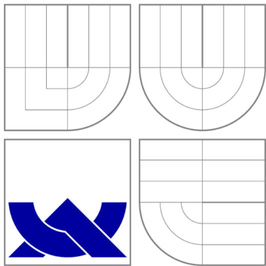
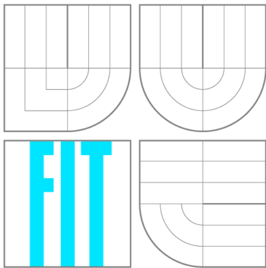



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY


FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS


# INFORMAČNÍ SYSTÉM PRO SPRÁVU LICENCÍ SOFTWARE

INFORMATION SYSTEM FOR SOFTWARE LICENSES

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS


AUTOR PRÁCE — TOMÁŠ ROHOVSKÝ
AUTHOR

VEDOUCÍ PRÁCE — prof. Ing. TOMÁŠ HRUŠKA, CSc.
SUPERVISOR



BRNO 2011

## Abstrakt
Tato práce se zabývá vytvořením informačního systému pro správu softwarových licencí. Tento systém je vytvořen v jazyce PHP s použitím Nette Frameworku. Architektura systému je založená na návrhovém vzoru MVP, který zajišťuje snadnou modifikaci a rozšiřitelnost současného řešení.

Práce také pojednává o technologií licencování softwaru. Konkrétně je zaměřena na licence typu LM-X. Je zde popsáno, jak tyto licence vypadají a jaké existuje současné řešení pro jejich správu. Analýza tohoto řešení byla základem pro návrh tohoto systému.

Systém podporuje uchování a správu objednávek a licencí, vytváření předdefinovaných šablon pro vytváření licencí, generování XML vstupu pro licenční generátor a ukládání samotných licenčních souborů.



## Abstract
This project deals with the creation of an information system for the managing of software licenses. This system is developed in PHP, using the Nette Framework. The system architecture is built on the MVP design pattern, which ensures easy modification and extensibility of the current solution.

The paper also deals with software licensing technology. Specifically, it focuses on the LM-X license. It describes these licenses and the solution for their management. Analysis of this solution was the basis for the design of this system.

The system supports the storage and management of orders and licenses, creation of predefined templates for creating of the licenses, generation of the XML input for license generator, and the storage of license files.



## Klíčová slova
licencování software, licence, LM-X, administrace, PHP, Nette Framework, MVP

## Keywords
software licensing, license, LM-X, administration, PHP, Nette Framework, MVP

## Citace
Tomáš Rohovský: Information System for Software Licenses, bakalářská práce, Brno, FIT VUT v Brně, 2011

# Information System for Software Licenses

## Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana prof. Tomáše Hrušky.

. . . . . . . . . . . . . . . . . . . . . . .
Tomáš Rohovský
May 25, 2011

## Poděkování

Chtěl bych poděkovat mému vedoucímu práce panu prof. Tomášovi Hruškovi. Dále Karlovi Masaříkovi, Zdeňkovi Přikrylovi a Radkovi Burgetovi za jejich odbornou pomoc. V neposlední řade také Marcusovi a Ronaldovi za jejich korektury.

© Tomáš Rohovský, 2011.
*Tato práce vznikla jako školní dílo na Vysokém učení technickém v Brně, Fakultě informačních technologií. Práce je chráněna autorským zákonem a její užití bez udělení oprávnění autorem je nezákonné, s výjimkou zákonem definovaných případů.*

# Contents


**1 Introduction** **3**
1.1 Structure of the Document . . . 3

**2 Goals of the project** **4**

**3 Analysis** **5**
3.1 LM-X . . . 5
3.1.1 License file . . . 6
3.1.2 License template file . . . 7
3.2 X-Formation License Distribution Service . . . 8
3.2.1 User . . . 8
3.2.2 Product . . . 9
3.2.3 Customer . . . 9
3.2.4 Order . . . 10
3.3 Justification for further application . . . 11

**4 Design** **12**
4.1 Database structure . . . 12
4.2 License history . . . 14
4.2.1 Copy of tree . . . 14
4.2.2 Copy of record . . . 15
4.3 Data presentation . . . 16
4.4 System logic . . . 17
4.4.1 Administrator . . . 17
4.4.2 Customer . . . 17

**5 Implementation** **19**
5.1 Technologies . . . 19
5.2 Nette Framework . . . 20
5.2.1 MVP . . . 20
5.3 Generally about models . . . 22
5.4 Specific models . . . 24
5.4.1 Order Manager . . . 24
5.4.2 License Manager . . . 24
5.4.3 Managers representing license properties . . . 24
5.5 Views . . . 25
5.6 Presenters . . . 26
5.6.1 Getting of models . . . 27

5.6.2 Actions . . . . . . . . . . 27
5.6.3 Data listing . . . . . . . . . . 28
5.6.4 Forms . . . . . . . . . . 29
5.6.5 Download of file . . . . . . . . . . 31

**6 Deploying of the application** **32**

**7 Conclusion** **33**
7.1 Future extensions . . . . . . . . . . 33
7.2 Personal benefit . . . . . . . . . . 33

# Chapter 1

# Introduction

Nowadays, software is needed to protect against unauthorized use, which in the case of commercial applications, could mean financial losses. For this purpose, there are software licenses, which are legal instruments that allows the use or redistribution of the software, which is protected by law.

Since the legal protection is no longer sufficient, it is necessary to use techniques that help prevent unauthorized use of our applications. Such a technique is protection by the license files.

Since applications are commonly used by several users, it is thus necessary to create multiple license files. This is where our problem arises: the management and storage of the license files. This problem is solved by the output of this project as an information system that will allow it.

This project is prepared in collaboration with a research group - Lissom, which works at the Department of Information Systems, Faculty of Information Technology at Brno University of Technology. The group is dealing with the development of environment for design of Application Specific Instruction-set Processors (ASIP) or a Multiprocessor System on Chip (MPSoC) [1]. The software is produced by Lissom, and sold by Codasip company. Information system for licensing is determined by the latter.

## 1.1 Structure of the Document

In Chapter 2 we set the goals of the project. There are described functionalities, which will be provided to users of this system.

In Chapter 3 performs analyses of licenses that are used for product licensing. An available solution is also discussed and analyzed. At the end of this Chapter, this report will justify the creation of a new solution.

The knowledge gained by analysis is used to create the system design, as described in Chapter 4.

In Chapter 5 describes the implementation of the created solution. Also mentioned are technologies used and important parts of implementation.

The output of this project is the information system, namely server application. It is elaborated in Chapter 6, of how the application is to be deployed on the server.

In Chapter 7 are discussed reached results, as well as future extensions of this project.

# Chapter 2

# Goals of the project

Generally the goal is to create an information system for license administration. Specifically, it will be a module for an existing information system on http://www.codasip.eu/, which will provide an interface for vendors and, in the near future, also for customers to negotiate software licenses.

The system for administration of licenses can be complicated since it can contain many functionalities. A good example of such a complex system is the X-Formation system License Distribution Service 3.2. The aim of this project is not to create a solution of this type. The aim is to create a solution that will meet the particular needs of the company Codasip.

However, it will not meet all of Codasip's requirements; it will instead be the foundation for an application that will be extended over time. It is therefore important that this project focuses on scalability and reusability.

Functionalities that are crucial for the system and thus must be implemented are as listed:

- storing and managing of orders for licenses,
- storing and managing of licenses,
- creation of licenses from predefined templates,
- generation of license templates for the generation of licenses.

Other functions which should be included and supported by the system are as listed:

- storing of license history,
- customer interface for creating orders and retrieving licenses.

In the best of scenarios the module created should be independent and achieve modularity, such that it can be easily used in another system.

# Chapter 3

# Analysis

## 3.1 LM-X

There are three ways to license software:

- Trial – no license file is required
- Standalone (node-locked) – a license file is installed on, and often locked to, an individual machine
- Network (floating) – a license server is deployed at site and a license file is installed on that license server

Products of Codasip are licensed by LM-X licenses [9]. LM-X consists of a client library, which is embedded in the licensed application and a license generator for creating license files. A license file contains text that defines the license agreement between customer and vendor. For example, the license defines whether the application is node-locked or floating, whether it can run on any computer or only on a specific one, whether the license will expire on a specific date, and so on.

Figure 3.1 describes how the node-locked license works. Client checks for a license is stored on the user's machine and thereafter, the user can use the application.

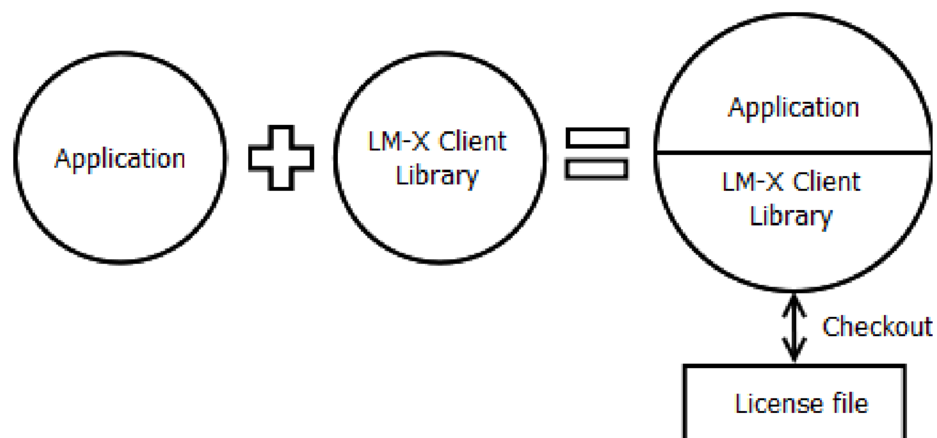



Figure 3.1: Node-locked licensing

The principle behind floating licensing is shown on figure 3.2. It is a client-server architecture based on TCP/IP protocol. The client will check for a valid license file stored on the license server before granting permission to run the application. Periodic heartbeats occur between the license server and protected applications.

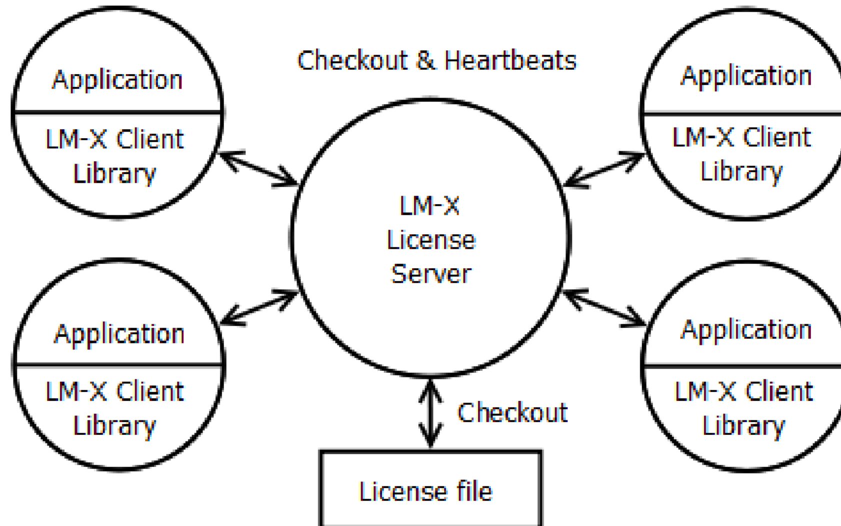



Figure 3.2: Floating licensing

### 3.1.1 License file

The license file is a simple text file, usually with extension `lic`. It contains features described in section 3.1.2. Each feature in a license has its own key, where all security information is encrypted.

The license is generated by the license generator, as is shown on figure 3.3. The license generator program is called xmllicgen. The input of the generator is the license template.

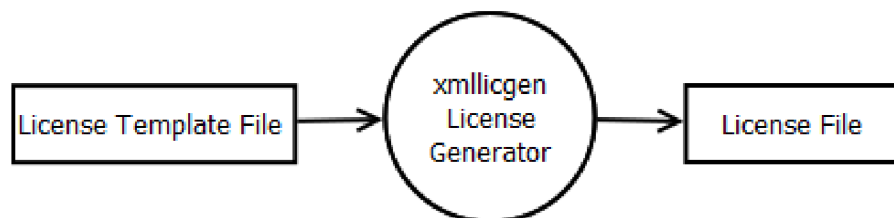



Figure 3.3: License generation

The features' keys are generated along with the license file. The key contains, for example, hostids that are feature locked and cannot be shown to the user.

With regards to this project it is especially important to understand how license templates look like, since its creation and its storing are some of the functionalities of the system.

### 3.1.2 License template file

The license template is an XML file. It consists of a LICENSEFILE tag which can optionally contain attributes CONFIGFILE, OUTPUTFILE and COMMENT. The first is for the configuration of the license generator, the second for the name of the output file and the third for a comment.

The LICENSEFILE tag encloses one or more FEATURE tags. The features can correspond to specific modules or features of the licensed product. Each FEATURE tag consists of a number of SETTING tags that specify the individual attributes for the feature. Settings are different for node-locked and for floating licenses, but some of them are common, such as END specifying of expiration date, MAJOR_VERSION and MINOR_VERSION for restriction of the feature version and so on. For example, setting COUNT is only for floating licenses. It specifies the number of licenses that can be issued simultaneously for the particular feature floating on the network. A complete list of the features is in [10].

The FEATURE tag can also contain one or more CLIENT_HOSTID or SERVER_HOSTID tags, which lock the license to either a client machine or a license server host, respectively. For each machine, you can lock the feature to multiple items (ethernet hostid, IP address, etc.). The license template can, for example, look like following:

```
<LICENSEFILE CONFIGFILE="myconfig.lmx" OUTPUTFILE="license.lic">
<LICENSEFILE COMMENT="comment of license template"/>
    <FEATURE NAME= "f2">
        <SETTING START="2011-05-08"/>
        <SETTING END="2012-05-08"/>
        <CLIENT_HOSTID>
            <SETTING ETHERNET="CAFEBABE01ABC9E5"/>
        </CLIENT_HOSTID>
        <CLIENT_HOSTID>
            <SETTING CUSTOM="CAFEBABE01ABC9E5"/>
        </CLIENT_HOSTID>
    </FEATURE>
    <FEATURE NAME= "f1">
        <SETTING END="2020-01-01"/>
        <SETTING MAJOR_VERSION="1"/>
        <SETTING MINOR_VERSION="5"/>
        <SERVER_HOSTID>
            <SETTING ETHERNET="C8A414C201A666DD"/>
            <SETTING CUSTOM="C8A414C201A666DD"/>
        </SERVER_HOSTID>
    </FEATURE>
</LICENSEFILE>
```

## 3.2 X-Formation License Distribution Service

License Distribution Service, or LDS, is a system for license fulfillment. It is a SOAP (Simple Object Access Protocol) web service [8]. It provides many functions, as follow:

- eliminating of waiting for receiving customer's hostid for licenses,
- enabling bulk license activations,
- interaction with resellers throughout the world,
- allowing users to reclaim lost licenses,
- reducing the time required to obtain license keys,
- and many more.

License Distribution Service can be run only in Windows-based systems and requires Microsoft SQL Server 2005 or 2008 and Microsoft SQL Server Management Studio Express. The LDS can be implemented to company information systems, or be run on the server, where it will manage everything by GUI application X-Formation License Service Administration. In this chapter we will discuss LDS on GUI application shown in figure 3.4.

The structure of the License Distribution Service is based on a tree hierarchy of Entities. Entities can represent countries, regions, companies, departments, etc. The semantics can be decided by the user. At the top of the tree is the root Entity. Each Entity can contain other Entities. Users in the tree have access to objects in their sub-tree. The exception is the products, which can be situated just under the root Entity. All Users have access to them.

There are four types of objects that can be stored for each Entity, as described in following table:

| **Object** | **Description** |
|---|---|
| Products | Products offered to Customers |
| Customers | Customers for which licenses are generated |
| Orders | Orders of licenses for products |
| Users | Users of license distribution system |

Table 3.1: Table of LDS objects

The system provides adding of Entities, Users, Products, Customers and Orders. With the exception of the root Entity, it is possible to remove all objects and Entities. However, this has to be done with regard to dependence. Moreover the GUI application provides searching in fields of Entities or objects, and filtering by object type.

For a clearer picture, the database schema of system is shown in figure 3.5. In Customers, Orders and Products you can see attributes called Data. Those are XML attributes, because they are used in SOAP messages during communication with the server.

### 3.2.1 User

User properties include name, entity, user type and permissions. It is possible to enable or disable User. User has access to objects in his sub tree.

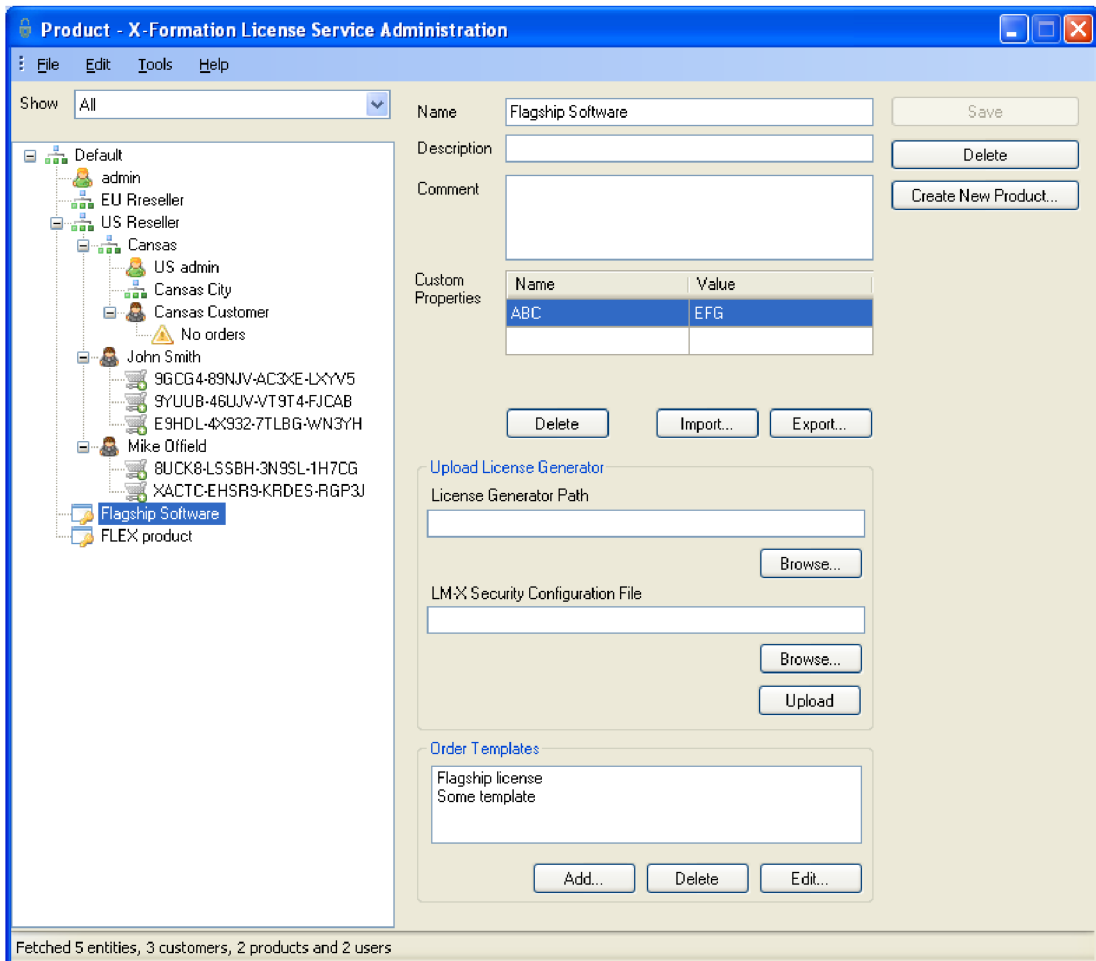



Figure 3.4: License Service Administration

### 3.2.2 Product

The following items can be set: name, description, comment, custom properties, license generator type, license generator and Order template. What is interesting on Product is the setting of the license generator, which will be used for the generation of license file. This setting contains the definition of the generator path. It is possible for one to choose between LM-X, Flex and custom generator. In the case of the LM-X generator, it contains also the definition of the configuration file path.

To expedite the operation, the user can create an Order template. Using templates, one can set up Orders associated with particular Customers and Products to automatically include certain features. These features include: activation start time, activation end time, number of licenses that can be generated for order, name, description, custom properties, list of hostid lists and feature list.

### 3.2.3 Customer

The properties of Customer include name, company, department, email, job title, entity, custom properties and comment. Although only the name is required, it is very useful to

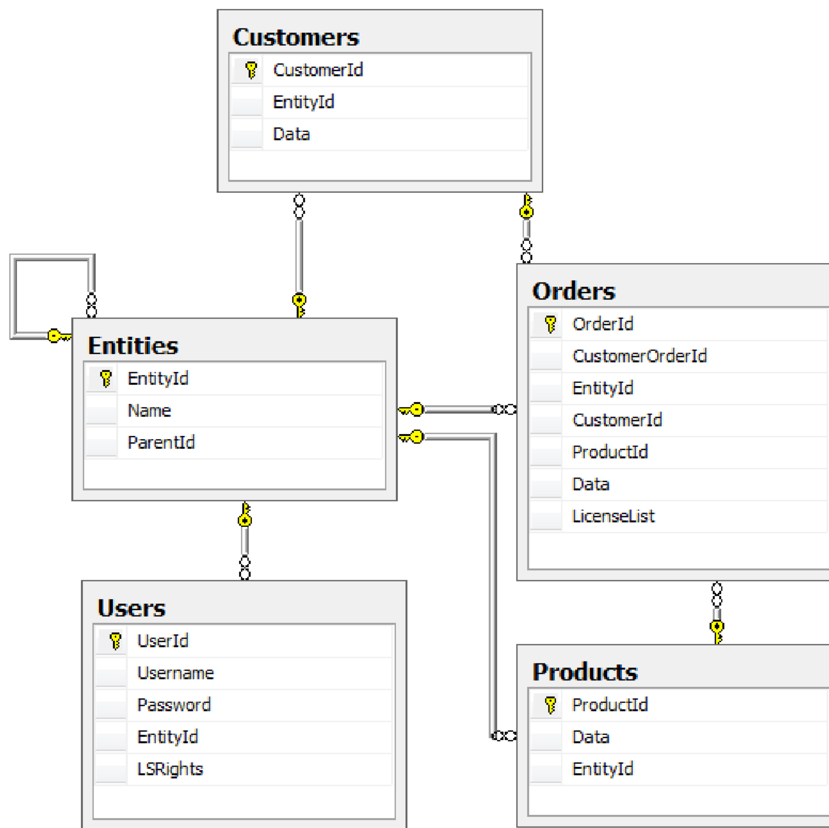



Figure 3.5: License Distribution Service database schema

provide one's email, because it can be used for sending Order by email.

### 3.2.4 Order

You can create empty Order or Order from a pre-defined template. The parameters are Order Id (automatically generated), entity, customer, product, activation start time, activation end time, license count, description, custom properties and features. An Order can be in the default inactivated state, i.e. without an attached license. It can also be in an activated state, where one or more licenses have been attached.

If the customer's email is specified, an Order can be sent via email message. One can also manage licenses. This means one can generate new licenses, provided that one has already defined the generator in Product. One can also view license hostids, view license text and delete or clear license.

Clearing a license removes only the license text and clears the activation date instead of deleting the entire license. Hostids remain the same.

If in the Order it is defined that the license is locked to some hostids, these hostids have to be filled before the license is generated. Typically the Order has all hostid values set to empty. It is interesting to note that hostids are not locked to the feature, but are locked directly to the license. According to the LM-X specification, the hostids are related to each feature [9].

## 3.3 Justification for further application

As written in section 3.2 there are existing solutions. Here a question arises: why do we not use this existing solution? It provides almost all functionalities which we require, and also much more than that.

At first glance the followingt appears to be the simplest solution. Deploy a web service to the server and manage everything with GUI. Although this sounds simple, the GUI application is not completely suitable. For example, it lacks the showing of all licenses and filtering according to some criteria. It is also not possible to store license history and other required functionalities. In short, the GUI is unsatisfactory.

Other solution, something more complicated, will be the implementation of License Distribution Service API to our system which thereafter allows us to adapt everything to our needs. This option seems like the solution to our problems. However, this is not true.

The main reasons for not doing so, and also why this project exists, are that this software requires payment, and that the time for use is limited. Moreover, with respect to our case, the limit of the number of managed licenses stands at only 100. Another important reason for not using the X-Formation solution is that it requires Microsoft products, and that it is expensive. Nonetheless, the analysis of the X-Formation solution was useful, because its architecture and principles will be the inspiration for the design of our new solution.

# Chapter 4

# Design

## 4.1 Database structure

The database schema was designed based on the analysis of the template license format carried out in section 3.1 and the analysis of LDS carried out in section 3.2. The database schema can be seen in figure 4.1.

The hierarchy concept of users and customers through the use of entities, which is used in the LDS, is very convenient. There is visibility of objects located in the subtree of user, and a logical segmentation of objects in the system. Therefore, this concept is used in our module.

The table `users` from the existing system was used in the new module. The only difference is that it is a subordinate table to the table `entities`. In our system, unlike the LDS, there is no separate table containing the customers. Customers and users are kept together in the table `users`. They are distinguished by the roles. User roles are already in the system database structure and therefore are not shown in the diagram.

The table `product` is also a subordinate table to the table `entities`. Since the products are always subordinated only to the root entity, this relationship may seem unnecessary. It was included to prepare the system for the event that products be offered, in the future, only in the particular subtree (for example, region). Another reason was that it is possible to simply display the tree with the hierarchy of objects in the system.

One of the requirements for the system was to provide for the creation of licenses by the predefined templates. Therefore the table templates were created. It is a subordinate table to the table `product`, because templates are related to products.

Next is the most crucial portion of the project. In the database schema are tables orders and licenses. Except for a few differences, these tables are the same. Table orders represent the orders. Each order can contain multiple licenses, which are items in the table licenses. Table `licenses` contains, among other things, columns `xml_file` and `license_file`. The first is for the content of the generated license template file, while the second is for the content of the license file. Besides that, the license template is stored as text in XML format, and is also stored in the database. It is like that because it is easily editable since there is no need to access the XML file.

Another solution, other than storing license template files and license files in the database, is storing it on the server's disk. This is useful when we want to store large files. These files are usually not too large. It was therefore decided that they be saved to the database. Moreover, we achieved by that centralization of storage.

In the database schema there are also tables `features`, `hostid_lists` and `hostid`.

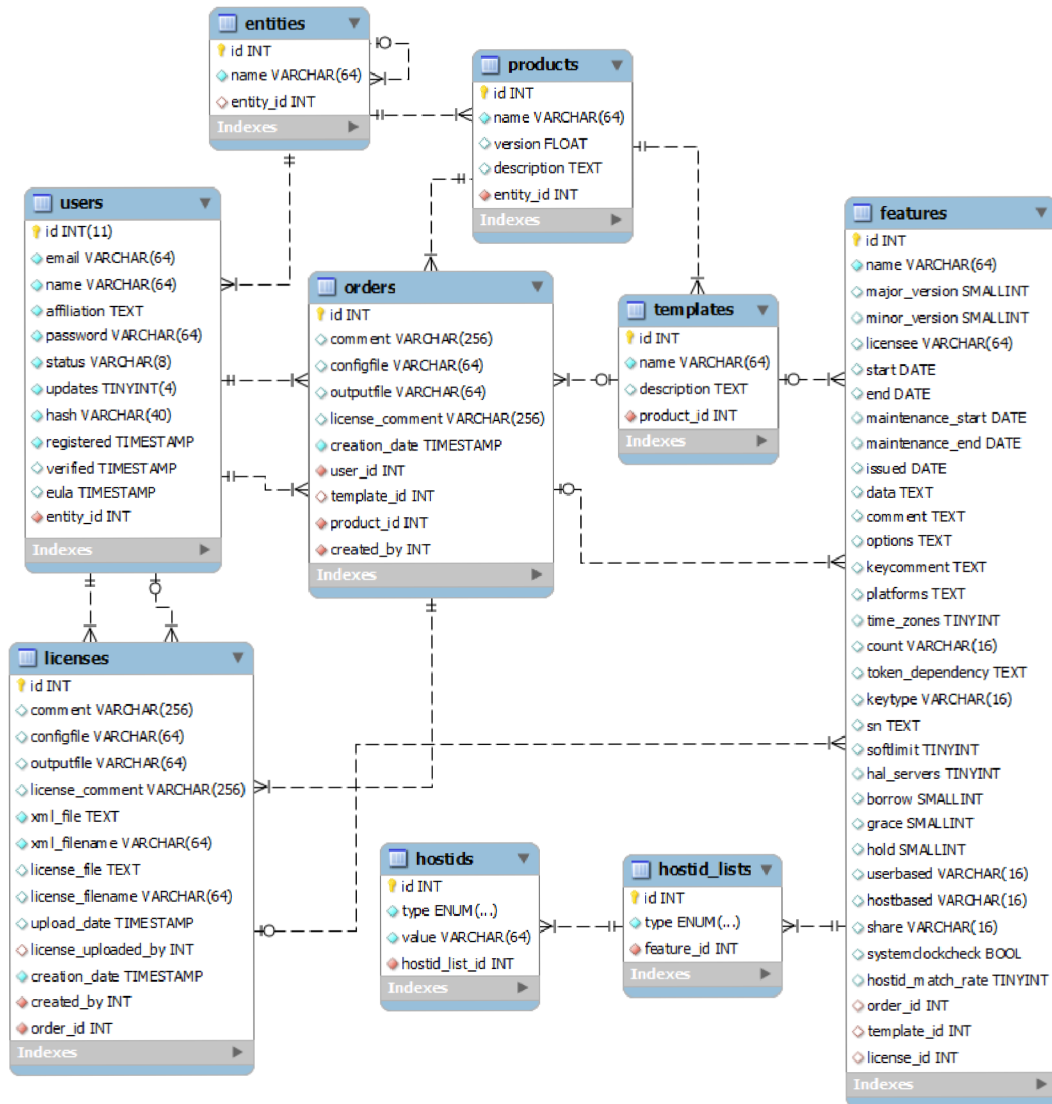



Figure 4.1: Design of database schema

They are nothing more than transformations of the XML structure of the LM-X templates for license generating to database schema. Since there are two types of licenses, server and local, there was initially some consideration about distinguishing the two on the database level. Without such distinctions, the user has to deal with a large number of settings. However, this is not the main reason for distinguishing the licenses on the database level, because it can be done with well-defined templates that allows the user to bypass the setting of features. This consideration, however, was determined to be false for another reason. It would prevent, for example, a simple change of a node-locked license to the floating license, and vice versa.

## 4.2 License history

As one of the requirements for functionality is storing of the history of licenses, it is necessary to consider how to implement it. Although implementation of this functionality is not necessary for this project, but it would be design how to solve that. In the event of manipulation of the data composed the license, the history of licenses should be preserved. Manipulating means adding, editing or deletion of data.

The licenses are stored in the table `licenses`. Their parts are stored in `features`, `hostid_lists` and `hostid` table. Therefore the license consists of a tree of records.

If the record in the table `licenses` is manipulated, it has no impact for form of the license file. Attributes that have impact for form of license template file or license file, are the configuration file, output file and commentary. The first two are not passed to the output license file and the third is used just for comment in the license file, whose presence in the license file is irrelevant. Therefore, when is manipulated with record of table `licenses`, the history of the license does not has to be preserved.

In the following two subsections, ways on how to solve history of the licenses are presented.

### 4.2.1 Copy of tree

When manipulating the records from one of the tables from `features`, `hostid_list` or `hostids`, the original records have to be preserved. The solution can be complete copy of the records forming the license. The original license would be a parent of the new license (ie, copied).

For example, if the action for deleting of hostid is triggered, the hostid is not actually deleted, but that creates a copy of the entire tree with the exception of record of the hostid. Record of when it was manipulated with the license, would be stored as the date of creation of the license. In essence, it goes about the creation date of the license.

The scheme of the corresponding database tables with the necessary data to support the license history is shown in figure 4.2. The data of which database will have to be included for supporting the history, is identifier of subsidiary license `child`. We can afford to use child, because it is not a classic tree structure, and element has only one child.

It is possible to reverse a semantic. So that table `licenses` would include an identifier of the parental license, i.e. `parent` attribute. In this case, it would be difficult to determine whether a record represents the history or current license. Apart from keeping child, where we simply distinguishing the history and current licenses by that the `child` is empty.

In addition, the table contains the `creation_date`, which handles the date with when the license was manipulated.

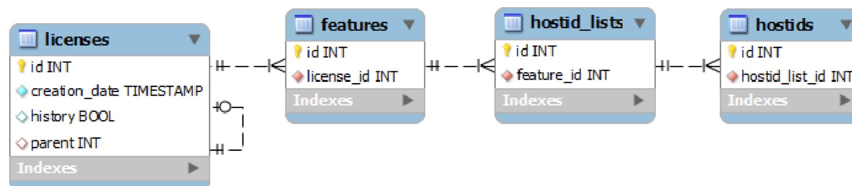



Figure 4.2: Database schema of license with history implemented by copying of tree

Since not all databases support simply queries over the tree hierarchies (used MySQL is one of them [4]), it would be good to optimize it by the string attribute which would contain a consecution of license history. For example, identifiers of licenses are divided by a mark.

Another solution to the problem of the lack of support of tree based queries would be to use inverted semantic of hierarchy – parent record as the attribute `parent`. New version of the license would contain as the attribute identifier of the first occurrence of the license. Which would remove the tree and the hierarchy would be constructed on the date of creation. In such a case, it would be also for easy recognition whether the record is for history, and to introduce boolean attribute `history`.

### 4.2.2 Copy of record

The disadvantage of this solution is the copying of the entire tree of records - a relatively large amount of duplicate data. Solution of this problem would be to only create a copy of a particular record, which is to be handled.

The database structure would look like what is shown in figure 4.3. Here, history is kept in the individual tables `features`, `hostid_lists` and `hostids`. As the tables are very similar to `licenses` table from the solution, which is based on the copying of the entire tree structure. It includes an attribute `child`. Unlike copies of the tree solution, tables have to include `deletion_date` attribute, which indicates when the record was deleted. Selection of records from records for obtaining the current form of license, would be processed by selecting all the elements that do not have attribute `child` and not set attribute `creation_date`.

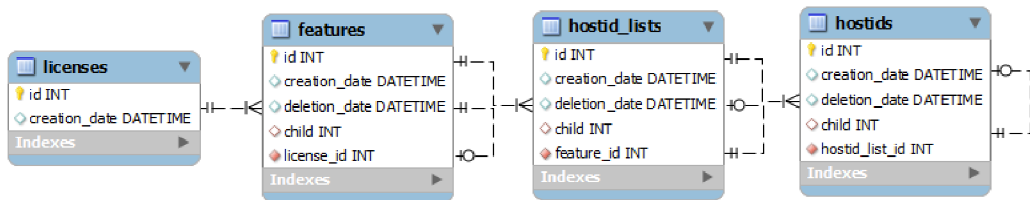



Figure 4.3: Database schema of license with history implemented by copying of record

We will demonstrate the principle on the same example – deletion of the hostid. If the action is invoked to delete the record hostid, it is not actually deleted, but only set the record deletion date of record. Alternatively, it may even set a child of the element to itself, in order to simplify the selection of co current state of license.

Deletion is not the best demonstration of functionality in this case. It is better to demonstrate the principle by editing the list of hostid. If the list of hostid is edited, a copy is created and the attribute `child` is set to the value of child identifier of the new record. It is important that the elements associated with the original record will be updated – assigned to the new record.

This preservation of licenses' history is better in terms of memory consumption, as it is now less tedious than copying the entire tree structure. However, this is at the cost of time, as there will be more queries for the obtaining of history. These queries will be also more complicated. This second way of storing history is more suitable for this project, because browsing of the history is rare.

Both designs can keep information of when the license was modified, what was modified and how it was modified. However, it would be better to add who modified the records of the tables.

## 4.3 Data presentation

Access to data in the LDS is mediated through a tree of objects. This approach is quite useful for the user to have an overview of the hierarchy of objects. However, it is not suitable if many objects are contained in the system. Furthermore, this approach does not allow for bulk actions with the objects, which is a big flaw especially in the case of the licenses. This approach is suitable only to a certain extent and therefore its presence can be considered secondary.

As the primary approach, the following was designed: show a list of all items and allow to restrict the list by using of the filters. The consequence of this approach is the ability to process data in bulk. For example, it is very important for the administration of licenses, whether they delete them, blacklisting, etc.

## 4.4 System logic

The module will use two types of users, so in other words, it should contain two roles. The first is an administrator and the other is a customer. The system on which this module is built supports dividing user roles, and future plans are to divide the functionality of the system to more than one role, particularly with regard to user administration. As an example, the role of an administrator who has assigned rights, will only process orders and generate the licenses for them, and so on. Generally, in design we will keep the basic distinction between the customer and the administrator.

### 4.4.1 Administrator

As a result of the design of the database schema and how present data was created, a use case diagram for administration part shown in 4.4. The use case diagram contains functionalities related to the objects, which are based on the corresponding database tables. These objects are Entities, Products, Users, Orders and Licences.

It may seem that there are no other objects contained based on database tables. However, these objects are not included, but transactions over them are hidden in the particular use cases of the object. So when we speak about product object, we speak about more tables than just `product` table. The main functionality is contained in editing and addition. Worth mentioning is that editing is not just for editing, but also for showing of object's properties.

On the left side from administrator are functionalities, of which is the system is based. These are actions with entities, orders and users. On the right side are the core functionalities, which are core for license administration. These are actions with orders and licenses.

Addition (or creation) of license is shown as a separate action. In fact it will be subaction of editing of orders, since licenses will be created from the orders. However for its importance, it was shown as a separate action. Hostids assigned to order can be empty, but in licenses, they have to be filled. When creating a license from the order, the administrator is prompted to fill up the hostids.

Other action related to the license is the downloading of XML file, for generation of the license file. When the administrator generates a license file, he can upload it to the appropriate license. Last action of license is the showing of license history.

A final note worth taking of, is the action is showing the tree structure of objects, but as was written in section 4.3 it is just secondary functionality.

### 4.4.2 Customer

The customer has access to actions with objects. However, as compared to the administrator, these actions are greatly reduced. The most important functionality provided includes downloading a file that is uploaded. There is also a need to pause over the fact that it is appropriate to inform the customer to upload the license file, for example, through automatically generated email.

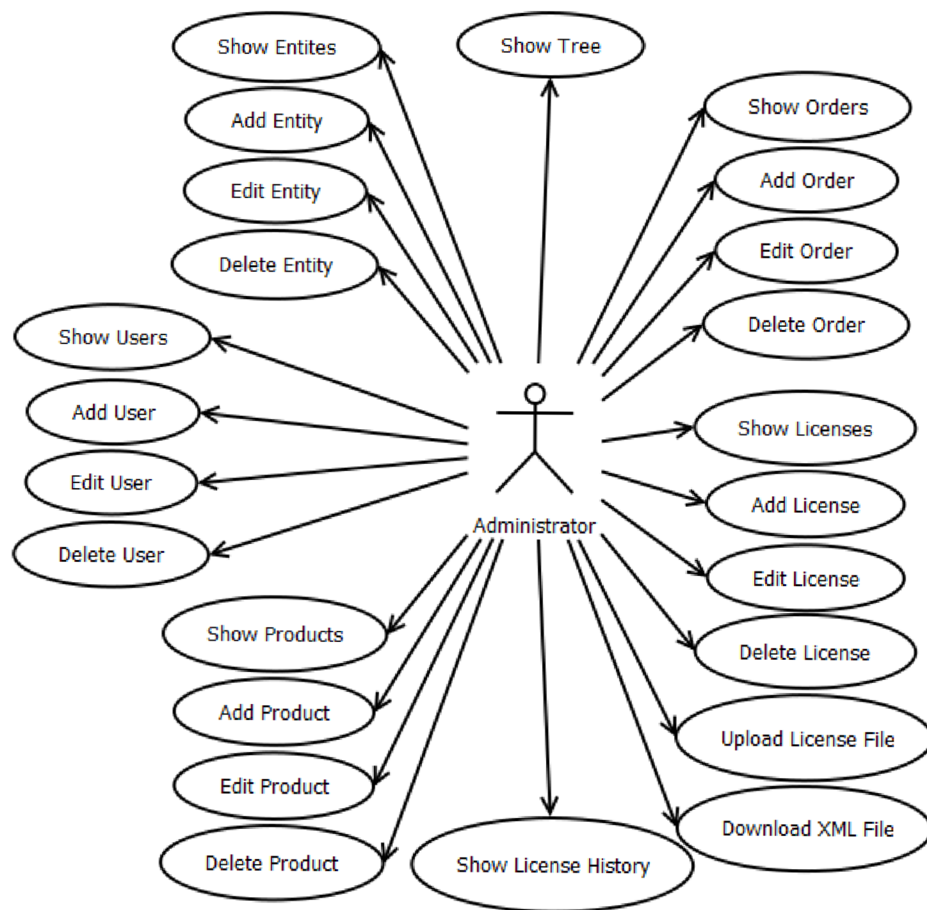



Figure 4.4: Use case diagram of administrator

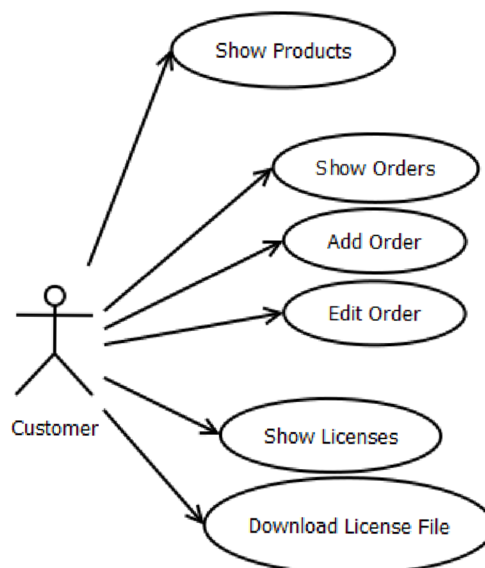



Figure 4.5: Use case diagram of user

# Chapter 5

# Implementation

## 5.1 Technologies

Since this project is implemented to an existing information system, technologies were predetermined. Nonetheless it is important to mention them.

The database is implemented in MySQL (tested and implemented in version 5.1.41). It is an open-source, fast and multi-platform relational database management system.

The application logic implemented in scripting is PHP 5.3.1. Due to the fact that PHP is server language, it was necessary to use a server. Apache 2.0. was used during implementation.

PHP is an imperative object oriented language. It is dynamic-typed, i.e. the data type of the variable is determined at the time of assigning a value. Another important feature is that the arrays are heterogeneous; the same array can contain items of different data types.

An object-oriented paradigm was also used in this project. Since PHP gives the programmer considerable flexibility, it is very easy to create a code leading to confusion or mistakes, as is the case in any language. For this reason, some rules were settled for supporting the purity of design. An example of this was the defining of every class either as abstract or final, and nothing in between.

Nowadays, implementing the information system in pure PHP without any framework is unthinkable. Therefore, for this system, Nette Framework was employed. More about them is written in following section.

Another technology which was used is JavaScript framework jQuery, which was mainly used as part of used components. When we speak about JavaScript it is important to mention that AJAX, a Javascript-based technology, was used for better user experience.

## 5.2 Nette Framework

Nette Framework is a Czech open-source framework for building Web applications in PHP 5. It eliminates security risks, which for the licensing system is very important. It supports AJAX (Asynchronous JavaScript and XML), DRY (Do not repeat yourself), and KISS (keep it short and simple). It fully supports the MVP design pattern (Model View Presenter). It is not necessary to use MVP, but it is clearer and more comfortable. Nette leads developers to clean object-oriented design of applications with emphasis on future extensibility. One of the great advantages of this framework is that around them there is a large active Czech community [2], and therefore one can get support directly from developers. So many components are available thanks to the existence of a community and the openness of the framework [3].

### 5.2.1 MVP

MVP is a design pattern, by Pecinocsky [6] or rather, software architecture, used in software engineering. The main aim of this architecture is to separate the business logic and application data from the user interface. MVP is derived from MVC (Model View Controller), which is more commonly known. Both divide applications into three layers:

- The data model (Model) – provides access to data and manipulation with it,
- The user interface (View) – converts the data represented by model into a form suitable for presenting to the user,
- The control logic (Presenter/Controller) – responds to requests from user and provides changes in the model or view.

Dividing three layers for implementation and above all, maintaining them separately, will make our application more robust and flexible.

The main difference between MVP and MVC is how requests by manager are accepted (Controller/Presenter) [7]. In MVC all application requests come straight through to the controller and it will decide what to do with the requests (forward to View or Model). It is illustrated on figure 5.1. MVP is different in that the application requests are accepted by View and forwarded to Presenter. It is illustrated in figure 5.2.

In the proper design according to the MVP pattern, the application developed in Nette framework should follow these rules:

- Model should not know that any Views or Models exists.
- View should not know about Model
- Presenter makes View familiar with the Model (not vice versa), and implements user actions. User actions are:
  - Change of View
  - Change of state
  - Command for Model

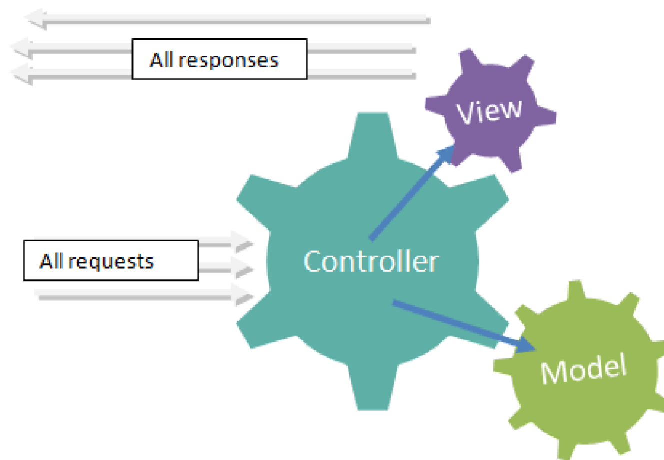



Figure 5.1: Principle of Model View Controller

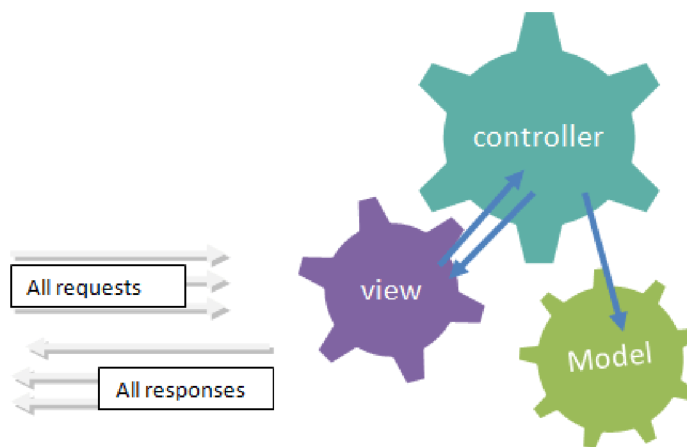



Figure 5.2: Principle of Model View Presenter

In the ideal case the application should follow these rules:

- Limit the activity of Models just to data acquisition and manipulation
- Logic in Views limit just for iterations and conditions
- Logic in Presenter limit just for:
  - Filling of Views
  - Creating of components
  - Communication with Models

## 5.3 Generally about models

Models are used to obtain data from a particular source, and in most cases from the database. Contrary to the views and presenter, which have an absolutely strict form, we have complete freedom in the creation of models. Usually the database layer is used in the model, and in our case it is dibi library, which is part of Nette Framework.

In order to work with data in the models, it is necessary to first connect to the database. This connection is realized only at one point in the project, in script loadconf.php. Registration is very simple:

```
dibi::connect(Environment::getConfig('database'));
```

The static class `Environment` represents an environment, where the application is currently running, and contains method `getConfig`. This method is for the obtaining of settings from the configuration file cnofig.ini, which contains settings of database:

```
database.driver = mysql
database.database = database
database.host = localhost
database.username = username
database.password = password
```

The configuration file contains, in addition the setting of the database, all the necessary settings to run applications on the server, e.g. the setting of PHP, variables, services etc.

A main consideration was, then, the method to access data through models. There are two possible options.

The first option is to use the Active Record design pattern, which simply means that one class will represent a record from the database, while the second will cater returning of records from the database in the format of the class.

The second is simpler, as it only consists of a class that will return the records from the database as an unspecified class, i.e. in the array.

Although the first option is the better approach in terms of software engineering for the ease of scalability of code, only the second option was implemented in most cases. The reason was that the application of specific class would be minimal. Even though the second option was mostly used, support was also implemented for Active Record technology. This will be discussed later.

The class which provides the reading of records from the database is called Manager. Besides returning records from the database, it performs other functions. The basic functions it caters to are as follow:

- Creating of record
- Reading of record(s)
- Updating of record
- Deleting of record

These four basic functions of persistent storage are in short called CRUD. In the case of a class based manager like in the Active Record design pattern, creating, updating and deleting would be contained in specific class. Other common but more specific functions include:

- Reading of records as pairs
- Reading of records as data source

The class diagram is shown on figure 5.3. For clarity attributes and methods are not shown. As you can see, there is convention of manager classes naming, database singular table name in camel case and Manager on the end. There are two abstract classes BaseManager and ClassBaseManager. The first is the parent for managers which return records with unspecified class, and the second is the records with specified class. In other words ClassBaseManager is the class which supports the Active Record design pattern. Thanks to the naming convention it is easy to notice the name of table, which should be used in database queries of common methods included in these two abstract classes.

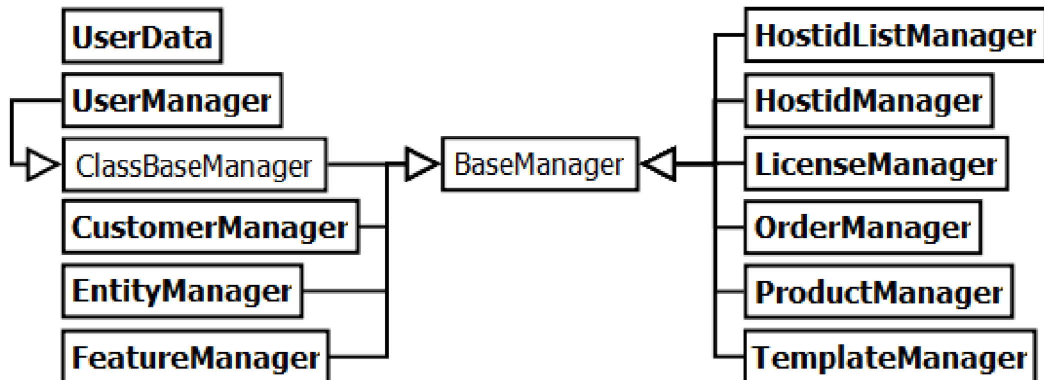



Figure 5.3: Class diagram of models

## 5.4 Specific models

The following sections describe the managers, which contain some behavior that makes them significantly different from the others and therefore worthy of mention.

### 5.4.1 Order Manager

An Order can be created plain or from a predefined template. If it is created from template, the identifier of template is recorded in order. But the template is editable. For this reason a copy of template properties has to be stored. Therefore, the properties of the template are gradually crawled and copied.

### 5.4.2 License Manager

For the same reason, the principle of copying of records was used in license manager, with the difference of it copying order properties to license properties.

Another function of this manager is the generation of XML templates, which are inputs for the license generator. The DOM (Document Object Model) was used for these purposes.

DOM is a platform- and language-independent interface providing access, creation or modification of content, structure or style of document or their part [5]. The principle of DOM is the storing of document as a tree containing elements of the document.

It was useful for just creating of content, because every property of license are stored in the database and can also be edited there. XML template is generated when you create a license or regenerated in the editing.

### 5.4.3 Managers representing license properties

It goes for feature, hostid list and hostid manager. First it is necessary to explain the affair, which apparently is not related to these managers. Due to the relatively high complications with implementation, the license history, especially of temporal nature, was not implemented. It has an influence on the editing of licenses. Instead of the implementation of history, the creation of a copy with changed properties would be used for editing of the license, so that the records in the database would be never changed, only created.

Before the license history is implemented, we have to provide editing of license. If we do it, we also have to regenerate template file. For this reason a mechanism which regenerates template files of relevant license is implemented in managers of license properties.

## 5.5 Views

As was said in section 5.2.1 views are used for presenting of data to user. The best way of creating views is by templates. Although PHP is in a way templating language, in its pure form it is not suitable for coding them. Hence various template engines have arisen. Nette also contains a template engine. However it does not have to be used. Instead of them, the Smarty template engine, for example, can be used.

The Nette templates have to follow a strict form. The template has to be named like an action, to which it belongs with suffi `phtml` and have to be stored in directory named like the template. The template includes information about what action of presenter draw on the screen. It contains HTML and macros. These macros are transformed with so-called filters. For templates of this project the Latte filter was used, which is part of Nette Framework.

Thanks to this filter it is possible to write constructions like this:

```
{if $showForm}
    Hello {$name} please fill this form:
    {control $informationForm}
    or go <a href="{link 'somewhere'}">here</a>.
{/if}
```

This short code contains macro `if`, `control` used for drawing of forms or components and macro `link`, that generate destinations. Code write variable `name` on screen. Normally there would be security risk of XSS, but Nette automatically escapes all variables.

## 5.6 Presenters

The Presenter communicate directly with models and mediate presentation of results in human-friendly form.

Before we say more about implementation of presenters in this project, we should know how they are stored. Basically we divide application to more modules, thus achieving modularity. For example, an application can be divided for front end or back end. In these modules, Presenters and also templates are stored; models are often stored separately. But everything is up to one. Nette presenters have to comply to strict form, which applies also to the naming of presenters. An example of a name of class of presenter for entities is such: `Admin_License_EntityPresenter`. The class name includes hierarchy of modules separated by '_' and on the end is name of presenter with `Presenter` suffix.

On figure 5.4 is the class diagram of presenters. For clarity attributes and methods are not shown. Names of classes are without models hierarchy prefix. Also, it is not shown.

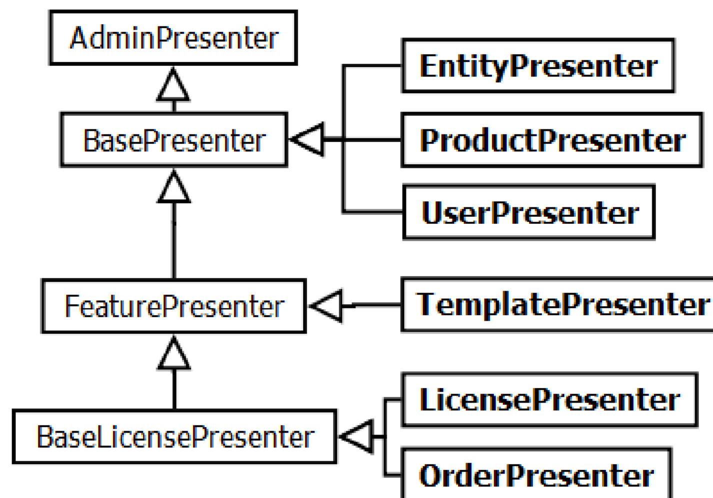



Figure 5.4: Class diagram of presenters

### 5.6.1 Getting of models

The Presenter in this project can use one or more models. Whenever it is necessary to use some of their methods, in order to avoid creating of instances of classes of models, a variation of the design pattern Singleton was used. This design pattern allows us to create a single instance of the class and then use other objects [6].

It is called variation, because the method for obtaining of object of model is not implemented as method of model's class, according to proper implementation of Singleton. It is however implemented as a method of the presenter, meaning there is only one instance of the model in the presenter. The aim of that was a lazy instantiation of class. The code of this technique is as follows:

```
private $someManager = NULL;

public function getModel()
{
    if(!isset($this->someManager)) {
        $this->someManager = new SomeManager;
    }
    return $this->someManager;
}
```

Another thing is how the getting of model is called. Nette provides calling of getter method just by using the getter's attribute, even if it does not exist:

```
$something = $this->model->fint($id);
```

### 5.6.2 Actions

Actions have names and correspondence to views. Every view has to be named like action. For every action two methods can be implemented in the presenter: `action<NameOfAction>` and `render<NameOfAction>`. The reason to why there are two and not merely one is that the Nette Framework is MVP. So the application contains of three parts. The presenter is one of them and work with the remaining two. Method with prefix action is for dealing with database. Method with prefix render is for dealing with views. For example there are set variables used in template. Setting of variable in view representing title can looks:

```
$this->template->title = "Orders";
```

Common actions of this project are default, add and edit. Default action is for listing of records. Add is for adding of record to database and usually is there implemented just adaptation of form component, which is often common for both editing and adding. Edit action is for editing and here is mostly taken care of filling the form by default values. Alternatively, other variables of the view.

### 5.6.3 Data listing

As was described in section 4.3 data should be displayed with the possibility of limiting them by some criteria and the possibility of sorting. Furthermore, it should be possible to process data in bulk. For this purpose a DataGrid component was used, which is written specifically for the Nette.

Before we discuss about DataGrid, it is good to know how the creation of components works. Components are created by method createComponent<Name>(), where Name is name of component, which is used in templates 5.5. This method is called only when it is written in template. Therefore it is basically a lazy initialization of components.

Component DataGrid can work just if we give that some source of data like:

```
$grid->bindDataTable($this->model->getGridDataSource());
```

It shows all columns, which are in data source, but usually we want to show specific columns. In this case we write only the columns we want, as follows:

```
$grid->addColumn('name', 'Name')->addFilter();
```

There is more to this code than just showing particular columns. There is also added filter for limiting of listed data. Actions with specific row are carried by this code:

```
$grid->addActionColumn('Actions');
$grid->keyName = 'id';
$grid->addAction('My Action', 'myAction');
```

Where keyName is identifier used as parameter of action. Another provided feature is multiple operations on rows:

```
$operations = array('delete' => 'delete');
$callback = array($this, 'gridOperationHandler');
$this->allowOperations($operations, $callback);
```

Array operations contain keys that are name of action and values, which are in turn names showed to the user. Array callback contains the definition of multiple operations handler.

Due to the fact that almost every DataGrid will support multiple delete action, class DeleteDataGrid, which inherits from DataGrid and extends it for multiple delete action and handling of that, was created.

For a better idea how the final component may looks like, is shown example in figure 5.5. This particular example is list of entities. As you can see, the user filtered the entities, in this case states that lie in Europe, in other words, are subordinate entity of Europe. Also some states are selected, which can be processed in bulk using of some actions, for example, deleting of these states. If should be displayed in the DataGrid many items is allowed to view only a certain amount of these elements. Other elements will appear by switching to the next page. It is possible to set the number of elements on the page. If paging is not satisfactory you can show all the elements. This functionality is also sometimes called paginator.

| | Name | Parent entity | Actions |
|---|---|---|---|
| | | Europe | Apply filters |
| ☐ | Czech | Europe | edit delete |
| ☐ | Switzerland | Europe | edit delete |
| ☑ | Austria | Europe | edit delete |
| ☐ | Germany | Europe | edit delete |
| ☑ | Slovakia | Europe | edit delete |

Selected: delete Send Items 1 - 5 of 5 | Display: 15

Figure 5.5: Listing of the items in `DataGrid`

### 5.6.4 Forms

Generally there are just two types of forms. The first is a form for confirmation, now used just for confirmation of delete action. The second is a form for input or editing of data.

Some events require increased user attention. The deletion of records is one of them. For this reason a confirmation form was implemented. Called `ConfirmForm`, it inherits from component `ConfirmationDialog`.

Action or signal for confirming is defined by three items: name of action or signal, callback called when confirmation is succeed and callback which contain questions. The user is always informed about the action and items over which the action is or will be performed.

The text of messages is adapted to whether the operation is carried out over one or more items. As exceptions exist in the English language for the creation of the plural, it was necessary to implement it with support for these exceptions in displayed question or information messages.

Here is how `ConfirmForm` is created in `EntityPresenter`:

```
public function createComponentConfirmForm($name)
{
    return new ConfirmForm($this, $name, $this->model,
    "entity", "entities");
}
```

As you can see, there are five parameters. The first two are common for all components - presenter itself and name of component. The third parameter is model, which is used for obtaining of items' names. The last two parameters are singular and plural name of item/items.

Nette provides for the creation of the forms component `AppForm`. This component is very powerful. More will be discussed later.

There was one thing which had to be considered. Input fields are rather similar in almost all forms for editing or creating of item. In these cases only one form for editing and creating of data is created. If an input field should be shown, recognition is realized

by testing of the HTTP parameter which identifies the item, rather said, whether it is set or not.

AppForm contains methods for creating many types of inputs: hidden, text, text area, select, submit etc. It is possible to define validation rules to inputs, or if it is necessary to use custom validation rules. Here is an example of field with validation rule:

```
$form->addText('name', 'Name:')
      ->addRule(Form::FILLED, 'Name is required');
```

For filling of the selects (ie. combo boxes), `getPairs` methods from models are used. `AppForm` lacks one important field, i.e. a field for date. For this purpose, there is an external component `DatePicker`, which is for example used in adding of feature as is shown on figure 5.6. It is based on jQuery component Datepicker.

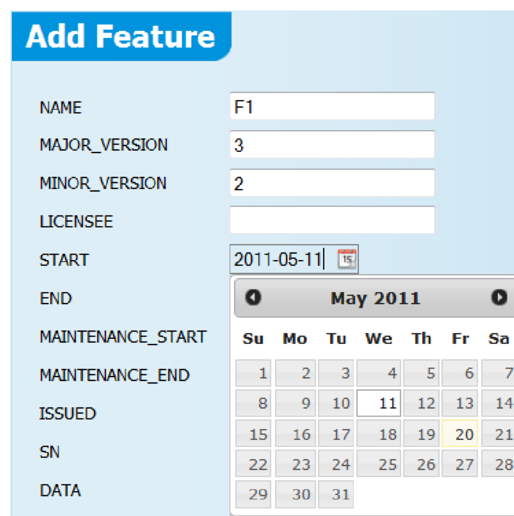



Figure 5.6: Adding of feature with date input

In one case it was necessary to solve the problem of depended select box. It was in `OrderPresenter`, where the user has to select product and then template, during the creation of a new order. It is solved by an external component `DependendSelectBox`.

There are two ways to ensure the processing of the form. One can either use `onClick` array for callback of particular submit item, or use onSubmit array for callback of the whole form. I used the second method, because the processing of request is handled as a whole, as opposed to the creation of method for every submit input.

### 5.6.5 Download of file

The template license file which is generated is stored in database as well as license file after uploading. It is necessary to provide the user the possibility of downloading these files. These functionalities are implemented as action of `BaseLicensePresenter`.

First file content and file name from database is obtained and then the file is transferred by using HTTP response. The code is done as follows:

```
$response = $this->getHttpResponse();
$response->setContentType('text/plain');
$response->setHeader('Content-Disposition',
                     'attachment; filename="' . $license[$fileName] . '"');
echo $content;
$this->terminate();
```

The content type is set to the HTTP response, and then the name of file is set in the header. After that the content of file is simply written by `echo`. In the end the presenter is correctly terminated.

# Chapter 6

# Deploying of the application

The information system, the result of this project, can be used by deploying on the server, which must meet the following requirements:

- Apache 2
- PHP 5.3
- MySQL 5.1

Versions are only recommended, because testing was carried out on them.

The easiest way to deploy the system is using of LAMP or XAMPP package, that provides the platform for running web applications. This is also a solution for using stand-alone running of the system. However, the user will lose access from anywhere in the world, as long as there is access to the Internet.

The installation is carried out such that the project is copied into the web directory of Apache. Scripts for creating of database structure are located in directory `/data` that must be run on the database server. The script that contains the license management is called `license.sql`.

In the instance of the removal of a license management, the following script can be used: `license_delete.sql`. It would also be necessary to remove the directory `/admin/app/AdminModule/LicenseModule` and the database models from the directory `/admin/app/models`.

# Chapter 7

# Conclusion

The system, which allows the vendor to centrally manage the software licenses of his products, is an information system designed for deploying on a server that is accessible over the internet. This system is used to store the license files and generate files that serve as input to the generator, of which outputs the license files.

In the beginning, goals of the project were set. The goals of making the system applicable for license management were met. It allows storing and managing of the orders and licenses. In addition, the system allows the creation of orders, and hence the licenses from previously predefined templates. Thus it provides the user of system the comfortable creation of licenses, when in the case of well-defined templates, only adapt the license for the use of a particular customer.

The solution was implemented in PHP using Nette Framework. The system architecture is built on the MVP pattern, which separates application logic from the user interface structure. It splits application into three layers. Thus, some modification has minimal effect on other layers.

## 7.1 Future extensions

There are extensions that would make the system a relatively complex solution. These include in particular, the extension by the customers' part, allowing customers to access their licenses. However, since the system was designed primarily as a solution for the company Codasip, which has very specific focus and thus a relatively narrow range of customers, it is not crucial for this time. However, that would be required in the future.

Another possible extension is to preserve the license history. The solution was designed and only remains to be implemented. Its implementation will take place without too much interference to the database structure and be relatively isolated from existing application logic.

## 7.2 Personal benefit

The size of this project required proper time scheduling. This was the first time that I was required to manage my time under such important circumstances. I have improved my skills in the estimation of time, in terms of particular activities. I learned how the licensing of the software works. The last benefit was the familiarization with PHP programming and Nette Framework.

# Bibliography


[1] Lissom. http://www.fit.vutbr.cz/research/groups/lissom/index.html, 2006 [cit. 2011-05-18].

[2] Nette Framework Forum [online]. http://forum.nette.org/cs/, 2011 [cit. 2011-05-11].

[3] Doplňky, pluginy a komponenty [online]. http://addons.nette.org/cs/, 2011 [cit. 2011-05-12].

[4] Paul DuBois. *MySQL*. Addison-Wesley, 2008. ISBN 0-672-32938-7.

[5] Jiří Kosek. *PHP a XML*. GRada, 200. ISBN 80-247-1116-8.

[6] Rudolf Pecinovský. *Návrhové vzory 33 vzorových postupů pro objektové programování*. Computer Press, 2007. ISBN 978-80-251-1582-4.

[7] Ronald Widha. Difference between model-view-presenter and model-view-controller [online]. http://www.ronaldwidha.net/2009/03/19/, 2009 [cit. 2011-05-12].

[8] X-Formation. License distribution service developers manual, 2009.

[9] X-Formation. LM-X Developers Manual, 2010.

[10] X-Formation. LM-X End Users Guide. http://www.x-formation.com/lm-x_license_manager/enduser.pdf, 2010.