Lund University
Lund Institute of Technology
Department of Communication System

Bachelor Thesis

2001-09-19

Software re-engineering

from function-oriented to object-oriented

Tutor
Thomas Olsson

Authors
Jonas Gyllenspetz
Steffan Tajti

Abstract

This report is a Bachelor thesis at Lund Institute of Technology, conducted by Jonas Gyllenspetz and Steffan Tajti at Timelox AB in Landskrona, Sweden.

The report examines problems with old software programs that, for several reasons, need to be updated. When updating the software, it is possible to rewrite the entire program, or take advantage of the obsolete one.

In this thesis the possibilities to take benefits from of an existing program, when developing a new, is performed as a case study. The case study is performed with a process divided into several phases.

The result of the case study shows, if programming languages differ considerably in structure and inbuilt function, the main benefit taken from an obsolete program is examining its functionality.

In the case study the new program was not implemented, why no comparisons regarding source code amount, effectiveness, and maintainability have been made.

Contents

Preface

This report is the result of the Bachelor thesis that completes our Bachelor of Science degree in Software Engineering at Lund Institute of Technology. The work has been conducted at Timelox AB in Landskrona, Sweden.

We would like to thank everyone who has helped us during our thesis. Especially our tutor Ph.D. Student Thomas Olsson (Department of Communication Systems at Lund University) for his feedback and support.

We would also like to thank our supervisor Martin Kjällman and Magnus Persson at Timelox AB.

Jonas Gyllenspetz

Steffan Tajti

1. Introduction

In many companies, worldwide, there exist old software systems that still provide essential business services [1]. These kinds of systems are called *legacy systems*. The maintenance of legacy systems provides three notable problems [1].

- The systems have often been modified several times by different programmers.
- The modifications are often made over a large period of time, maybe 10 to 20 years.
- In many cases there is a lack of documentation of the systems.

Consequently, it is impossible that neither a single person nor a company have a complete understanding of the system. When these problems with the system reach a point in time, when it is too expensive or too complicated to maintain, the system must be re-implemented [1, 3, 4].

There are three options to choose from when re-implementing legacy systems.

- The first one, sometimes called “cold turkey” [5], is to develop a new system from scratch and reject the existing one [1, 5].
- The second one is to organise and reconstruct the existing system without changing the functionality. These work procedures are called *software re-engineering* [1,3,5,6,7].
- The third option is to reconstruct the existing system in small incremental steps, while it is still in production. This option is sometimes called “chicken little” [5].

The second alternative is preferred, because it has two key advantages compared to developing an entirely new system. The advantages are reduced risks and costs [1, 3, 5].

Approaches to software re-engineering and the use of them are investigated. The knowledge is applied on an existing legacy system, developed in a function-oriented programming language, which encodes magnetic cards.

1.1. Objectives of the thesis

There are two objectives of the thesis.

- The first objective is to provide knowledge on to what extent it is possible to reuse source code in a re-engineering process.

- The second objective is how to obtain an understanding of the difficulties involved, when the existing system is written in a function-oriented programming language and the new system is written in an object-oriented programming language.

To meet the objectives and get sufficient knowledge, a literature study and a case study is performed. The case study is divided into three parts; background, objectives and method. The usage of CASE-tools in the case study (see section 2.4 Supporting technology) is rejected. A requirements specification and an architectural design of a suggested new system are created. The new system is not implemented.

1.2. Programming languages and environments

The system, written in the function-oriented language C, with MS-DOS environment, is still in working order at the company Timelox AB. The system's functionality is to handle different algorithms, which are encoded as hexadecimal strings on magnetic cards. In addition to that, the system handles text strings, which are printed on magnetic cards. The system will be re-written in the object-oriented programming language Visual Basic or C++ in order to work in MS Windows9x/Me environment. The main purpose for re-engineering the system is to make it more maintainable. The new system will have the same features as the existing. The new system is not implemented in this thesis. Programming languages and environments are more carefully described in section 3.3 (Company) and 3.5 (Program languages in case study).

1.3. Overview of the thesis

The thesis is divided into five parts.

- Chapter 2 consists of an overview of the concept of software re-engineering. In this chapter an approach to software re-engineering is performed. This approach includes a process with five key phases. In the phases of the process it is possible to incorporate, one or more, of five activities. These activities are also described in this chapter. Furthermore, chapter 2 includes a presentation of supporting technology in software re-engineering.

- In chapter 3 the case study definition is presented. A survey over the company and the company's hardware and software is performed. A method based on the approach to software re-engineering in chapter 2 is also handled in this chapter. Finally, chapter 3 presents the programming languages involved in the case study.

- The implementation of the case study is described in chapter 4. The method in chapter 3 is followed. The different sections in the chapter are file description, module diagram, evaluation of existing system and development of the new design.

- Chapter 5 presents the case study's analysis. In this chapter, recommendations for improvements are discussed. A diagram, which shows what phases and activities included in the case study, is presented.

- Finally in chapter 6, conclusions and summary are presented.

2. The concept of software re-engineering

2.1. Introduction to software re-engineering

Changing software systems, without changing their functionality, is called software re-engineering [1]. The difference between software re-engineering and ordinary software engineering is that earlier developed systems are used as input in the re-engineering process (see Figure 1).

There are two essential advantages with re-engineering compared to developing new software. The advantages are reduced risk (there are often problems with development, staffing, and specification in new software) and reduced costs (the cost of re-engineering is often significantly less than the costs of developing new software) [1].

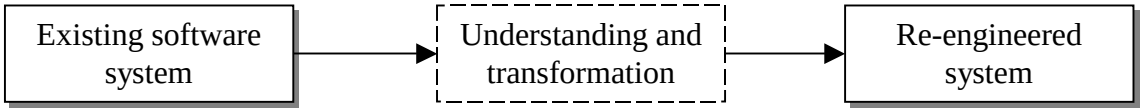


Figure 1. Software re-engineering

Software changes are mainly made for making the systems more maintainable in order to extend lifetime and to reduce maintenance costs. Sometimes software changes are necessary due to new requirements, which was not included in the original system. The new requirements are for example; changes in operating system environments or the need for implementing new functionalities. Other desirable improvements are time constraint, productivity and quality [3, 6].

The existing system can be used for creating software specifications in order to get overall comprehensibility for new development. It can also be used for translating a system's programming language (for example from C++ to Java). The amount of source code used depends of the input quality and the expected outcome [6, 8].

Another interesting issue is that more and more people are involved in changing software (for instance enhancements as software re-engineering). Figure 2 shows the extrapolation for the number of programmers (worldwide) working on new projects, enhancements, and repairs [7]. In the 1990's, only three out of seven programmers were working with new projects. The forecast predicts that in the 2020´s only a third of all programmers will be working with new projects.

Due to increasing requirements for maintenance and improvement, the area of software re-engineering is gradually becoming more important.

Year	New projects	Enhancements	Repairs	Total
1950	90	3	7	100
1960	8 500	500	1 000	10 000
1970	65 000	15 000	20 000	100 000
1980	1 200 000	600 000	200 000	2 000 000
1990	3 000 000	3 000 000	1 000 000	7 000 000
2000	4 000 000	4 500 000	1 500 000	10 000 000
2010	5 000 000	7 000 000	2 000 000	14 000 000
2020	7 000 000	11 000 000	3 000 000	21 000 000

Figure 2. Programmers involved in different software activities.

2.2. Software re-engineering process

To obtain a better understanding, when starting the software re-engineering process, the existing system has to be carefully analysed. Once an organisation understands the concepts and the factors that influence the change of a software system, it becomes possible to harness the direction of the development. The technique for re-engineering the software is to establish a process [3, 5, 7]. In the process, several phases are possible to identify. Typical phases are system inventory, strategy determination, impact analysis, detailed planning and conversion [7].

One approach to re-engineering is to comprise the process into five key phases [3]; *preliminary inventory analysis, encapsulation, application analysis, production standardization,* and *design recovery*.

2.2.1. Preliminary inventory analysis

Before re-engineering begins, a preliminary inventory analysis of the existing system is performed. This is done to determine the overall scope of the re-engineering effort. This first phase is a reduced version of the next two phases in the process, the encapsulation and the application analysis. In this phase it is not necessary to develop a detailed inventory of the system's components. The main objective is to determine the limitation of what to do in the process. The requirements specification for the new system is also created in this phase [3].

2.2.2. Encapsulation

Encapsulation is essential to re-engineering because it ensures that all system components are identified. An accurate component inventory must be developed before the analysis in the next phase is performed [3].

This phase is conducted to identify all possible components in the system and also to find components that are not a part of the system. When defining the components, it is possible to use software tools, to make an automated analysis. It is recommended to use it with a manual analysis. This approach is recommended, in order to find misplaced system components that are not found with automated analysis tools. Using a combination of both manual and automated analyses provides the most accurate inventory in the least amount of time [3].

The main objective in this phase is to ensure that all system components are identified. When the inventory is performed, it is possible to ignore the components, which are not important to re-engineer [3].

2.2.3. Application analysis

When the application analysis is performed, there are two attributes to take in consideration.

- Ability to support the system's functional requirements
- System design and use of technology

Evaluation of these two attributes in the analysis provides insight about what is important to improve in the system's re-engineering process. For example, a system that adequately supports the needs of the user but operates with obsolete technology is more valuable than one that provides little or no functional support to the user, but operates with all the latest technology.

The main objective in this phase is to provide a better understanding of functional fulfilment and how to improve the technical quality of the system [3].

2.2.4. Production standardisation

Production standardisation transforms an existing system into one that performs better, is easier and more cost effective to maintain and operate. It is possible to approach the existing system from two directions simultaneously, functional and technical. A functional knowledge of the system is required to identify and document the attributes that are supported by the system. The technical aspect of this approach provides detailed knowledge of the processing within the system and the information necessary to improve the maintainability and performance of the system [3].

When the re-engineering effort is completed in this phase, the system is expected to be more cost effective to maintain and operate. The system is also expected to better react to internal and external forces. For example, it is easier to incorporate new components in the system, when the technical and functional aspects of the system are improved.

The main objective of production standardisation is to improve the existing system after years of changes performed by different programmers using different programming styles and techniques [3].

2.2.5. Design recovery

The final phase in this approach to re-engineering is design recovery. Design recovery captures elements of the current system design, with the possibility to incorporate these elements into a CASE tool (see section 2.4 Supporting technology). This phase provides the ability to accurately document the functional and technical aspects of the existing system [3, 6].

The documentation produced from this phase provides insight into several functional purpose of the system.

- The main components of the system.
- The technology used to provide system functionality.
- The applications that use the system.
- The interface to other system.
- The architecture of the system.

2.3. Activities in software re-engineering process

In every phase in the software re-engineering process it is possible to incorporate activities. There are five activities, which are explained and investigated. The activities described here are: *source code conversion, program structure improvement, program modularisation, data re-engineering,* and *reverse engineering* [1].

2.3.1. Source code conversion

Source code conversion is the simplest form of software re-engineering [1]. This activity is used when it is possible to convert a program written in one programming language to another. This is typically done when the programming language in some way is obsolete. Reasons for source code conversion may be lack of skilled staff for maintenance and support or hardware platform updates. Automatic conversion is feasible, but in many cases this is impossible because it is very difficult to make a complete automatic conversion A reason why automatic conversion is impossible is the lack of corresponding constructs between languages [1, 8].

Source code conversion can be separated into two different categories, source code *translation* and source code *transformation* [9]. Source code translation is used when conversion is made between different programming languages such as COBOL to C. When conversion is made in the same programming language, between different versions or dialects, this is called source code transformation. Transformation can also be difficult because the same syntax can have different behaviour in different dialects (see example below). This is known as the homonym problem and exists for instance in different dialects of COBOL [8]. An example where the same syntax has different outcome in different dialects though the language is the same.

```
PIC A X(5) RIGHT JUSTIFIED VALUE ‘HELLO’.
DISPLAY A.
```

The code above gives with the OV/VS COBOL compiler the expected result, displaying the word ‘ HELLO’ right justified. But using the COBOL/370 compiler the same word has a trailing space and is left justified.

2.3.2. Program structure improvement

The previous need to optimise memory use, due to computer hardware limitations, is one of the reasons why many computer programs have complex structures. Consequently, these computer programs are problematical to understand [1]. Another reason for insufficient structured programs are programmers having limited understandings and skills in software development [4]. Therefore, program structure improvements are performed, with the aim to make the programs easier to understand and maintain.

Source code may consist of unstructured control logic, cryptic variable names, or complex conditions [4]. For example, conditional statements including negation expressions can be made more understandable excluding this (“not”, “!=”, etc, depending on language). In any case, excluding negations, not always mean that the code becomes easier to understand.

Furthermore, source code can exist without being reached, as developers do not dare to remove code they are uncertain about. This kind of code is hard to discover without making an extensive analysis [1].

Program structure improvement can be made automatically. The automatic analysis may generate a diagram, which shows the flow through the program. Simplification and transformation techniques can be applied to the generated diagram without changing its semantics. With this technique, unreachable code is identified and can be removed. Once simplifications are completed, a new program is generated. This automatic analysis may not work properly if the program is written in a non-standard language dialect. [1]

Problems with automatic program restructuring are amongst other things, losses of in-line comments and documentation in the program. However, this is often not significant, since these are often out of date or of no importance after restructuring [1].

2.3.3. Program modularisation

Program modularisation is the process of re-organising a program so that related program parts are collected together in modules or objects. For example, functions handling the interface are put in a separate module. By collecting similar parts it is easier to find redundancy and to formulate a better overview of the program. Program modularisation is performed manually by inspecting the code. Different tools can help with browsing and visualisation, but to automate the process is virtually impossible [1].

Program modularisation is done to simplify maintenance and to improve understanding of the program, to enhance cohesion within parts, and to minimise coupling.

Several different types of modules may be created during the program modularisation process [3]. These include:

- **Data abstractions:** Abstract data types where data structures and associated operations are grouped.
- **Hardware modules:** All functions required interfacing with a hardware unit as for example a printer.
- **Functional modules:** Modules containing functions which carry out closely related tasks as for example calculations or graphics.
- **Process support modules:** Modules where the functions support a business process or process fragment. This might in a bank system be the part of the program that handles the functionality required when redrawing cash from automates.

2.3.4. Data re-engineering

Data re-engineering is the process of analysing and re-organising data structures (and sometimes even data values) in a system in order to make the system more understandable [1]. This activity may be a part of the process of migrating from a file-based system to a DBMS-based system (see Appendix A - Glossary) or changing from one DBMS to another. Changes in the software may affect the data storage, data format, or the data values [4]. The objective with data re-engineering is to establish a manageable data environment.

2.3.5. Reverse engineering

Reverse engineering is the opposite of the expression forward engineering. Forward engineering is often mentioned as the process of moving from high-level abstract specifications to detailed low-level specifications for implementation [6, 7]. The primary purpose for reverse engineering a software system is to increase the overall comprehensibility of the system for both maintenance and new development [4, 6].

Reverse engineering can be used when deriving a systems design and specification from its source code [6]. The process does not necessarily move from source code to specification. Variations in the process depend on available input and expected output. In some situations the source code might not be available and the only input is the executable file.

When starting the reverse engineering process, it is possible to make an automated analysis by using tools for program understanding. This is, however, not enough to re-create the system specification. Engineers also need to make manual annotations [3]. The annotations and the information collected by the automated analysis, are stored in a system information store. This storage is used for generating documents of various types such as program and data structure diagrams.

After the system design documentation is generated, it is still possible to add more information to the information store. This is done in order to re-create the system specification. This procedure usually involves further manual annotation of the system structure.

The different steps in reverse- and forward engineering can simplified be categorised into four parts [9].

- **Objectives**
  The objectives express the purpose or need for the software. This can be more general, for example company business goals.

- **Requirements**
  The requirements describe the software system for achieving the objectives. The formulation of requirements is part of any software process, but there can be major differences in what this activity is to include.

- **Specification**
  The specification (design) describes the software system in an explicit way, with understanding of how the system works and interacts with the surroundings.

- **Implementation**
  Implementation is the step where the system is implemented according to the specification.

A software process including these steps, usually in this order, is called forward engineering. Reverse engineering include one or more of the same steps but in the opposite order (see Figure 3).

The initial level, when using reverse engineering, depends on the quality of the available documentation. Furthermore, there is not always a need for recovering the objectives. After completing the reverse engineering process, the outcome is applied on the new system.

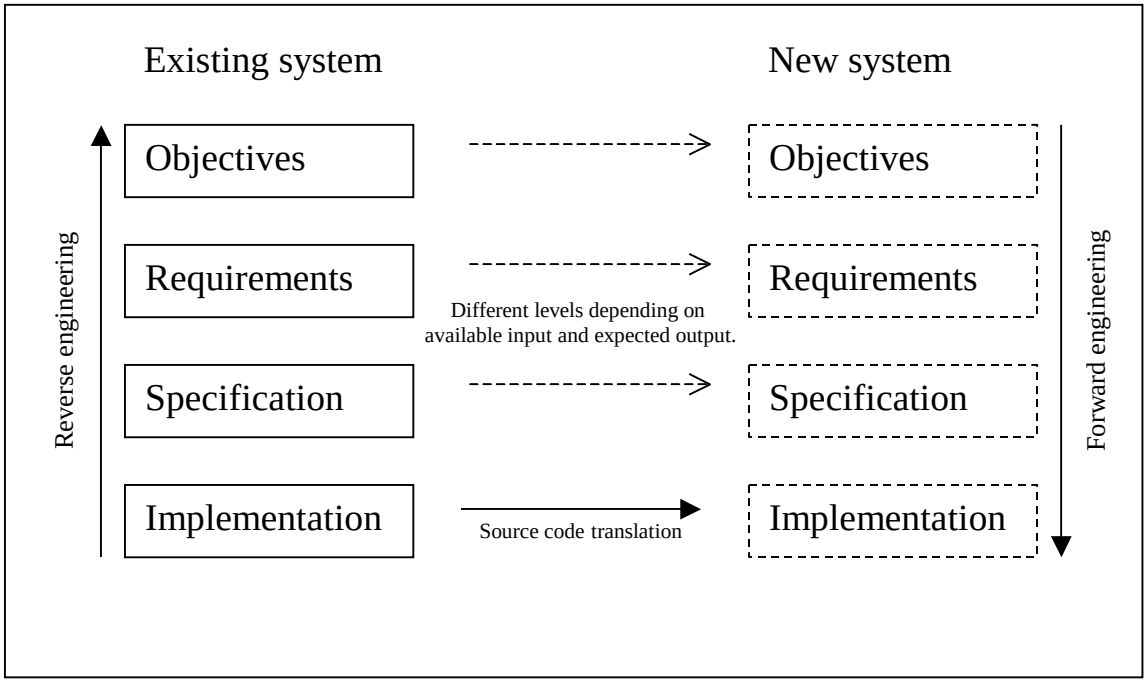


Figure 3. Reverse engineering compared to forward engineering.

2.4. Supporting technology

It is very laborious and difficult to change software [3, 8, 10]. Programmers are frequently hesitant and sometimes even resistant to re-engineering the program even if they are convinced that it will improve the system and reduce the maintenance costs. Their arguments are: it will take too long time, it is impossible to meet the objectives and it is too difficult to make the change [3].

Assistance in this dilemma is to apply a supporting technology named computer-aided software engineering (CASE) [6]. Reverse engineering is rapidly becoming a recognised and important component of future use of CASE-tools [6]. CASE by definition is a technology that applies an automated engineering-like discipline for the specification of computer software system design, software development, testing, maintenance, and also project management [3]. CASE provides an opportunity for re-engineering.

For example, CASE-tools can be used for analysing software. They can include facilities such as diagram tools, prototyping facilities, an audit or checking facility to guarantee completeness, a repository to store data flow diagrams, data entity relations and reports and a link to other repositories [3]. The outputs are functional requirements, data structures, database or dataset designs, and report definitions. Other CASE-tools can be used to create UML diagrams from source code and also to make new source code from UML diagrams [1].

With CASE-tools it is possible to improve the quality in the re-engineering process. This is performed with reuse of design elements and through reverse engineering. Furthermore, together with manual analysis of the source code, CASE-tools enhance productivity and provide a high-quality system [3].

2.5. Summary

The objective of software re-engineering is to reorganise and modify existing software systems in order to make them more maintainable. This is made without changing the original functionality. Modification might change the source code completely (different language) or cause minor changes as for instance minor program structure improvements.

Modifications are often performed to improve the system structure and to make it easier to understand. Reasons for re-engineering a system instead of starting from scratch are expected cost reduction and lesser risks.

A process comprised in five key phases is one approach to software re-engineering. These phases are called preliminary inventory analysis, encapsulation, application analysis, production standardization, and design recovery.

In the process's phases one or more activities are performed. They describe what is actually done with the software. The activities are; source code conversion (changing language or dialect), program structure improvement (improving the structure), program modularisation (gathering parts that are related in modules), data re-engineering (re-organising data structures), and reverse engineering (create representations of the system at a higher level of abstraction).

A supporting technology, in re-engineering process, is to use automated tools as CASE. The use of CASE-tools does not exclude manual analysis of the source code being changed.

3. Definition of the case study

3.1. Introduction

This chapter describes the primary objectives and method of the case study. It also includes an introduction to Timelox AB, the company where the case study is performed. Furthermore, a description of the existing system (to be re-engineered) and the desired new system is performed. The systems software languages and differences between them are also described.

3.2. Objectives

The primary objective of the case study is to acquire deeper knowledge concerning the existing system and to find code and significant algorithms to re-implement in the new system. The existing system's function-oriented structure is examined and evaluated in order to design a new structure in an object-oriented manner. This results in important information, where significant files are found and examined, and in addition, irrelevant files removed (described in chapter 4).

3.3. The company

The company where the case study is performed is Timelox AB. They develop, manufacture, and market electronic locking systems where magnetic cards and smart cards are used as keys. Hotels, municipalities, county councils and other public services are amongst their customers. Magnetic cards used by guests and employees are encoded and printed by card encoders at hotels and companies.

3.3.1. Printer used in production

In Timelox's production a printer called Protechno is used. Protechno prints text and encodes magnetic cards. These cards are special cards used by operators and administrators. They are for instance used for initiating locking systems or for unlocking doors in case of emergency. Every locking system is delivered with a unique set of operator cards.

3.3.2. Existing system

The existing system used today, to control Protechno, is obsolete and difficult to maintain. This results in heavy maintenance costs, which makes the system expensive. In addition, it is impossible to add source code to the system, because of the complex program structure.

There are 149 different ways to encode and print text on magnetic cards. The different ways are represented as article numbers in the source code. Every article number describes one way to encode and print text on the magnetic card. The codes are in some cases clearly stated in the source code but some of the codes have complex algorithms and randomised characters. When calculating these algorithms, data from a data file is used. The data used from the data file is in some cases changed and rewritten.

The system has been modified several times during fifteen years of usage and approximately ten programmers have been involved in the modifications. The documentation of the system is inadequate. Therefore, there is no programmer at Timelox that has a complete understanding of the system.

The system is written in C and runs on Microsoft-DOS.

3.3.3. Desired re-engineered system from Timelox

Timelox has decided to develop new modern software that, besides the opportunity to use multiple printers, will have the same functionality as the existing. A new printer called Fargo has been purchased. Initially this new printer will be used together with the new developed software.

The new re-engineered system will be written in Visual Basic and will be running on the Microsoft Windows 9x/Me platform.

3.4. Method

To meet the objectives in the case study, a method based on the approach to software re-engineering in section 2.2 (Software re-engineering process) is used.

3.4.1. First phase: Preliminary inventory analysis

Before re-engineering begins, a preliminary inventory analysis (see section 2.2.1) of the existing system is performed.

The first step is to obtain a broader understanding of the existing system. Informal interviews with the employees in the production are made. A demonstration of the existing system is also given. As the interviews are informal, without predefined questions, this is not documented. The main purpose is to determine a better initial start point.

The first activity used, when making the existing system’s preliminary inventory analysis, is reverse engineering (see section 2.3.5). The source code, with some useful but mostly cryptic annotations, is used as input when starting the activity. The result is that a requirements specification is written.

3.4.2. Second phase: Encapsulation

The next phase is encapsulation (see section 2.2.2). The activity used is program modularisation (see section 2.3.3). The objective in this phase is to identify all system components. This is made by a more careful examination of the existing source code, than made in the preliminary inventory analysis. The result of this phase is a module diagram, which shows how the function-oriented system is constructed.

It is possible to ignore some components, because they are not significant for the system and do not need to be analysed in the next phase.

3.4.3. Third phase: Application analysis

Application analysis (see section 2.2.3) is the third phase and the activities used are program structure improvement (see section 2.3.2) and data re-engineering (see section 2.3.5). In this phase important files are even more carefully examined than earlier phases. Important data are collected and stored in a database. Files of no interest for further re-engineering are ignored and removed.

3.4.4. Fourth phase: Production standardisation

The production standardisation phase (see section 2.2.4) is ignored in the case study. The reason for this is that the new system is developed in an object-oriented manner and it is hard to apply on the existing system. Furthermore, it is not enough to improve the system with production standardisation.

3.4.5. Fifth phase: Design recovery

The last phase is design recovery (see section 2.2.5). Two UML diagrams, use case diagram and class diagram, are made manually to get a better design for the new system. Reverse engineering (see section 2.3.5) is the activity used in this phase. In this phase the existing system is documented properly in both technical and functional aspects.

3.5. Program languages in case study

The existing system in the case study is written in the programming language C and runs in Microsoft-DOS environment. The new program is to be developed in the programming language Visual Basic and will run in a Microsoft Windows 9x/Me environment. The class diagrams and the use case models for the new design is developed in Unified Model Language (UML).

3.5.1. C

C is a programming language from the 1970s. It is a function oriented language, meaning that the program is structured by define and call functions. Program flow is controlled using loops, if-statements, and function calls. C allows a precise control of input and output data. This precise control is possible by the use of indirect addressing (pointers) [12].

If no requirements specification exists and there is a need to get a better understanding of a program, it is possible to make a module diagram. Modules do not exist in C but can be simulated by collecting parts of the source code that are logically connected. Even if programs not are developed in such manner, it is possible to create module diagrams manually. The module diagram includes different files and the connection between them.

There are two types of files in C, which are of special interest in the case study's module diagram, source code files and header files. The source code file includes functions for the program. The header-file is an include file, which contains declarations of functions for input and output. When including a header file in a source code file, all information stored in the header file is available for the source code file.

3.5.2. Visual Basic

Visual Basic is an object-oriented programming language [11]. Interfaces are created by drawing controls, such as text boxes and command buttons, on a form. Properties for the form and controls are set by values as captions, colour, and size. Afterwards the code for the controls is written.

3.5.3. Differences between C and Visual Basic

Visual Basic is quite different from C. The main difference is the code structure, where C-applications are written in a function-oriented manner and Visual Basic-applications are written in an object-oriented manner [11]. Furthermore, Visual Basic uses different modules. A program may consist of a single module (for the specific application) but functions and methods may also be collected in separate modules for future use in other applications or for getting a better overview (see section 2.3.3 Program modularisation).

It is possible to write a Visual Basic application function-oriented, but it is not possible to write a C application object-oriented. Another key difference is the development environment in Visual Basic, which generates the code for the graphical interface.

Other differences between the languages are declaration of data types and their behavior. Variable names are similar but when, for example, working with strings, declarations in C are made by using arrays (see Figure 4). Furthermore, there are

boolean variables in Visual Basic for the logical values of true and false. This variable does not exist in C.

Visual Basic example of data type declarations

```
Dim a as Integer
```
- Integer between – 32 768 to 32 767.

```
Dim b as String
```
- String up to 65000 characters.

C example of the same data type declarations

```
int a;
```
-Integer between – 32 768 to 32 767.

```
char b[50];
```
- String up to 50 characters (using characters in an array).

Figure 4. Differences in data type declarations

Other examples of differences, besides semantics, are statements and comparisons that differ in syntax. The if-statement shown in figure 5 shows differences in how comparisons are made and how statements are ended. Differences in comparisons for testing equal values differ in the use of "= =" in C and "=" in Visual Basic. The use of "=" both for comparisons and assignment in Visual Basic can cause confusion when comparing the two languages. Another example when comparing values that differ is the use of " < > " in Visual Basic and " != " in C.

Visual Basic example of an if-statement

```
If a=b Then
  a = 7
  b = 5
End If
```

C example of the same if-statement

```
if (a==b){
  a = 7
  b = 5
}
```

Figure 5. If-statement syntax differences

4. Implementation of the case study

The implementation of the case study is performed with the process described in 2.2 (Software re-engineering process) and the method described in 3.4 (Method).

The implementation includes an overview of the existing system with file descriptions and module diagrams. Further analysis of the existing system, with elimination of files and investigation of remaining files, is also included. Finally, the development of the new design is handled and the explanations of UML diagrams are made.

4.1. File description

To obtain a good overview of the existing system a preliminary inventory analysis (see 2.2.1) is performed.

The existing system has a very complex structure and therefore very difficult to maintain. The structure is complex, mainly because of continuous source code addition. Because of it's structure, without database handling and general methods or functions, it is necessary to add new source code every time the company wants to add a new card for coding and printing. In addition it is impossible to add more source code to the program, because if more code is added it becomes virtually impossible to compile.

The system consists of 51 files. There are 15 source code files (approximately 15 000 lines of code, including comments), 16 header files, 2 text files and 1 data file. The remaining 17 files are of two kinds, those used when making the system and those generated by compilation.

After achieving a good overview of the system a requirement specification is written (see Appendix H).

4.2. Module diagram

After the preliminary inventory analysis is performed and the requirement specification is written, a more detailed investigation of the files is required. This is performed with the encapsulation phase (see 2.2.2). This phase ensures that all components in the existing system are identified. Furthermore, to find components that not are a part of the system.

As a part of the encapsulation phase, selection of files of importance is in the first attempt concentrated on source code files, header files, data files and text files. Files used when making the system and those generated by compile are not included in further analysis. Left for evaluation, after the first reduction, are 34 files. The entire file description is found in Appendix C.

To obtain an understanding of how the files are connected to each other, a module diagram [12] (which includes the header and source code files) is used. Using a module diagram is made because it is a good way to get an understanding of programs with complex structures [12]. The entire module diagram is found in Appendix B.

A module is a separate part of the program. Its task is to handle data that are logical connected [12]. For example, one module handles the graphical presentation and another handles the mathematical standard functions.

In general, every module consists of 2 parts, the module specification and the module body (see Figure 6). However, in some specific cases the module consists of either the module specification or the module body.

The specification in the module diagram is the header file, explaining how the module is constructed. It consists of declarations and definitions of functions and variables from the source code file. The body in the module diagram is the source code file. It includes definitions of internal and external functions and variables. These are declared in their own module specification.

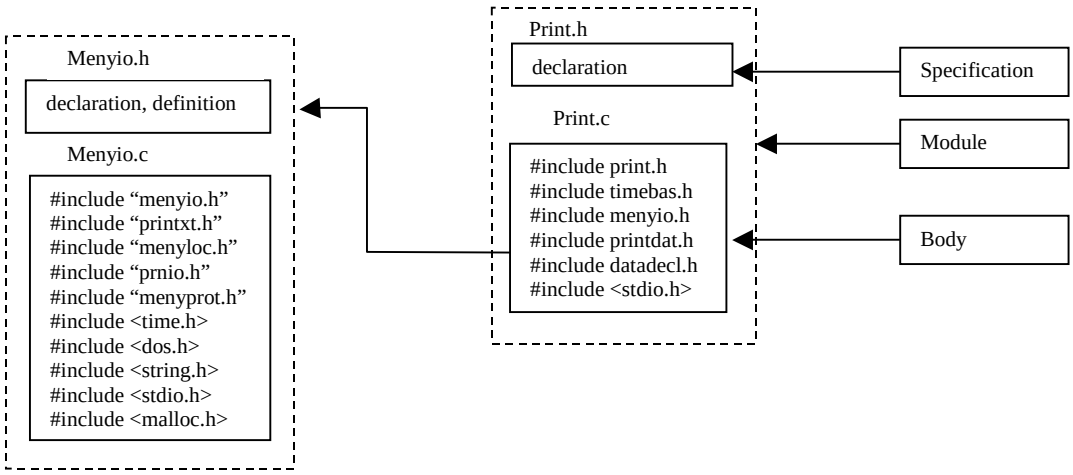


Figure 6. An example of modules and the connection between them

4.3. Analysis of the existing system

When choosing an approach for further analysis of the existing system, considerations are taken to the ability to support the functional requirements of the new system (see Appendix D). An application analysis is performed (see 2.2.3). This approach is chosen to find and eliminate files, which are of no interest compared to the functional requirements and the new systems programming language. The reduction of source code is also made for making the source code easier to comprehend. Investigating unfamiliar source code deeper, without any type of tool is considered to take too much time. Furthermore, it could even be a waste of time if the source code is found to be of no appraisal use.

At this stage, overall understanding of how to provide functional fulfilment and how to improve the technical quality of the source code are the primarily goals. All source code is briefly examined and reduction decisions are made after overall comprehension. Making quick decisions is not considered to be a risk, because rejected source code will always be reachable after reduction. When making the examination, discovering that programming

styles differ considerably, confirms that several programmers have been involved in development and maintenance.

4.3.1. Reduction of remaining files

Application analysis has two attributes to take in consideration; ability to support the system's functional requirements and system design and use of technology. These two attributes provide insight, in what to improve in the system's re-engineering process. To obtain this, further analysis is performed to identify all components in the source code. After this, irrelevant files are excluded for several reasons.

- Files, describing the interface in the existing system are not significant, because the interface in the new program is to be developed in Visual Basic. The source code could help when creating a similar interface as the existing, but this is not a requirement and therefore not included. This leads to considerable reduction of files.

- Files, including the code for communication between the existing system and the old printer, are written in a complex manner. Because of the complex structure and that the old printer is not used in the new system, files related to the printer are excluded.

- One of the two text files describing the system is out of date and therefore excluded, due to many following updates of the system. The other text file describes how to register a new article number in the existing system and is also of no use. Hence, both are reduced and are not taken in further investigation.

- There are two files that not are incorporated in the program. These files are most likely files used during earlier development and therefore excluded. They are called A.c and New.c. The names and contents indicate that they are some sorts of test files.

After the reduction of files, only 4 files remain.

4.3.2. Investigation of remaining files

One of the remaining 4 files is a data file, used to store information between printing occasions. Information stored in this data file, is for example, number of cards produced for a specific article number.

The other three files describe which article numbers are used in the system, information about the way they encode cards, and the printed text on the cards. The three files include 149 article numbers (see Appendix E). Almost every article number refers to a particular card to be printed. A few of the article numbers refer to sets of cards. A set article number refers to other article numbers. Article numbers referring to single cards have information about; 25 hexadecimal character code, the coercivity (see Appendix A – Glossary), the text printed on the card, what type of card to insert

into the printer, at which track (see Appendix A – Glossary) to encode the card, and additional information. The additional information is special information regarding the specific card (see Appendix G for a source code example).

To collect and store all information the data re-engineering activity (see 2.3.4) is used. The information is stored in a database and used when the new program will be developed. The database handler used is Microsoft Access as it works together with Visual Basic.

The article numbers differ in type of magnetic encoding (see Appendix A – Glossary) and text printing. To be able to get an overview of them, they are divided into different categories (see Appendix E). Three main categories of article numbers are discovered; cards with fixed code and text printing, cards with text printing only, and finally, cards with variable encoding and text printing.

The main part of the article numbers are represented as separate functions in the source code, which results in a large amount of similar code at several locations in the program.

The article numbers, that handle cards with variable encoding (different encoding at different coding occasions), have complex algorithms and these article numbers need to use a data file to store and retrieve information. The data file also handles the enumeration for certain article numbers. The enumeration consists of the sequence number (totally printed cards), print number (the number printed on the card) and code number (the number encoded on the magnetic strip).

4.4. Development of the new design

When starting developing the new design there are a lot of issues to take into consideration. The approach is to make a design recovery (see 2.2.5) of the existing system. The functional requirements from Timelox (see Appendix D) and the requirements specification (see Appendix H) have to be taken in account when making the design recovery.

When the design recovery is made, design issues for the new design are taken into consideration. Some of the issues are; how the new design has to be effective, simple, easy to maintain, object oriented and organised. UML is used to design the new system.

The documentation produced from the design recovery phase provides information about; main components of the system, technology used to provide system functionality, other applications that use the system and interface to other system.

4.4.1. UML diagrams

To get a better understanding of how to design the program, UML diagrams (see Appendix F) are created [2, 13].

In the UML use case model, two actors are identified, the administrator and the user. The administrator has authority to add, remove or make updates in the database, which is connected to the system. The database stores all relevant data from the existing system. The user gets information about card or card sets and print and encode cards, but the user has no authority to add, remove or make updates in the database.

In the UML class diagram a design model is described. This model is well structured compared to the module diagram (see Appendix B) that describes the existing system.

4.4.2. Incorporation of a database

To get an understanding over the entire article numbers, program structure improvements (see 2.3.2) and data re-engineering (see 2.3.4) are used.

A database is used to store information about the article numbers. The information about the article numbers is collected from the source code and from the data file.

In this way, it becomes easier and more effective to obtain and store information. Other advantages are that the information is more structured and it is easier to get an overview over the contents of the system. Furthermore it becomes easier to make general functions connected to the database instead of separate functions for each article number. This reduces the need for changing the source code, when adding a new article number in the system.

5. Analysis of the case study

To show results and knowledge of the case study an analysis of the study is performed. In addition, recommendations for improvements are discussed. The method used to re-engineer the system (see 3.4 Method) is divided into different phases (see 2.2.1 – 2.2.5) and activities (see 2.3.1 – 2.3.5). Several of these phases and activities were taken into consideration. Furthermore, a diagram (see Figure 7) is presented. This diagram shows the phases and activities included in the case study.

5.1. Analysis of activities and phases used

In the first phase of the re-engineering process, preliminary inventory analysis (see 2.2.1), a manual reverse engineering (2.3.5) is performed. This is performed when deriving the systems design and specification from its source code and reorganising its structure.

The documentation about the existing system turned out to be inferior. There are approximately 15000 lines of source code in the existing system but only 5 pages of documentation. This is the reason why the reverse engineering activity was very time-consuming.

The next phase, encapsulation (see 2.2.2), is also time-consuming due to the lack of documentation. The activity in this phase is program modularisation (see 2.3.3). Difficulties are to understand how the source code's modules are connected to each other. Furthermore, inexperience about syntax and construction of the function-oriented language C makes this activity time-consuming.

In the application analysis, (see 2.2.3) two activities, program structure improvement (see 2.3.2) and data re-engineering (see 2.3.4) is performed. These two activities highly increase the understanding of the existing system, due to a better system overview.

The last phase performed is design recovery (see 2.2.5). The approach using CASE-tools in this phase is rejected after discussions with responsible personnel at Timelox AB. The reasons are; it will take too much time and effort to get an understanding of a new application. In addition, it is not financially motivated to buy a CASE-tool that will be used for just one special occasion. Earlier studies also show that it is difficult to find a suitable CASE-tool, due to the various forms of source code [10].

5.2. Analysis of activities and phases not used

The production standardisation phase (see 2.2.4) is ignored in the case study. The reason is that the new system is developed in an object-oriented manner and it is hard to apply on the existing system

The activity, source code conversion (see 2.3.1) is not adequate because of the increased possibilities provided by a more modern language as Visual Basic [1, 8]. The problems are that the syntax and the semantic differ considerably; this makes it impossible to make a code conversion between the programming languages [8].

Phases / **Activities**	Preliminary inventory analysis	Encapsulation	Application analysis	Production standardisation	Design recovery
Source code conversion					
Program structure improvement			**X**		
Program modularisation		**X**			
Data re-engineering			**X**		
Reverse engineering	**X**				**X**

Figure 7. Phases and activities included in the case study.

5.3. Experiences

In the beginning of the case study much time was devoted to the documentation of the existing system. The documentation turned out to be inferior, but some assistance was taken when the specification and design for the new system was developed.

With insufficient or lack of documentation, the time spent to get knowledge of the system increases. This results in higher costs. An experience report shows that producing the necessary architectural documentation during the recovery project costs eight to twelve times as much as producing the same set of documentation during the original development project [10].

A recommendation, when starting up a case study, is to first make a survey of the existing documentation, and if the documentation is inferior, not spend more time trying to understand the existing documentation. The problem is to realise that the documentation is inferior. It is not an easy decision to ignore the existing documentation and instead concentrate at the existing source code. This type of decision is probably easier to take with more knowledge and experience of similar case studies.

In case study the main focus was deriving how the cards were encoded and what was printed on the cards. With an inferiorly documented existing system it is hard to form an understanding of the system, without making a time demanding survey of the source code.

If a survey of the source code is performed, it is important to get an overview over the system and try to ignore irrelevant code, which are of no interest in the new system. Source code of no interest is code, which handle the graphical interface, and code, which communicate with I/O components. This code, in the new system, is handled by Visual Basic.

When the new system is to be developed, it will be possible to decrease the amount of source code. The reasons are that the existing system does not have a database and generally algorithms in the source code for the article numbers.

6. Conclusions and summary

Software activities are predicted to move more towards maintenance and improvement, rather than development of new software (see chapter 2). Maintenance and improvement can, for several reasons, include changing the programming language, which makes it more difficult because of programming language differences. When changing programming language, the wish is to take as much benefit as possible from existing software code.

6.1. The concept of software re-engineering

Literature studies showed that this is a well-investigated area. Several of the articles and books in the reference list have their own definitions of the concept re-engineering and describe smaller areas (for example reverse engineering and code conversion) in different ways. This makes it sometimes difficult trying to separate the different parts and place them in their right surroundings; though the overall definition is similar. No unified work-processes are found, but instead several recommendations of how to proceed in the process.

The concept of re-engineering is not just about translating source code, but also about more general program improvements. When changing software, these improvements are incorporated in a natural way.

6.2. From function-oriented to object-oriented

In literature it is claimed to be complicated (even impossible) to make a conversion from a function-oriented programming language to an object-oriented programming language [1,4,7]. The main reason, for the difficulties between these conversions is, besides syntactic differences, the entirely different program structure.

Observations showed that the programming structure differed completely in Visual Basic compared with C (see section 3.5.3). The results of these observations are, that it is impossible to implement any of the source code from the existing system and transform it to the new program.

It would be interesting though, to investigate the process when moving from C to C++ and afterwards to Visual Basic. The reasons are that C++ is a similar language to C regarding syntax, data declarations etc., but differs in its semantic, because C++ is an object-oriented language like Visual Basic.

Modern programming languages with supporting technology such as support for user interface drawing and possibility to make easy connections to common database handlers such as Microsoft Access has made it easier when updating software systems. Moving from Visual Basic to C without a database handler would probably be much more difficult. Besides handling the database, the graphical interface has to be dealt with.

6.3. The case study

Making a case study on an existing obsolete system turned out to be a good way for getting an overview of the difficulties with re-engineering (described in chapter 5). The size of the system used in the case study was manageable without using analysing tools. Not using any tool gave a sense of having good control of the process. If the existing system would have been larger, the use of tools would most likely been necessary.

The module diagram (see Appendix B) created of the existing system is not used after it is created, which might indicate that unnecessary work is made. Although, it is not used in further work the actual making of the module diagram provides information about the existing system's structure.

The result of the case study shows that understanding the system is the main benefit in this particular case. It is possible that certain algorithms can be translated and used in the new system. Most probably though, is that these algorithms not will be used as they are, but instead used for getting information.

6.4. Meeting the objectives

The first objective, in this thesis, is to obtain knowledge to what extent it is possible to reuse code. Due to the possibilities to ignore big parts of the code from the existing system, it turned out, that understanding the existing system's functionality is the main benefit when meeting this objective.

The code, possible to reuse, are data that are implemented in the database for the new system and also some specific algorithms, which explain, how card encoding works. Consequently, instead of actual reuse of code, data collection and function descriptions are benefits when meeting this objective.

The second objective, understanding difficulties when moving from a function oriented programming language to an object-oriented language, could not actually be examined. The reason is that large amount of code is excluded from the system.

Instead, other difficulties exposed are; identify important parts of the system, obtain an overall picture of the unstructured system, and decide if the system documentation is reliable.

7. References

[1] Ian Sommerville, *Software Engineering,* Addison-Wesley, Sixth edition, 2001, ISBN 0-201-39815-X

[2] Perdita Stevens and Rob Pooley, *Using UML, Software engineering with objects and components,* Addison-Wesley, 2000, ISBN 0-201-64860-1

[3] Howard Wilbert Miller, *Legacy Software Systems,* Digital Press, 1998, ISBN 1-55558-54195-1

[4] Linda Wills and Philip Newcomb, *Reverse Engineering,* Kluwer academic publishers, 1996, ISBN 0-7923-9756-8

[5] Mats R. Gustafsson and Lars-Åke Johansson, *Metodik för reverse engineering/ reengineering - ett eftersatt område,* Svenska institutet för systemutveckling, Rapport nr 17, Oktober 1994

[6] Elliot J. Chikofsky and James H. Cross II, *Reverse Engineering and Design Recovery: A Taxonomy,* Auburn University, USA, 1990

[7] Arie van Deursen, Paul Klint, and Chris Verhoef, *Research Issues in the Renovation of Legacy Systems,* In J-P Finance, Fundamental Approaches to Software Engineering, FASE99, LNCS, pp 1-23, Springer-Verlag, 1999

[8] Andrey.A Terekhov and ChrisVerhoef, *The Realities of Language Conversion,* Faculty of Mathematics and Mechanics, St. Petersburg State University, Russia, 2000

[9] Alfs T. Berztiss, *Reverse engineering, Re-engineering, and Concurrent Engineering of Software,* Department of Computer Science, University of Pittsburgh, 1995, ISSN 1101-8529

[10] L. Bratthall and P. Runeson, *Architectural Design Recovery of a Family of Embedded Software Systems - An Experience Report,* Proceedings First Working Conference on Software Architecture (WICSA1), San Antonio, Texas, USA, pp. 3-14, February 1999

[11] Eric Winemiller, David Jung, Pierre Boutquin, John Harrington, Bill Heyman, Ryan Groom, Todd Bright and Bill Potter, *Visual Basic 5 Super Bible set,* Waite Group Press, 1997, ISBN 1-57169-111-1

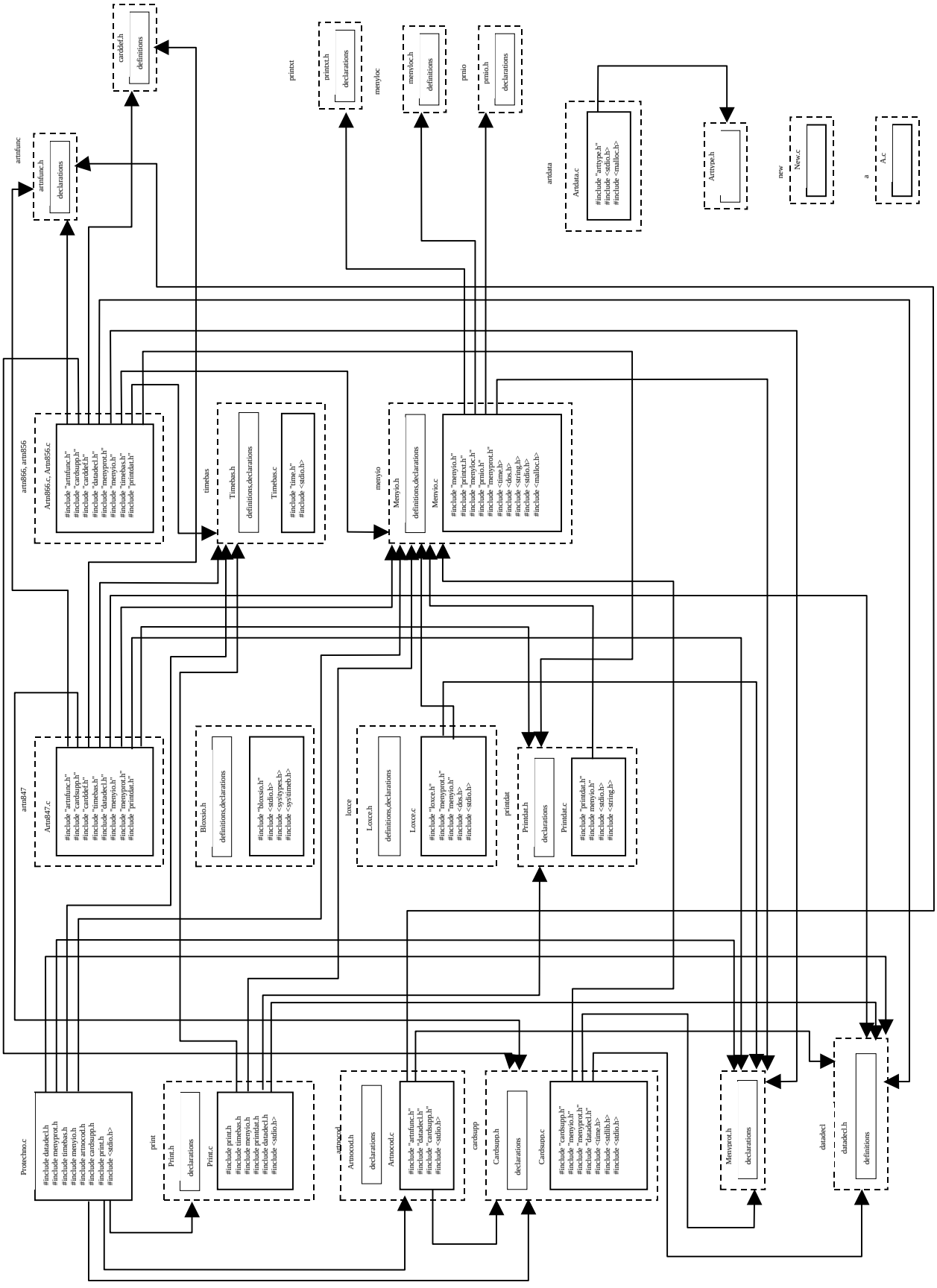
[12] Ulf Bilting and Jan Skansholm, *Vägen till C,* Studentlitteratur, Second edition, 1990, ISBN 91-44-26732-0

[13] Terry Quatrani, *Visual Modeling with Rational Rose 2000 and UML,* Addison-Wesley, Second edition, 2000, ISBN 0-201-69961-3

8. Appendix A - Glossary

Coercivity	A technical term used to designate how strong a magnetic field must be to affect data encoded on a magnetic stripe. Coercivity is measured in Oersteds (Oe). A more precise definition is; the intensity of the magnetic field needed to reduce the magnetization of a ferromagnetic material to zero after it has reached saturation.
DBMS	Short for “Database Management System”. A DBMS is a computerised record-keeping system that stores, maintains and provides access to information.
Magnetic encoding	Magnetic stripe cards have been in existence since the early 70's. Magnetic stripe technology is widely used throughout the world and remains the dominant technology for transaction processing and access control. The stripe consists of tree different tracks, where one or more are encoded with high or low coercivity.
Track	There are tree tracks on a magnetic stripe, which can be encoded. These are defined by ISO Standards regarding for instance location, character data size, and bit recording density. Bit Recording Density, BPI (Bits per Inch) default ISO Standard selections are 210 BPI for track 1 and 3 and 75 BPI for track 2.

9. Appendix B – Module Diagram



10. Appendix C - File description

Source code files

Protechn.c	Main module with head menu and calls to functions according to what is chosen in the menu
Print.c	Function for card printing
Artnocod.c	Function for card encoding and card printing according to article number
Cardsupp.c	Definition of functions, which are used in Artn847.c, Artn856.c and Artn866.c
Artn847.c	Definition of functions for article number 847xxx, with fixed and variable code
Artn856.c	Definition of functions for article number 856xxx
Artn866.c	Definition of functions for article number 866xxx, with fixed and variable code
Bloxio.c	Functions that communicate with the kernel
Loxce.c	Functions that communicate with the kernel
Printdat.c	Functions that communicate with the Protechno printer
Timebas.c	Functions that handle time delays and show time on screen
Menyio.c	Functions that handle input and output from menus
Artdata.c	Functions that handle the communications with Artno.dat
A.c	Functions that not are incorporated in the program
New.c	Functions that not are incorporated in the program

Header files

Artnfunc.h	Declarations of functions from Artn847.c and Artn866.c
Arttype.h	Declarations of functions from Artdata.c
Artnocod.h	Declarations of functions from Artnocod.c
Print.h	Declarations of functions from Print.c
Cardsupp.h	Declarations of functions from Cardsupp.c
Menyprot.h	Declarations of functions from Menyio.c
Datadecl.h	Definitions of constant values, which are used in the program
Bloxio.h	Definitions and declarations of functions from Bloxio.c
Loxce.h	Definitions and declarations of functions from Loxce.c
Printdat.h	Declarations of functions from Printdat.c
Timebas.h	Definitions and declarations from Timebas.c
Menyio.h	Definitions and declarations from Menyio.c
Carddef.h	Declarations of constant values for text
Printxt.h	Declarations for menu texts and information about every article number
Menyloc.h	Definitions of digit buffers
Prnio.h	Declarations of functions from Menyio.c

Text files

Manual.txt	A manual how the menus in the program works
Program.txt	A short description about the file structure in the program

Data file

Artno.dat	File for storing changes between coding and printing

11. Appendix D - Functional requirements from Timelox

It shall be possible to feed an article number for an individual card or a set of cards into the new application. The user of the application shall get information if the card uses high or low coercivity.

The application shall run against some kind of text file or database, which contains all information about the stored article number. Every article number holds information about the text to be printed on the card, the code to be printed on the magnetic strip, the cards coercivity and the card type. Every card set has an article number, which refers to the specific cards in the set.

The application shall be able to encode the magnetic cards in two different ways. Either with the same code regardless of the numbers of cards or cards consisting of numbers in sequence and/or randomised numbers. In the case with numbers in sequence the number should be read from the database/text file and rewritten and saved with its new value in the database/text file. Finally there are some cards, which are encoded with more complex algorithms.

There shall be a function in the application that makes it possible to print optional text in optional positions on the card.

12. Appendix E – Categories of article number

	Quantity
FIXED	87
FIXED-SET-DEPENDENT	2
	89
PRINT	20
PRINT-DEPENDENT	2
PRINT-SET	1
PRINT-SET- DEPENDENT	1
	24
VARIABLE	8
VARIABLE- DEPENDENT	6
VARIABLE- DEPENDENT-NOPRINT	2
VARIABLE- INTEGRATED	2
VARIABLE- NODAT	1
VARIABLE -SET-DEPENDENT	7
VARIABLE- SET-DEPENDENT-IDENTICAL	5
VARIABLE- SET-TWO	2
VARIABLE- TRACK2	3
	36
SUM	**149**

Description of main and sub categories

FIXED	Main category, handles article number with fixed code
PRINT	Main category, handles article number with text printing
VARIABLE	Main category, handles article number with variable encoding
SET	Sub category, handles set of cards
DEPENDENT	Sub category, dependent of other article numbers
NOPRINT	Sub category, encodes the card but does not handles text printing
INTEGRATED	Sub category, an article number that is integrated into another article number
IDENTICAL	Sub category, an article number that handles identical cards in a set
TWO	Sub category, handles two cards with variable code
TRACK2	Sub category, handles cards encoded on track 2

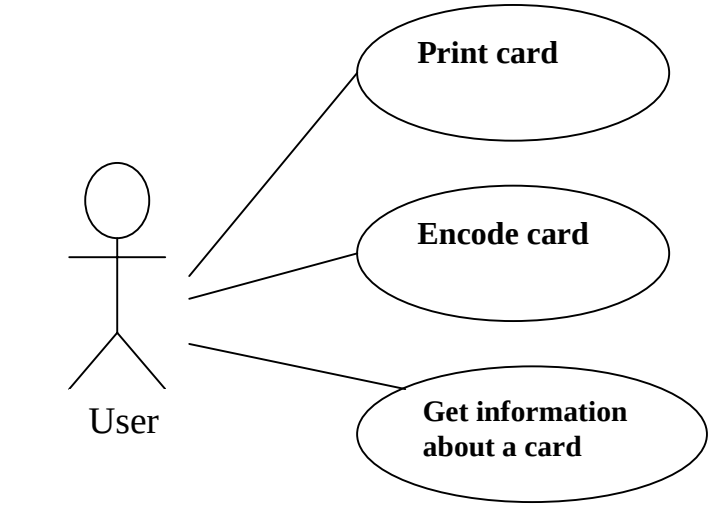
13. Appendix F – UML diagrams

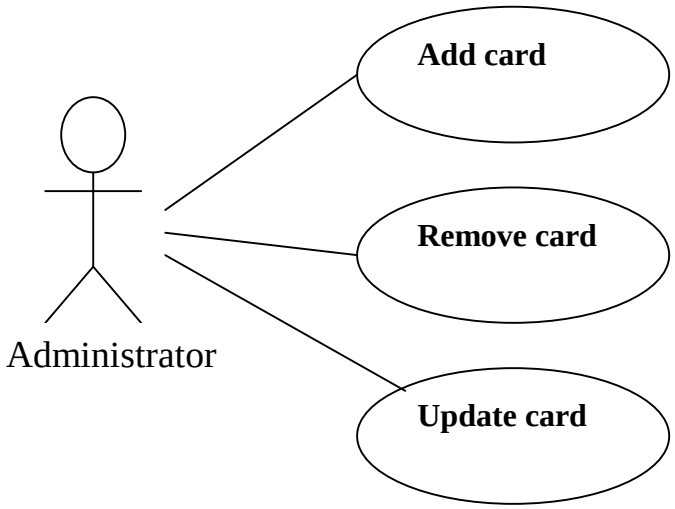
Unified Model Language (UML)

When investigating how a program is constructed, it is important to get an understanding of the design and architecture of the entire system. One approach is to use Unified Model Language (UML) [2]. UML is a diagram-based design language. In UML it is possible to construct different models of design, because it is inconceivable to capture everything about the design in a single type of diagram. Two types of UML diagrams used in the case study are:

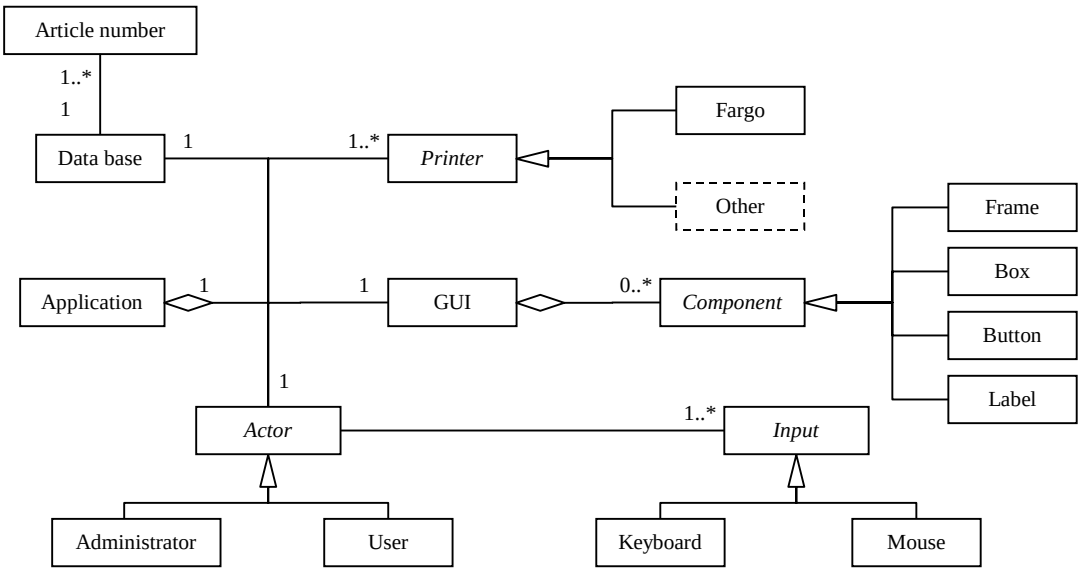
- The use case model describing the required system from the users point of view.
- The class model describing the elements of the system and their relationships.

UML use case diagram





UML class diagram



Classes

Database	Handles all information about the article numbers
Printer	Prints text on the card and encode the magnetic strip
Fargo	The specific printer used
Other	Possible to use other printers
GUI	Handles the graphical user interface.
Components	Super class for the components in GUI
Application	Starts and executes the program
Actor	Super class for User and Administrator
User	Uses the program. Has no authority to make changes in the database.
Administrator	Administrate the program and the database.
Input	Super class for Keyboard and Mouse
Keyboard	Handles the input when the user types on the keyboard
Mouse	Handles the input when the user uses the mouse

14. Appendix G – Source code example

Example of an article number in the existing system's source code:

```
/*****************************************************************
*  Name:       fno_847501
*  Function:   Generates an "Emergency 0"
*  Input:      Int number.
*  Output:     Int ready (errorcode).
*  Uses:       screen_handler,display_number_of_cards ( CARDSUPP.C ),
*              card_write  ( LOXCE.C ),
*              printer f. (PRINTDAT.C) and  functions from MENYIO.C.
*****************************************************************/
int fno_847501 (int number)
{
  int index = 0, ready = CARDCODING_OK, no_of_cards, radindex, rad = 14, kol = 3;

              no_of_cards = screen_handler (TEXT_847501, &number, 847501);

              while (index < number)
              {
              display_number_of_cards (number-index);

              if (fnobusy())
                            return ready;
              delete_bitmap();
              set_horizontal();
              select_character(02);
              for (radindex = 0; radindex < rad; radindex++)
                            line_feed();

              send_text("EMERGENCY-0",kol);
              send_text("  847 501  ",kol);


ready = card_write ("E0039BA500100002546000000",CARDLENGTH, 0 );
if (fnobusy())
                            return ready;
                            if (ready == CARDCODING_OK)
                            {
                                          proprint();
                                          delay(6);
                                          index++;
                            }
                            else
                            {
                                          reset_printer ();
                                          delay(10);
                            }
              }  /* while */
              return(ready);
}
```

This is what will be printed on the card.

This is what will be encoded on the card.

15. Appendix H – Requirements specification

Kravspecifikation

1. **Funktionella krav**

Generellt

1.1 Korten ska skrivas ut på en skrivare.

Användaren

1.2 Användaren ska genom inmatning kunna ange ett artikelnummer för ett enskilt kort i programmet.

1.3 Användaren ska genom inmatning kunna ange ett artikelnummer för ett set av kort i programmet.

1.4 Användaren ska genom artikelnumret få information om vilka slags kort som ska laddas i skrivaren.

1.5 Användaren ska genom en fråga från programmet ange hur många kort som skall tillverkas.

1.6 Användaren ska genom en fråga från programmet ange hur många set av kort som skall tillverkas.

1.7 Användaren ska kunna trycka valfri text på valfri position på kortet.

Programmet

1.8 Programmet ska hämta information via en databas eller en textfil.

1.9 Programmet ska skicka information till en databas eller en textfil.

1.10 Programmet ska kunna koda magnetkort med fast kod d.v.s. koden som ska ligga på korten är samma oavsett hur många kort som ska kodas.

1.11 Programmet ska kunna koda kort som innehåller löpnummer.

1.12 Programmet ska kunna koda kort som innehåller slumptal.

1.13 Programmet ska kunna koda kort som innehåller löpnummer och slumptal.

1.14 Programmet ska kunna koda kort med komplicerade algoritmer.

1.15 Programmet ska kunna hantera plastkort med hög koercivitet och plastkort med låg koercivitet.

1.16 Varje artikelnummer ska bestå av 6 siffror.

1.17 Programmet ska räkna upp eller ner löpnummer per artikelnummer i specificerad algoritm.

Administratör

1.18 Administratören ska kunna ändra, ta bort och lägga till artikelnummer i databasen.

Databasen/Textfilen

1.19 Databasen eller textfilen ska lagra all information per artikelnummer

1.20 För varje kort ska det finnas information på databasen eller textfilen om vilken text som ska tryckas på kortet.

1.21 För varje kort ska det finnas information på databasen eller textfilen om var texten ska tryckas på kortet.

1.22 För varje kort ska det finnas information på databasen eller textfilen om vilken kod som ska läggas på magnetremsan.

1.23 För varje kort ska det finnas information på databasen eller textfilen om kortet ska vara med hög eller låg koercitivitet.

1.24 För varje set av kort ska det finnas information på databasen eller textfilen om vilka artikelnummer som ingår i setet.

1.25 För varje kort med löpnummer ska aktuellt nummer lagras i databasen.

2. Icke funktionella krav

2.1 Programmet ska skrivas i Visual Basic

2.2 Programmet ska vara objektorienterat

2.3 Programmet ska vara utvecklat för Microsoft Windows plattform

2.4 Programmet ska vara enkelt att underhålla.

2.5 Programmet ska vara enkelt att bygga ut.

2.6 Programmet ska kunna anslutas till en annan skrivare utan större modifieringar.