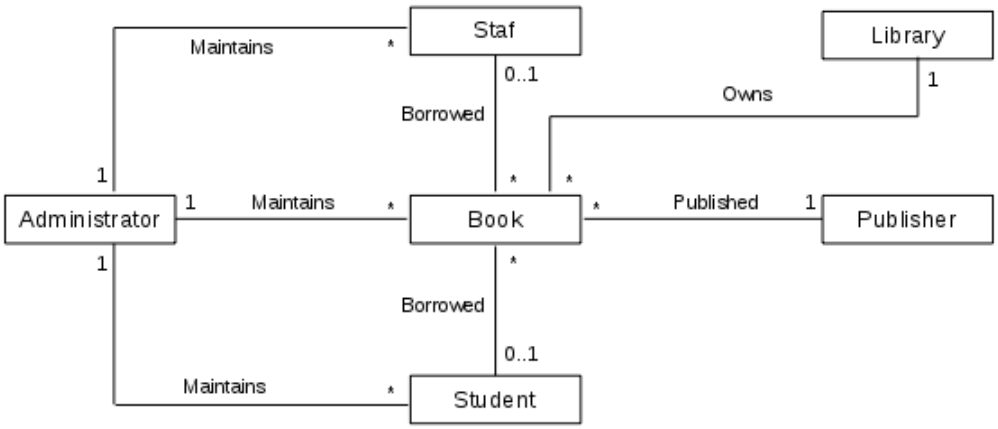
UML stands for *Unified Modeling Language* and is the diagramming intended to models software in system development. UML defines various types of diagrams, which are aimed at both analysis and design, and in the following I will briefly describe the most important of these diagrams and including the most basic drawing rules. The idea is that by means of diagrams you can draw a model of a program including the most important algorithms, and whatever I have said above, there is certainly good uses. If you are participating in a project to develop a software solution, then UML is excellent to support the process where you jointly by example on a blackboard are modeling a system using UML diagrams. Here I find modeling diagrams much more important than as documentation of the final solution. However, one must also not here underestimate the value – especially for creating an overview, but very detailed diagrams, I find rarely valuable and I do not feel that the value is commensurate with the resources it takes to produce them.

In short, I find diagrams useful as part of the system development process and as a documentation to provide an overview of a large and complex system.

In this appendix, I will as mentioned introduce UML. There are many rules and not all are mentioned, but the emphasis is on the rules – and diagrams – which I find most useful. The diagrams are all drawn using *LibreOffice Draw*, and most examples are taken from the project in this book and the development of a library program and will partly illustrate the basically drawing rules, and also where and how diagrams can be used.

DOMAIN MODEL

I will start with a *domain model*, which is a diagram that during the analysis can be used to illustrate the main concepts that are part of a software solution. The diagram consists of conceptual classes that are drawn as rectangles, as well as associations between these classes. Below is a domain model for the library program:



The diagram shows the key concepts for this program. It's a simple diagram, but it can be useful at a preliminary stage in the analysis to provide an overview of the key concepts in the upcoming program's problem area. In this case there is shown 6 key concepts, but it is by no means clear which concepts to be there, and in this case there could well be more. However, you should avoid making the diagram to detailed and keep focused on the main concepts. It is important to note that the classes in the domain model is not the same as software classes. The domain model will be an inspiration for the classes that the program should consist of, and many classes from a domain model will subsequently also at the design could be recovered as software classes, but not necessarily all. When you draw a domain model you must stay focused on the conceptual concepts and not thinking in software engineering.

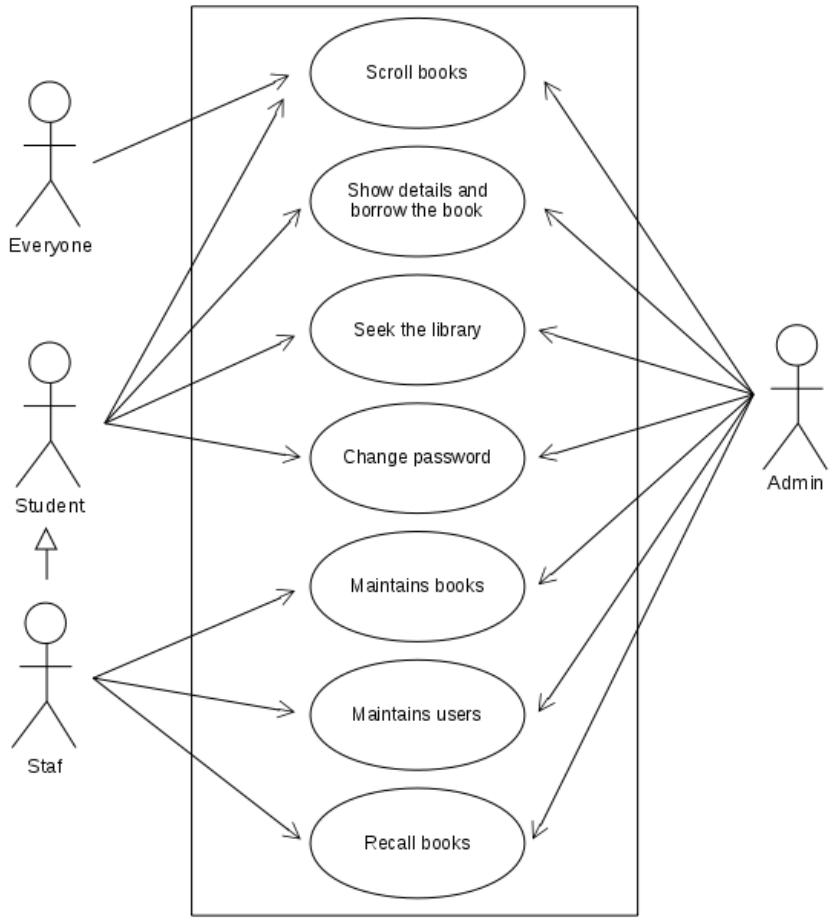
Between the domain model classes, there may be associations that are drawn as straight lines and possible also with a name, as shown above. An association expressing one or another connection or relationship between the model's classes. For example there is an association between *Book* and *Publisher* corresponding to that a book is published by a publishing house. For associations may be attached multiplicities, and as such indicates the association *Published* that a publisher may be associated with 0 or more books, while a book is associated with exactly one publisher. You can use the following multiplicities:

- * 0 or more
- 1..* 1 or more
- 1..n 1 to n
- n exactly n
- k,m,n exactly k, m or n

A domain model will consist of at least conceptual classes and associations between these classes but not necessarily multiplicities, and often you do not give the associations names. If you like, you can also choose to assign attributes to classes such as shown below, which may help to explain what a conceptual class means, but often it will just complicate the diagram without adding extra documentation. However, there is no sharp distinction between what are classes and what are the attributes, and applying classes with attributes, you also get the chance to show that you found all the main concepts.

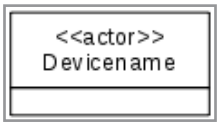
USE CASE DIAGRAM

A use case diagram is used to model the environments interaction with the program and is also an example of a diagram which is used during the analysis. Here, it is used typically for illustrating user interaction with the system. There are two concepts in the diagram. One is an *actor* that indicates an entity that program interacts with. It will often be a person, but it may also be a hardware device or another system. The second element is a use case that is the name of a job that the program must perform. The following is a use case diagram for the library system.



Use cases are drawn as ellipses with a short text telling what the use case is about. A use case is an operation (a user's interaction) with the program within a short period of time that can be perceived as a single session. For example an actor can perform a search in the library and have as a result a number of titles that matches the search criteria. Similarly, a user can view details about a book and possibly borrow it.

Actors are drawn with a symbol to illustere a user and with a name below. In this case, there are four actors, which are all persons, but the actors do not have to be a person, but may, for example also be a harware device or another program. If so, you sometimes draw an actor as follows:



The interaction between an actor and a use case is shown with an arrow (sometimes just a non-oriented line). In this case, the arrows each time goes from the actor to the use case, but it need not be the case and such it may be the application that sends a message to a hardware device.

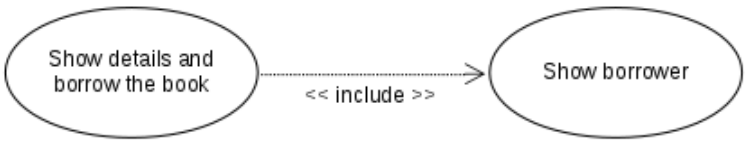
In particular, it is allowed that an actor inherits another actor, and it simply means that the actor who inherits does everything (use the same use cases) that the actor who is inherited can plus any other use cases. In the diagram above it is illustrated by the fact that the actor *Staf* inherits *Student*. An important consequence of inheritance is that it can make the diagram simpler (fewer arrows) and thus more useful to read.

Sometimes you also choose (see above) to draw a rectangle around all use cases, where everything inside the rectangle is perceived as the system, while the actors are outside and are external.

One of the challenges with a use case diagram is what actually are use cases, and as mentioned above, it should be an action that is limited in time and has the nature of a session with the system. Looking at the above diagram more of the listed use cases are not really use cases, but a collection of use cases. Consider as an example *Maintains books*. It actually consists of several use cases:

- Create book
- Modify book
- Remove book
- Borrow book
- Return book

and the same applies in several places. The problem is that the number of use cases easily become very large and the diagram corresponding confusing. Do you have to be absolutely correct, it would typically be required to draw more use case diagrams. Sometimes includes a use case using another use case, and it can in the diagram be show as:



Similarly, a use case be an extension of another use case, which you can show as follows:

Use case diagrams can not stand alone – at least not as shown in the above example. It is necessary to have a description of the use case:

1. The name and a brief description in the form of a text.
2. The formal requirements and including the result of performing the use case, and it can be seen as a contract where the use case guarantees a certain result in the case that it is carried under certain conditions.
3. Pre conditions are conditions that must be met for the specified use case may be performed and these pre conditions can be seen as part of the contract.
4. Scenarios, as a formal description of the sequence of actions or events that occur between the actor and the system when the use case is performed.

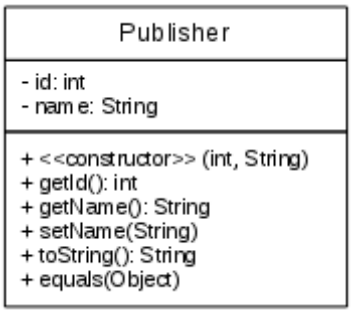
An example of a formal documentation of a use case could be:

Use case:	Seek the libray
Name:	SeekLibray
Description:	The user enter or selects search criteria Isbn13, that must start with the search text Isbn10, that must start with the search text Title, that must contains the search text Edition Year Publisher Category Author as first name and last name separated by a space Text, that must contains the search text If a search criteria is blank or missing it should be ignored. The result is a list of the books that match the search criteria. Are there no books that match, the list is empty.
Operations:	Enter search criteria Select a criteria for *Edition*, *Year*, *Publisher*, *Category* Click *Search*
Preconditions:	If there are selected a *Publisher* or a *Category* they must exists.
Postconditions:	There is created a list with the *Book* objects, that matches the search criteria. The content of the database is not changed.

As perhaps clear from the foregoing it may be comprehensive to draw use case diagrams with corresponding description of the specific use cases, but to make a complete use case documentation you have also made most of the requirements specification, and you can perceive the use cases as a formalized method to preparation of the requirement specification.

CLASS DIAGRAM

The most commonly used UML diagram is undoubtedly the class diagram, which is a diagram used in the design, and which shows many of the program's classes. It is also the most complex diagram with many drawing rules. The diagram is usually used in the model layer and the controller layer. A class is modeled as follows:



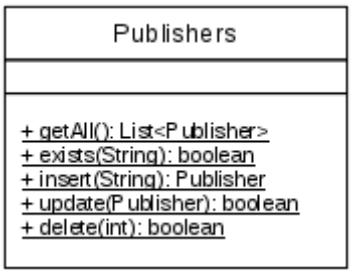
and represents the class *Publisher*. The figure consists of three rectangles, where the top is the name of the class, the middle is the class's variables, while the last is the class's methods. Members (both variables and methods) preceded by a + are public, while members preceded by a – (minus) are private. A figure as above therefore shows what a class consisting of and including the members' visibility. Note especially the syntax of a constructor. Formally you should also specify the name, as for example

```
+ <<constructor>> Publisher(int, String)
```

but the problem is that the line easily becomes long and the class corresponding will fill a lot, and as a constructor in Java has the same name as the class, I allows to omit the name. Also note that I have defined types for both variables and methods. It is not always I do that, and if you feel to achieve the same without these types, it's okay. It is important to remember that the meaning of the diagrams is to give an overview.

There is some more drawing rules, and below shows a class that defines a variable that has package visibility, as well as a method that is protected:

As another example, the following class models the class *Publishers*:



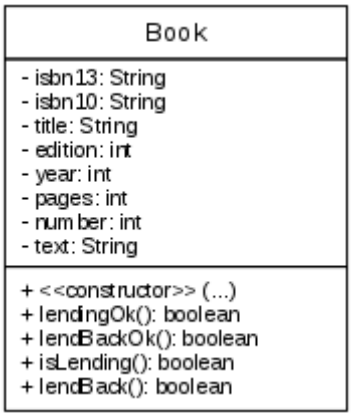
This class has no variables and the methods names are underlined. The later says that it's static methods, and UML uses the same syntax for static variables. As a final example the figure below shows a model of the class *Book*:

It is a very large class, and is not very readable (and actually the class *Book* is not particularly large, and programs will often contain classes that are much larger), so a little about how you appropriately can model classes. It has also something to do with what class models should be used for: Shall they be used to document how the program should be written, or should they be used to document how the program is written, and in my opinion they should definitely be used for the first.

With regard to variables you should specify only the essential, and therefore the variables that are necessary to understand what it is for a kind of objects the class models. In this case it is in fact also the case, perhaps except for the variable *id*, which is used to identify a specific object with a number. This variable is alone used to solve a technical problem, because the library may have books that do not have an ISBN.

With regard to methods apply roughly the same, that you only need to include methods that describe something essential, and as such it may be appropriate not to write get and set methods and leave it up to the programmer to decide which of these methods is necessary. In the same way I will only show the class has a constructor, but if there are more, I show only one.

Correspondingly, the class can be conveniently modeled as follows:



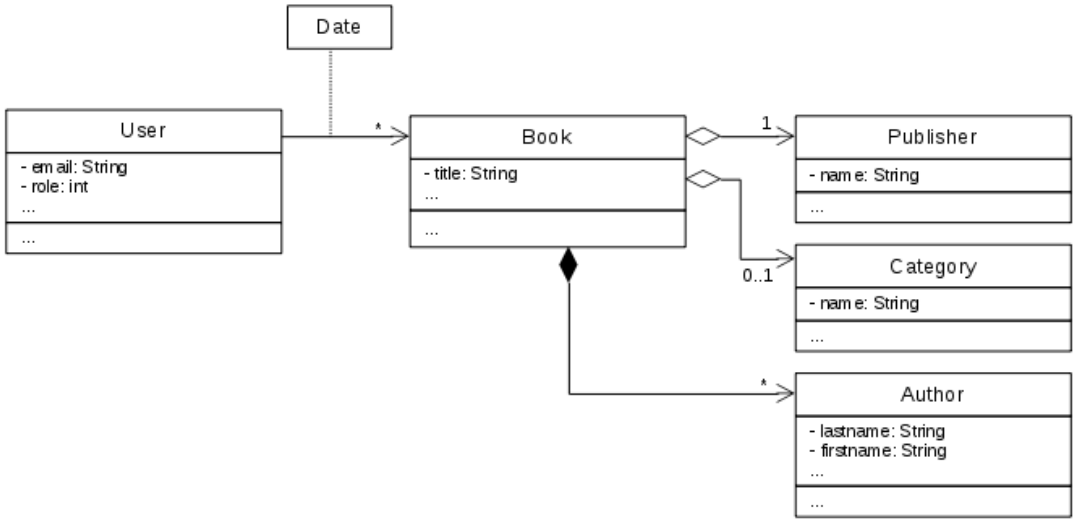
If we again consider the first version of the class model, you can see that there are get and set methods for *Publisher*, *Category* and *Author*, and thus as if the class is missing some variables. It is, however, not the case (not here at design time) as they corresponds to the relationships between classes.

If you considers the above class models, they will usually require additional documentation, where it as minimum is necessary to document the methods, and UML also has syntax on how to do that. I prefer, however, to document the methods with plain text that briefly explain the meaning of each method – and possible also variables.

If there are multiple classes, there will be relationships between these classes and, in general it can be

1. association
2. aggregation
3. composition
4. inheritance

These relationships are illustrated by a class diagram, and below shows a class diagram for the model layer in the library program. Let me immediately say that the diagram is not drawn correctly, since the types of relationships are chosen to show the syntax. If you look at the classes *User* and *Book*, there is shown a relationship between the classes. It was drawn as an arrow from User to Book, and lists a multiplicity indicating that a *User* may be related to more *Book* objects. It is an example of an association which indicate a loose relationship, such that it does not necessarily exist in all the program's lifetime. In this case, indicate the relationship, that a *User* has borrowed a book, and when the user at a time handing the book back the relationship disappears. Therefore, it is shown as an association. That the multiplicity is shown as * means that a user can borrow more books. This means that the relationship must be implemented as a list of objects of the type *Book* in the class *User*.

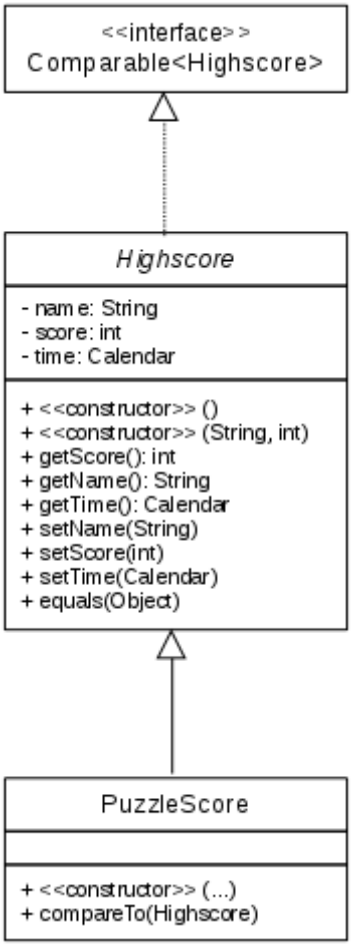


There is also a relationship between the class *Book* and the class *Publisher*, which, because of the multiplicity tells that there is associated exactly one *Publisher* object to a *Book* object. The relationship is implemented as an instance variable of type *Publisher* in the class *Book*. The relationship is shown as an *Aggregation*, meaning a fixed connection between the two objects. A book does not change publisher, and one can perceive an aggregation in the way that it refers to something that the book consist of. The aggregated object can, however, have its own life and can exist even if the aggregation disappears and the object can be aggregated to multiple *Book* objects. An aggregation thus reflects a stronger relationship than an association.

Similarly, the class *Category* is aggregated with the class *Book*, but this may indicated with the multiplicity that a book does not necessarily have a category, but if it has, the category does not change.

The diagram also shows a relationship between *Book* and *Author*, and when the multiplicity is *, it must be implemented as a list with *Author* objects in the class *Book*. The relationship is also designed as an aggregation, but it is a stronger aggregation, which is called a composition. The difference is that the authors are again seen as part of the book, but with a stronger connection, such that if a *Book* object is deleted then also the *Author* objects that are part of it must be deleted. That is, the author objects can not have their own lives, which is not quite correct (the implementation of the program are different). An author can in principle have written several books and thus be associated with multiple *Book* objects. When I above have shown the relationship as a composition, it is only to show the syntax.

Then there is inheritance, and to illustrate this I consider classes from the program *Puzzles*. The class *HighScore* is modulated as other classes, but you should note that the name is shown in italics. This means that the class is abstract. It is because it implements the interfacse *Comparable<HighScore>*, and when the class does not implement the method *compareTo()*, it must be abstract. You should note how you in a class diagram shows an interface and how a dotted arrow shows that a class implements this interface. Also note that the arrow is shown as an open triangle that generally means inheritance.



The class *PuzzleScore* is a concrete class that generalizes the class *HighScore*, and in the case of a derived class rather than implementation of an interface it is shown with a solid arrow.

The above is the most important of the class diagram, but there is more. However, above, wil in most examples be sufficient, and it is as mentioned the most widely used UML diagram.

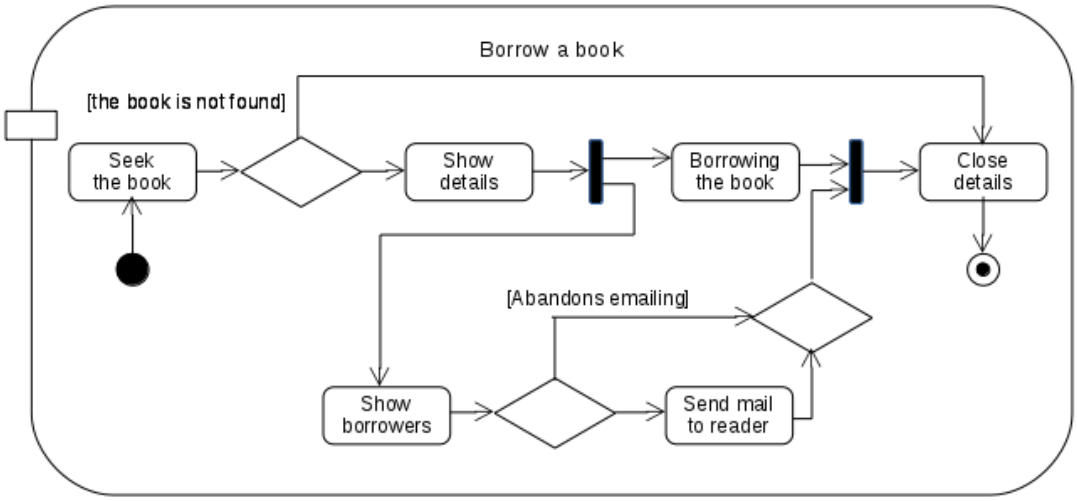
Above is mentioned three diagrams. The first two

- domain model
- use case diagram

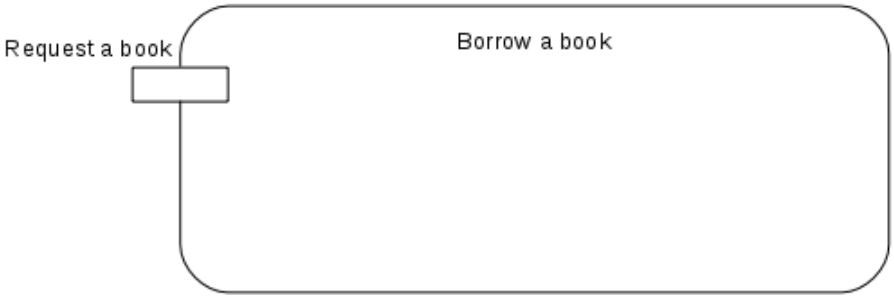
are aimed at the analysis phase, and the class diagram is used in the design phase. I will show another four diagrams, all diagrams to be used in the design phase.

ACTIVITY DIAGRAM

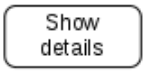
An activity diagram is used to show a sequence of operations, and thus a particular workflow from start to finish, which illustrates the different paths that are followed on basis of various resolutions. Activity diagrams are typically used in the controller layer to describe business processes. An example of an activity diagram for the activity that a user wants to borrow a book might be:



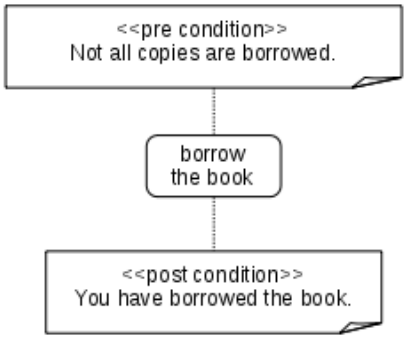
An activity is shown as a rectangle with rounded corners, where the top has a short descriptive text:



and the activity contains a number of actions that are illustrated by rectangles with rounded corners:

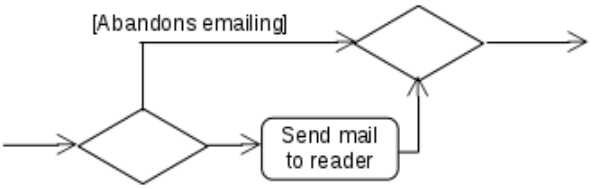


The flow between actions and conditions are displayed with arrows that along with the diagram's symbols show roads that can be followed from a start symbol to a final symbol. The start element is a filled circle, whereas the final element is a circle with a black dot inside. There are also an alternative finish symbol with a cross inside, and the difference is that the first ends the entire activity, while the latter just finishing a road within the activity. It is also possible to assign both pre conditions and post conditions for an action:



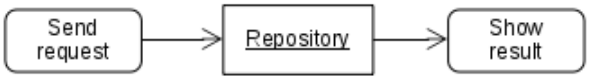
that partly describes when the operation can be performed, and what the results are (but not how the operation is performed):

A diamod represents a branch or a merge that gathers roads. That is that the same symbol is applied and you can add a name. In the case of a branch, a condition in square brackets is applied to a path. As an example, the above diagram both has a branch and a merge:

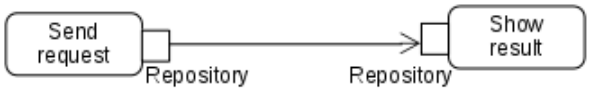


A diagram may also contain fork and join nodes that are drawn as a black bar (vertical or horizontal), and again are used the same symbol for both. They indicate the start and end of simultaneous threads (in the example above, the symbols are actually used incorrectly, since there is no case of multiple threads, and the example should also only show the syntax). A join is thus different from a merge, as a join synchronizes the two incoming paths and provide a single result which is first available after both the input paths is closed.

There is some more drawing rules associated with an activity diagram. For example it can be shown that an action will send data for processing by an object in the following manner:

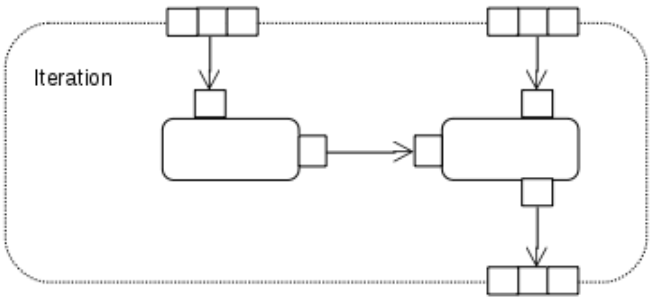


An alternative notation for the same are:



It is also possible to illustere an exception handling:

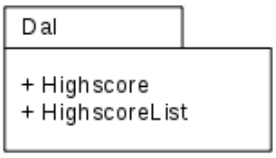
Below is shown the syntax for a loop:



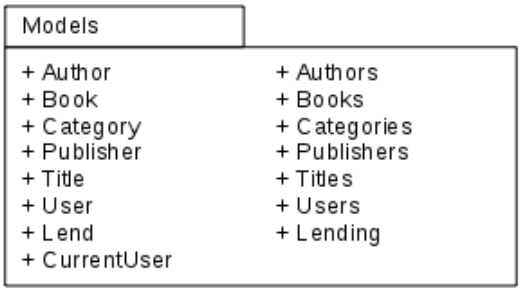
The figure is called an expansion region and is a structureret activity carried out several times. Input and output are displayed as a group of three squares and a label may indicate *iteration*, *parallel* or *stream*.

PACKAGE DIAGRAM

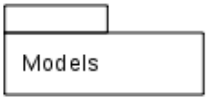
Next, I will mention a *package diagram*, which as the name says, is used to provide a top view of an application's packages and how the program's classes are organized into packages, and one sometimes says that package diagrams are visualizing an application's namespaces. A package diagram is useful to show the program's architecture. A package is drawing as follows:



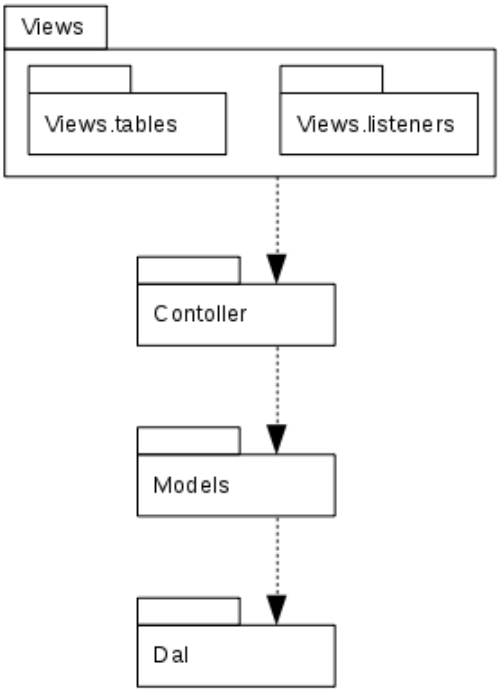
showing a package called *Dal* which includes two public classes. As another example below shows a package from the the program *Library*:



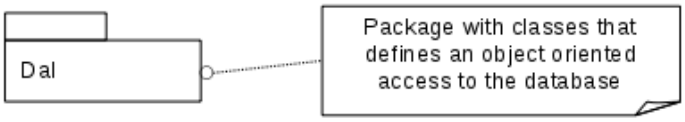
Actually, it's not always you are interested in showing which classes a package contains, but only that the architecture including the particular packages. You then uses the notation:



A package diagram is an overview of a program's packages, and as an example the following package diagram shows a package diagram for the program library. The diagram shows the program's architecture and what packages are in the individual layers.


Views
Views.tables
Views.listeners
Contoller
Models
Dal

As shown, a package diagram is a very simple diagram that may be used to model the architecture of a design, but there are in fact a lot of more syntax. For example you can assign comments:

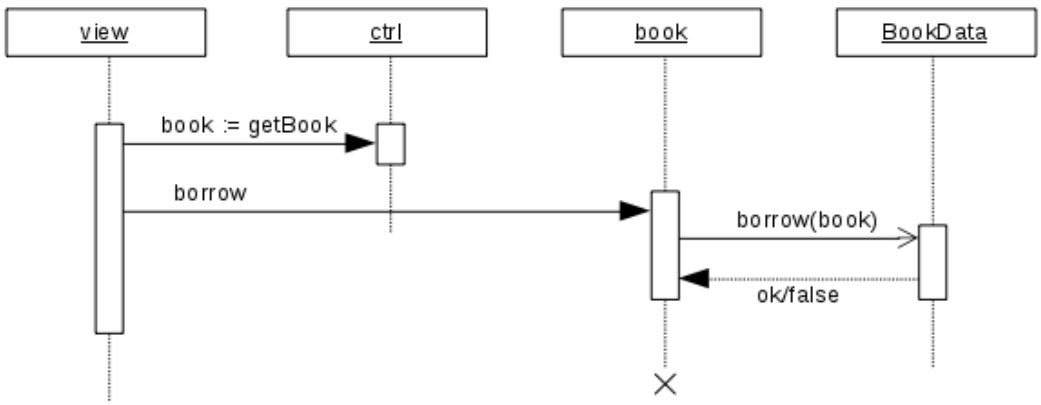


and there is also a syntax to indicate that a package imports a second package and is merged of other packages.

SEQUENCE DIAGRAM

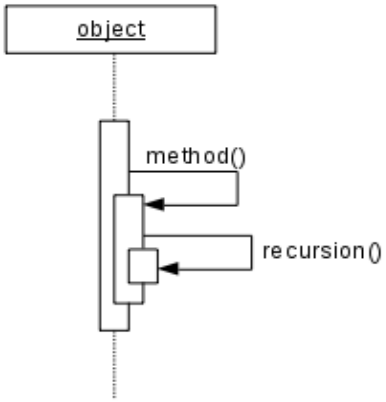
A sequence diagram is an interaction diagram, which shows how the objects interact with each other. The diagram shows several objects drawn horizontally next to each other, and for each object are drawing a line, showing the object's lifeline. Sequence diagrams are suitable to show how the objects are sending messages to communicate with other objects, but they are not suitable for the logic of complex procedures. As an example the sequence diagram below shows which objects have been included in the function to borrow a book. There are four objects:

- *view* is an object that represents the dialog box to a user's borrow of a specific book
- *ctrl* is the controller for this dialog box
- *book* is a model class, that represents the book, to be borrowed
- *BookData* is the repository that represents the table *Book* and the object which stores information about a borrow in the database

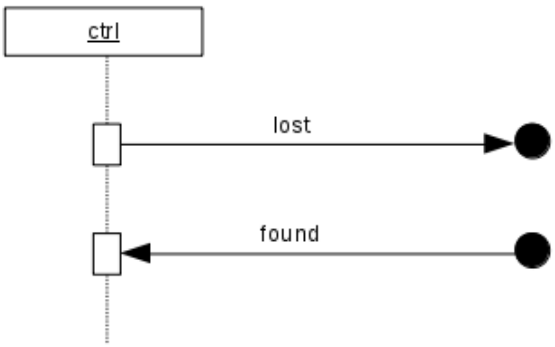


The diagram does not describe exactly how the program works, but is changed a bit to illustrate more about the diagram. The dotted lines represent the objects' life-lines, and the vertical boxes on the lifelines illustrates that the object performs something. When the user wants to borrow a book, the *view* object turn to the dialog's controller to provide a reference to the book to be borrowed. It is drawn as a solid arrow and illustrates a synchronous call of a method in which the calling object waits until a value is returned. When the *view* object get this reference, it calls the *book* object to borrow the book, and it happens again with a synchronous call. The *book* object then calls the *BookData* object to register the borrow, but this time it's with an asynchronous call, as indicated by an open arrow. The *BookData* object can then at some point send a return value back (thar is an event). During the *book* object is drawn a cross. It should illustrate that this object is removed after the function is performed. It does not happen in the program but is included here to show the syntax.

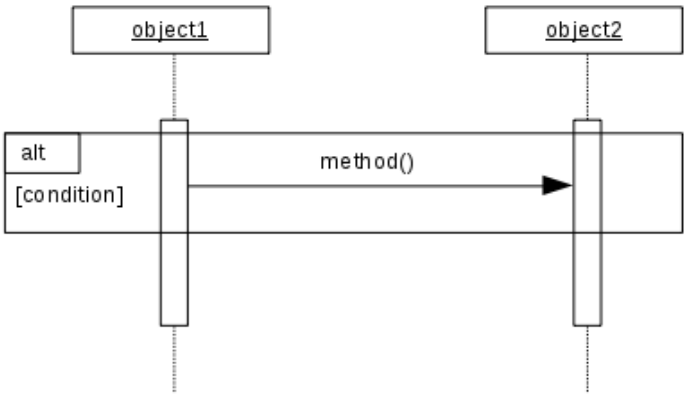
The sequence diagram is relatively complex, and there are actually a number of other options. Below is shown how an object can call a method on itself:



As another example, is below shown how an object may send messages to objects which are not part of the diagram, and the like how to receive messages from other objects:



One talks in this context about lost and found messages. You can also use conditions in a sequence diagram where one or more operations are performed only if a condition is met:

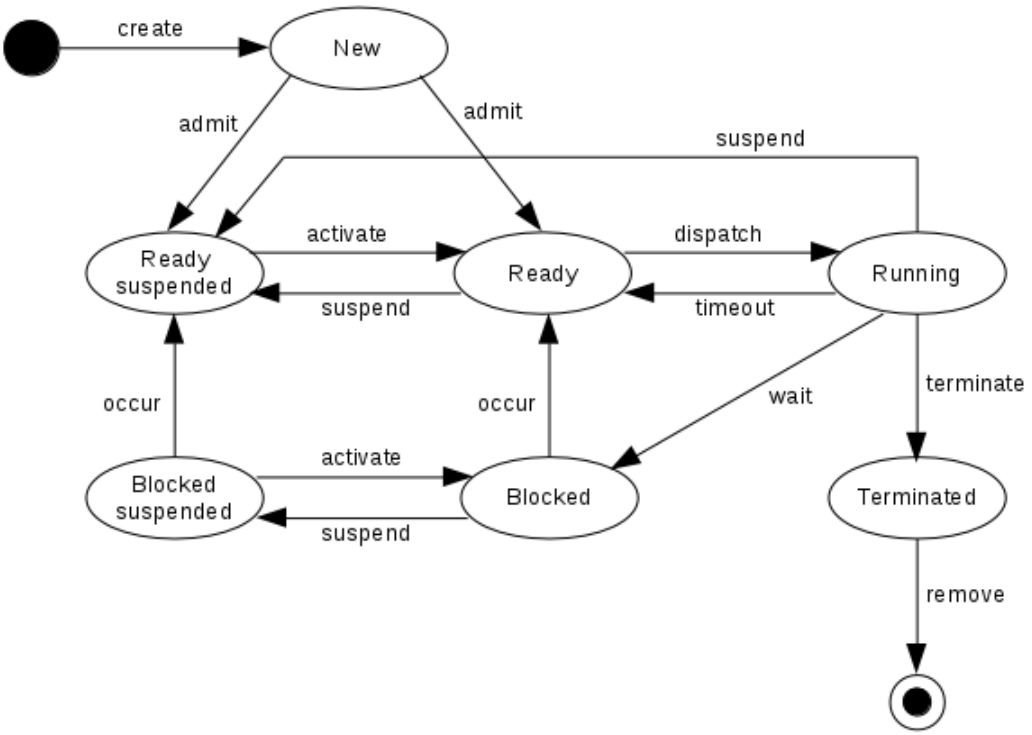


and if you draw a dotted line through the box, you are modeling an *else* part. By replacing the word *alt* with the word *loop*, you has similarly have modeled an iteration.

Sequence diagrams can be excellent to show operations in the control layer, but they also have their limitations. First, you need a good drawing program, otherwise they are difficult to draw. Secondly, the diagram – even with a good drawing program – my be uncluttered if there are many objects, and the diagram tends to grow in both width and height. There is an alternative called a *communication diagram* where the objects need not be drawn on a horizontal line. The syntax is somewhat the same, but the diagram is easier to draw. However, it is not easier to read and quickly becomes unmanageable.

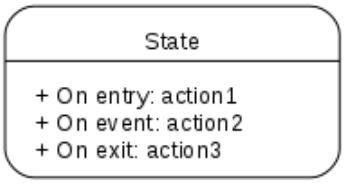
STATE MACHINES

A *state diagram* or a *state machine* is a diagram that models an object, which may be in different states, and wherein the object can change state due to various events that may occur. A classic example is a process for a running program. When you start a program, the operating system will create a process that represents the running application, and the process starts its life in a *new* state. After this the process' life can be modeled in a diagram, as shown below, where the ellipses represents states and the arrows represents state transitions.

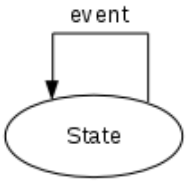


When a process is in the state *new* it can switch to one of two other states: *ready*, which means that the process is ready to run on the CPU (when it has time) or *ready-suspended*, which means that the process is ready but not yet can be performed (for example because it is started as a service or batch job that should run at a later time). If a process is *ready suspended* it can switch to *ready* mode (when the operating system sees the time for it to be ready occurs). A process which is *ready* (is in the ready queue), can again be *ready suspended*, but it can also switch to *running*, which means that the operating system select (dispatches) the process for the CPU. A running process can switch state caused by 4 events. Firstly, there may be a timeout if it has used CPU in the maximum permitted period. The process then becomes *ready* again. It can also be *ready suspended* (if the process for instance executes a *sleep()*) or also i can switch to *blocked*, which means that it is waiting for a resource (typically an IO operation such as read a block on the disk). Finally, it may be, that the process is completed (the program is finished), and the process will then change to a *terminated* state where all allocated resources are released before it is removed. When a blocked process waiting for a resource is ready to proceed (for example because the disk controller signals of that disk operation is performed), then the process can goes back to *ready*. A blocked process can also switch to *blocked suspended* (for example because of a swap), and from here it can either switch back to *blocked* or *ready suspended*.

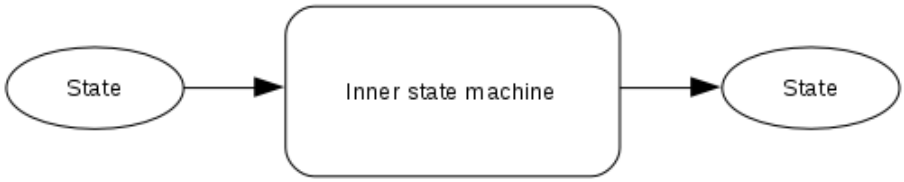
A state machine is particularly suited to modeling the life cycle of objects in the same way as above that periodically changes state, and there are actually many of that kinds of situations in practice. Therefore, it is a useful diagram, and the diagram is typically drawn as shown above, but there are extensions. For example you can for a state specific actions to be performed when the object switches to that state, or when the object changes from the state, and you can even specify actions to perform if the state change is due to a specific event:



Of course an object can change state from a certain state to the same state:



Finally, I mention that a state diagram can be composed of several state diagrams. If you have a large state diagram, it becomes unmanageable and you can then display an inner state machine as follows:



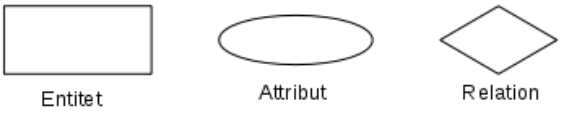
The above is by no means a complete description of UML, but it would show a little about what UML is and can be used for, and some of the diagrams that exists and the main drawing rules. There are a number of tools that can be used to draw different UML diagrams, and applying UML, one should take an interest in these tools and take the time to learn how they works, because the work to draw diagrams otherwise may be too extensive. As an example I can mention the program *Dia*.

APPENDIX B

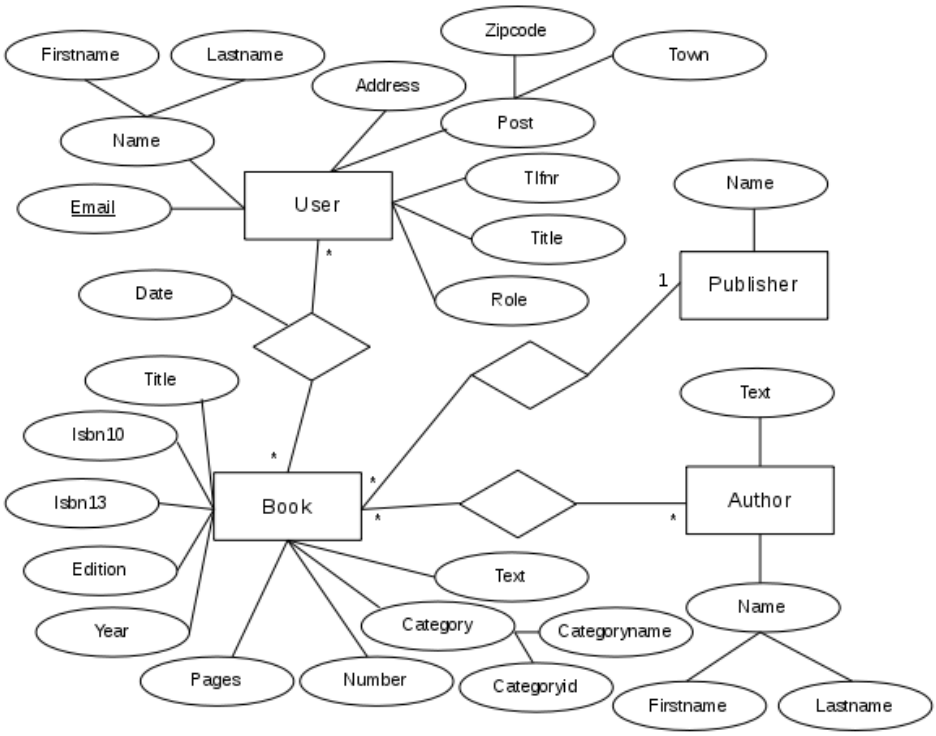
In this appendix, I will see a little on database design. When designing a program, you determines which data has to be stored in the database, and the question is how the database should be designed and including which tables there needs to be. There is not a clear answer, but there are some pretty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

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 a 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 libray 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 a *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.

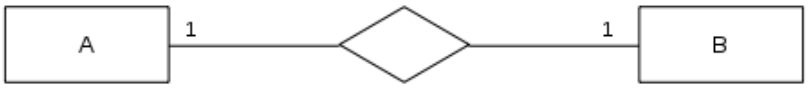
Between the two entities *Book* and *Author* is a relationship that is a many-many relation that appears with an * on both sides. The relation indicates that a book may be related to several authors, corresponding to that a book may have multiple authors. Similarly, an author may be related to several 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 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* you 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 borrowed a book. Since a user may borrow 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 borrowed.

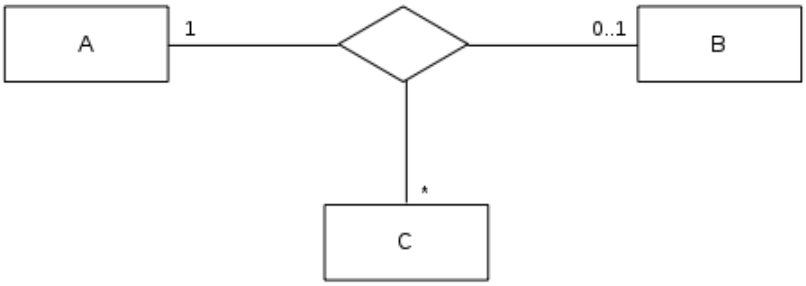
The above is a typical ER diagram, but there are actually several drawing rules. Overall, you can use the same multiplicities as described during the domain model. Specifically, you can have one-one relation:



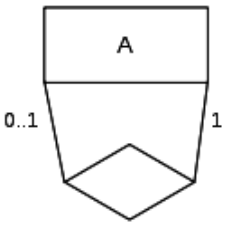
This figure indicates that an entity A must correspond to exactly one entity B, and conversely that an entity B should correspond to exactly one entity A. The relation could also be

indicating that an enitet B must corespond 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:

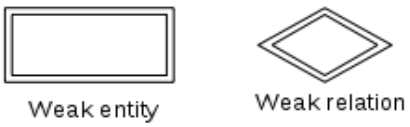


In particular, you should note that an entity can have a relation to itself:

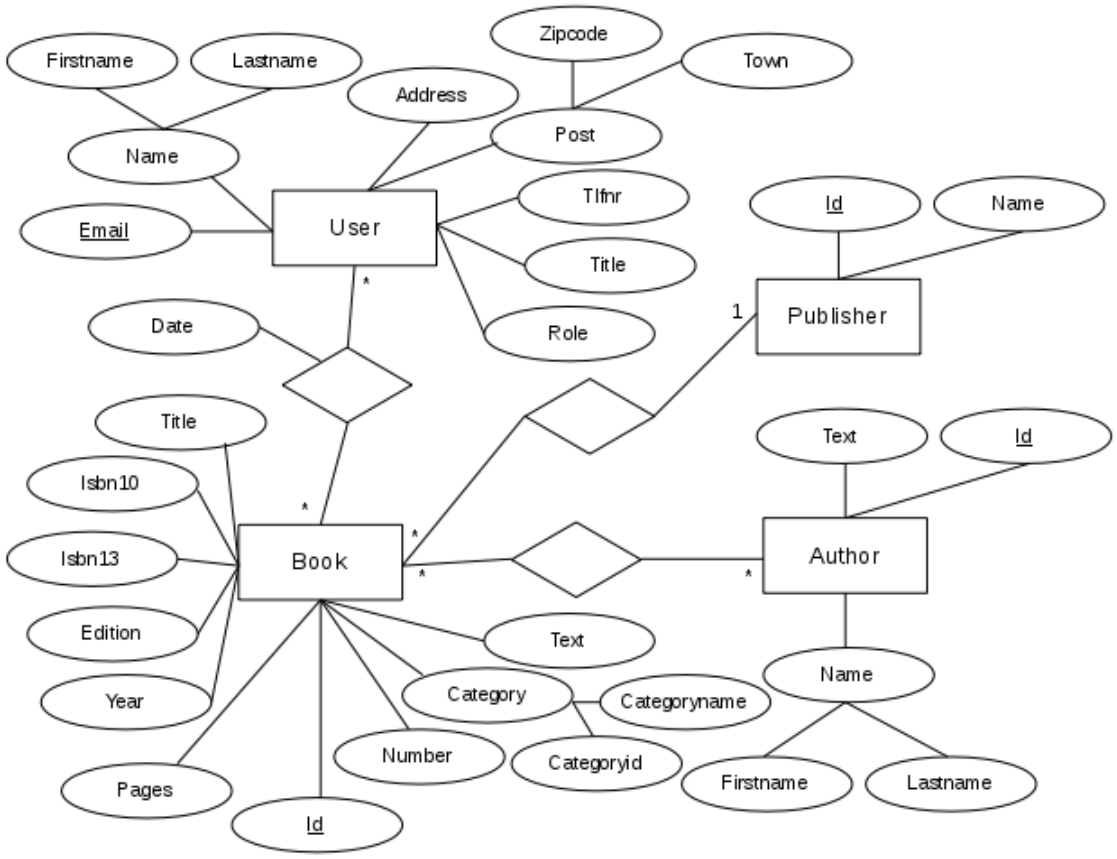


An example would be “son of”.

Finally, there is something called a weak entity, and it are entities, which can not exist unless they are related to another entity. A weak entity is shown 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 waek 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.

MAPPING TO RELATIONAL MODEL

When he 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 imaging 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 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 can not be changed, and when students often change email addresses, it is not suitable for the key, and also email adresses are long strings, that are not especially suitable as keys. The result of the first step is as shown below:

Book

Id	Isbn10	Isbn13	Title	Edition	Year	Pages	Number	Catid	Name
	Text								

Author

Id	Firstname	Lastname	Text

Publisher

Id	Name

User

Id	Emai	Firstname	Lastname	Address	Zipcode	Town	Tlfnr	Title	Role

Step 2

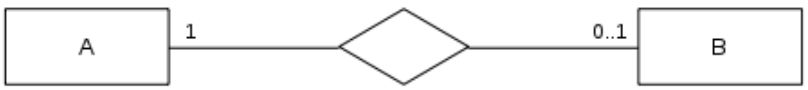
Then all weak entities are mapped in the same way, and the only difference is that each relation, this time will have an additional field which is the primary key from the regular entity that the weak entity is associated with (additional fields if the primary key is composed). The key in the relation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

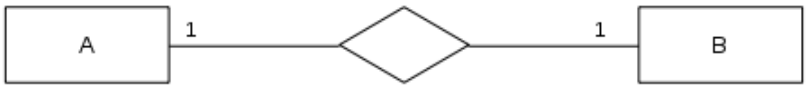
Step 3

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 is talk about following relation

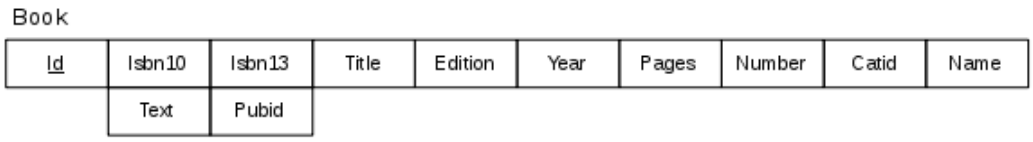


you must choose one of the sides and place the foreign key there (and possibly attributes). Alternatively, you could create an entirely new relation that then must contains the primary keys from both entities that then are a composite primary key in the new relation, and they will also each be foreign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

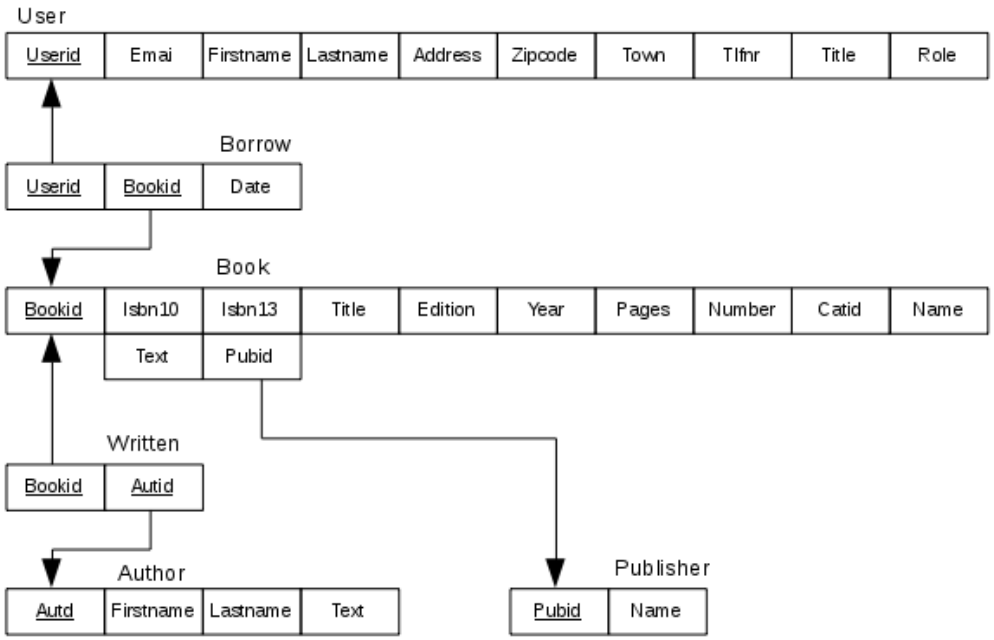
Step 4

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*:



Step 5

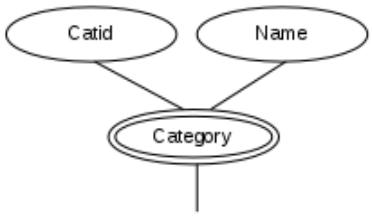
Many-many relations between two entities are always mapped as a new relation that includes the two primary keys from the two entities and possible attributes associated with the relation. The two primary keys is a composite primary key in the new relationship, and they are each foreign keys to the two 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;



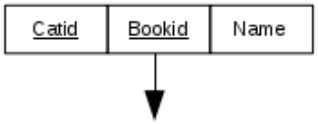
Note that there are two new relations that I have called, respectively *Written* and *Borrow*. The arrows indicate foreign keys. Note that I also renamed some of the attributes.

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:

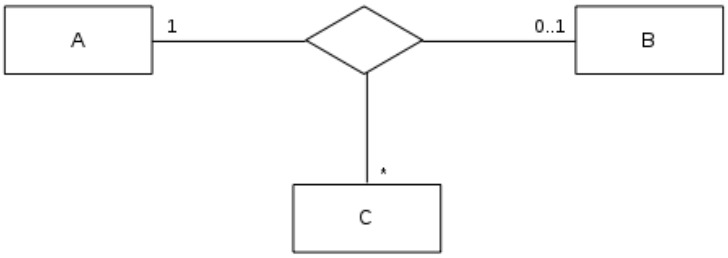


to illustrate that a book could have several categories. If need be, the multivalued attribute must be moved into a new relation, that in addition to the two attributes consists of the primary key from the original relation (*Book*). The new relationship will have a composite primary key, and *Bookid*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

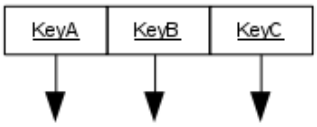


Step 7

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 design is in principle finished, and the outcome was in this case 6 relations, all of which have a primary key and contains no multivaled 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 wee 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.

NORMALIZATION

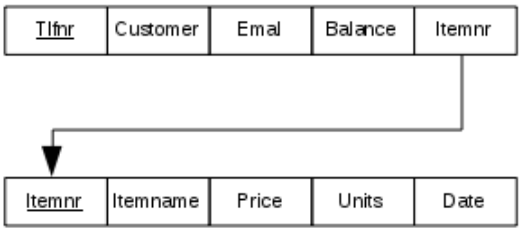
Wee say that a relational model is on first normal form if it consists of relations which has no multivalued attributes, and if all relations have a primary key. If you have completed the above mapping, it will automatically be satisfied, and the above model is then on first normal form. You can also 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 defintion 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, that for customers shows what products they have bought:

Tlfnr	Date	Customer	Emal	Balance	Itemnr	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:

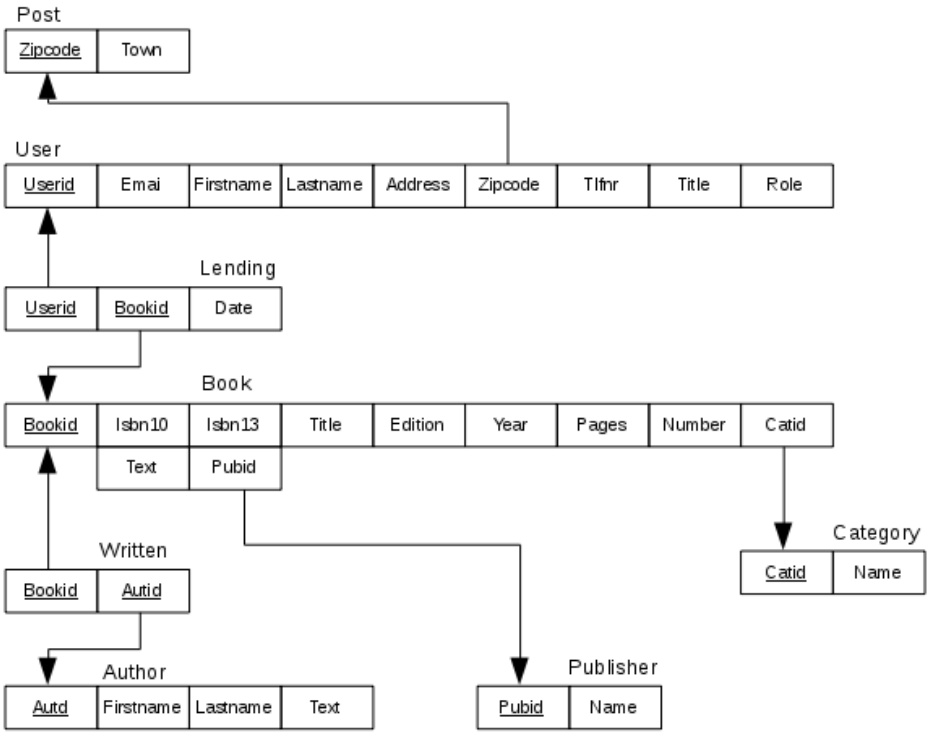


The goal of the second normal form is to ensure that the database tables do not contain data concerning more things, since it can make it more difficult to understand database content.

A relational model is on third normal form, if it is on second normal form and, if for any relationship 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 attributit 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 wee say that there is a transitive dependency. Looking at the current example, it is clear that relations *Publisher*, *Author*, *Written* and *Borrow* all is 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 considers 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:



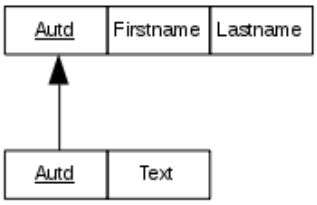
and it consists therefore of 8 relations.

The aim of third normal form is to eliminate redundancy, which means that the same information is recorded at multiple places. For exammple 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 means unnecessary space consumption, but the main reason is that it becomes more difficult to maintain the database, as it may means 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 means 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 needs to change at one place.

OTHER DATABASE IMPROVEMENTS

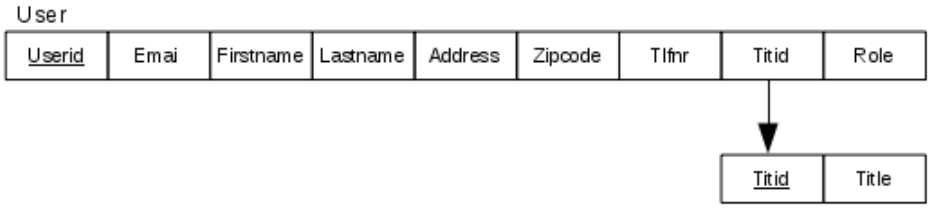
If you look at the normalization process, it will typically entails more tables in the database, and it is in principle nothing wrong with that but it may mean that in requests to databases are required with many JOIN operations that are actually complex to perform by the database management system. 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 denormalisation where one goes the other way and turn tables together, even if it means that the database then contains redundancy. The reason for denormalization 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 his 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 it's own relation with the primary key of the table *Author* as key. The new relation thus has two attributes, where *Autid* is both primary key and foreign key to the table *Author*:



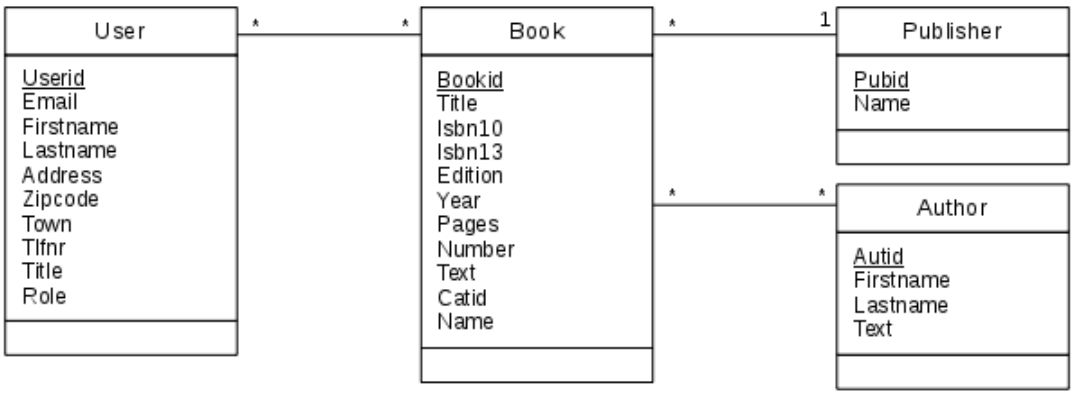
Now, you should not generally move all attributes that can be NULL to their own table, because it can get the number of tables to explode and thus complicate the database unnecessary, but if you have an attribute as *Text*, that very rarely has a value, you should consider to do it.

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.



THE USE OF A CLASS DIAGRAM

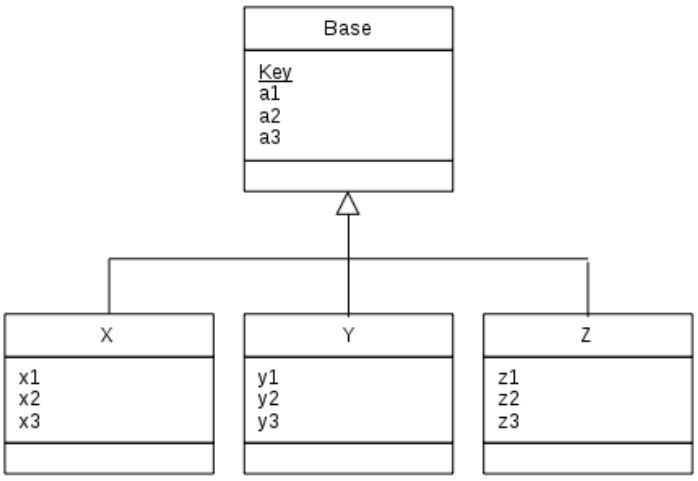
The basis of the above approach to database modeling is an ER diagram, but you might as well start with a class diagram:



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 uses 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 can 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 comes first in action in step 1 in the mapping.

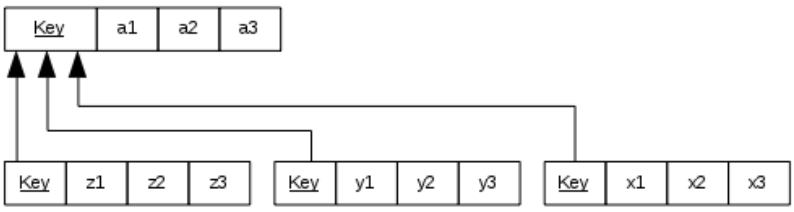
The construction of the database is a design activity, and the starting point will typically be a design of the model layer, and then you perhaps already has a class diagram that can be adjusted as a basis for the database, but what do you do if this model contains inheritance? The relational model does 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 forces 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:



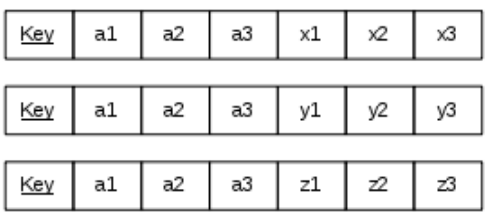
where there of course does not need to be three derived classes, and where each class can have both fewer and more attributes. I will show 4 options for mapping this structure of a relational model.

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 can not 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:

Key	a1	a2	a3	x1	x2	x3	y1	y2	y3	z1	z2	z3	t

This solution requires an additional attribute – referred to herein 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 a 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.

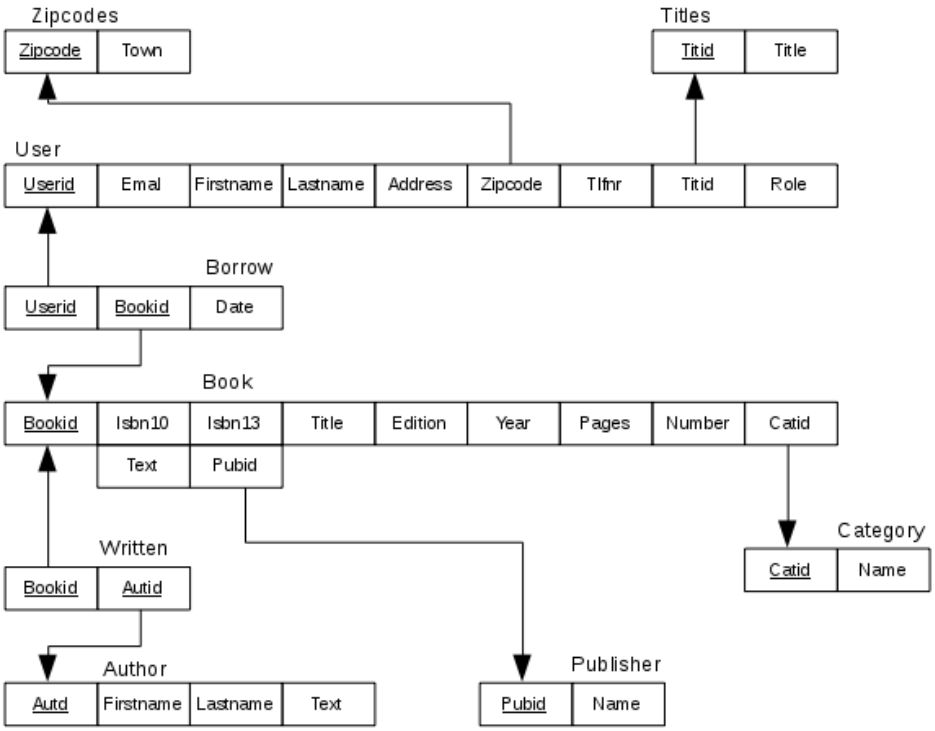
When designing databases it can actually happens that an object can be several things, as for example both an X, Y and Z. If you have such a design, consider the following mapping, which is a variant of the above:

Key	a1	a2	a3	x1	x2	x3	y1	y2	y3	z1	z2	z3	t1	t2	t3

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

CREATE THE DATABASE

After you have modeled a database by the above procedure and has 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. surrogat 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
publisher_name varchar(50) not null,        # the publisher's name
primary key (publisher_id)
);

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

```
create table authors (
 author_id int auto_increment not null,  # surrogat key for an author
 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 titel
 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
  ipcodes_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,      # ths 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)
);
```