DEGREE PROJECT 2004:PM07

Department of Technology, Informatics and Mathematics

Requirements Engineering and Prototyping in a Legacy Software Setting

Anders Norgren

HÖGSKOLAN
TROLLHÄTTAN • UDDEVALLA

DEGREE PROJECT

University of Trollhättan · Uddevalla
Department of Technology, Informatics and Mathematics

Degree project for master degree in Software engineering

Requirements Engineering and Prototyping in a Legacy Software Setting

Anders Norgren

Examiner:	
Doctor Steven Kirk	Department of Technology, Informatics and Mathematics
Supervisor:	
Junior Lector Jonas Amoson	Department of Technology, Informatics and Mathematics

Trollhättan, 2004

2004:PM07

EXAMENSARBETE

Requirements Engineering and Prototyping in a Legacy Software Setting

Anders Norgren

Sammanfattning

Programvaruutveckling i en miljö där det finns gammal programvara kan påverka programvaru-utvecklarens öppenhet för alternativa lösningar. I denna rapport kommer detta att studeras likaväl som projektet. Projektet går ut på att dokumentera den gamla källkoden och ta fram krav på nästa generation av en programvara som är producerad och använd i en industriell miljö. Användandet av prototyper under kravhanteringsprocessen kommer också att diskuteras.

En sak som kommer att tas i beaktande under detta arbete är om dokumenteringen av den gamla källkoden påverkar programvaru-utvecklarens öppenhet. Resultatet visar med stor sannolikhet att programvaru-utvecklarens öppenhet för alternativa lösningar inte påverkas, trots den gamla miljön, utan istället får hjälp från den gamla källkoden. Dessutom kommer fördelen med en formell kravhanteringprocess och användandet av prototyper att demonstreras.

Preface

This paper is the result of a degree project of the special year of software engineering at the University of Trollhättan/Uddevalla. The accomplishment of the paper would not have been possible without the support from my examiner Dr. Steven Kirk, my instructor Junior Lector Jonas Amoson and last, but not least, my wife Kate Norgren, who’s support made all this possible.

Requirements Engineering and Prototyping in a Legacy Software Setting

Anders Norgren
Dept. of Computer Science
University of Trollhättan/Uddevalla
P.O. Box 957, SE-461 29 Trollhättan, Sweden
anders@norgrens.se

Abstract

The process of software evolution in a legacy software setting might affect the developer's open-mindedness. In this paper we will study this issue during the project as well as the project itself. In this project the tasks are to document the old source code and elicit requirements for the next generation of a piece of software produced and used in an industrial setting. The use of prototypes in the requirements elicitation process will also be discussed.

One issue considered in this work is the question of whether the documentation of the old source code affects the open-mindedness of the developer. The result shows that the developer most likely will not become narrow-minded because of the legacy setting, but instead gets help from the legacy code. In addition, the effectiveness of having a formal requirements engineering process, and also of prototyping, is demonstrated.

1. Introduction

Software evolution often means that bugs are fixed and functions are updated and/or added. In order to add new functions there must be an investigation in order to find out what additional functionality is wanted by the users. Rugaber's [1] interpretation of software evolution is "the process of adapting an existing software to conform to an enhanced set of requirements". Therefore the process usually starts with requirements engineering to sort out what the users want from the next generation of the software.

Requirements can be divided into three groups [2]:

- User requirements are what the users want from the system.
- System requirements are about the system services and constraints.
- Software design specification is an abstract description of the software design.

This paper only looks at user requirements towards a new version of a software package. The software under investigation in this study was originally developed in the early 1990s and therefore this work is performed in a legacy software setting.

Sommerville [3] writes about legacy systems and the risks involved with replacing the system or keeping it and continue to update it. Some risks that he mentions are:

- Almost never a complete specification of the legacy system.
- Business processes and the operations of the legacy system are much intertwined.
- The legacy system has business rules that are not documented elsewhere.
- System documentation is incomplete and out of date.
- Many years of maintenance has corrupted the system structure.

When working in a legacy software setting it is possible that the developer will be influenced by the old software. This is a matter that the author will discuss from an empirical point of view after having participated in this project.

The focus of the case study in this paper is a part of the legacy software used concerned with calculation and the work is divided into three parts. The first part is to analyze and document the current source code for the calculation. The second part is to gather requirements from the users for new input to the program and the third part is to analyze and construct a new algorithm for the calculation. One question to be answered is if a legacy software setting is helping or hindering the requirements engineering process.

2. Background

The client company required secrecy and therefore some cryptic names will be used, both for company names and products. Also method names, variables and other concerning the source code will be changed.

A study will be performed on a specific part of the software, Sware, used by the company CompX. CompX manufactures a product PROD that comes in three different types, A, B and C. Each type comes in different sizes and configurations. Sware is used by CompX to list the most feasible variants of PROD that suits a customer's need.

When using the software the first thing to do is to select which type will be used. Then the user enters all the necessary parameters, after which the software excludes all variants that do not fit the requirements, calculates a rating value for each variant, and then sorts the result based on the rating. The user then selects the appropriate variant based on the sorted results and his experience.

This software was originally designed back in 1992 and has been updated several times since then. The development, programming and updating of the software has been performed by an external company Gsoft. CompX does not have any real documentation of Sware and are now facing another update of Sware.

The main reason for the update of Sware is that the distribution of Sware is out of date, because it uses diskettes or CDs to distribute updates of the databases, according to Svensson, a product specialist at CompX and who also was in charge for the main project. This is due to the fact that the software is locally installed on a user's computer. CompX is primarily thinking of the new version of Sware as a web based solution with the possibility to run Sware on a local computer with no Internet connection, and has therefore started up a project for this. At the same time an update of the functionality of Sware can be considered since the code needs to be rewritten anyway. This paper only covers a small part of that project.

3. Methodology

The methods that will be used in this paper will differ depending on the subject. As mentioned this work is divided into three parts that will each require a different method.

3.1. Documenting the source code

The first part is the documentation of the old source code. The possible methods here are to document the code line by line, i.e. how the code does the work, or in chunks, i.e. describing the intended purpose of the code. Since the focus of the project is to develop a new version the choice fell on the latter approach, since this gave an understanding of what the original requirements could have looked like. This topic will be discussed further in Section 4.2.

3.2. Requirements engineering

The next part is the requirements engineering, and there the choice was to do the requirements elicitation alone or together with the users. The choice fell on doing it together with the users on two reasons. The first reason is to, as Antón [4] writes, understand the problem, and since the author's knowledge about the products was inadequate, help from the users was necessary. The users could provide the required understanding. The second reason is that, as stated by Clavadetscher [5], user involvement is the key to success. Only the users know what they want, the developer can only guess. More information about this method will be given in Section 5.3.

3.3. Prototyping

The last part is the prototyping and here the choice was simple - use a graphical development environment to design a user interface and use that as the base for the discussion on how to gather the required input for the calculation. More about the use of a graphical development environment will be given in Section 6.2.

3.4. Layout of this paper

First the documentation of the source code will be described. The next step is the requirements engineering which will be divided into several subsections like introduction, gathering and analysing the requirements. Thereafter the prototyping will be in focus followed by the result. This will be followed by a discussion, conclusion, and some recommendations for further work.

4. Documenting the old source code

4.1. Overview

The source code was written in C++ [6, 7] and was lacking comments throughout the entire code. Furthermore, there was no specification of the legacy software, and that was a drawback in the process of understanding how the software worked. The only existing documentation was a document from 1998 that covered a part of the code for the calculation. This was helpful to get a quick understanding of how the calculation was performed. In addition, a guided tour of the program provided in a face-to-face discussion by a representative of the client company helped understanding the functionality of the program.

The source code for the calculation consisted of a total of eighteen files consisting of nine header files and nine with the actual code. A header file contains structure declarations and prototypes for the functions that use those structures [6, 7]. The source code contains the code for structure related functions and the code that calls those functions. Of the nine files of code, there were six that handled communication with the database, and three that dealt with calculating and sorting the appropriate products. Therefore there were only three files of interest in the documenting process.

To give an overview on how big the source code files are their size will be stated using lines of code (LOC), which is commonly used as a productivity measurement, but that is not the issue here. LOC does not say anything about the complexity of the source code since there can be more than one statement per line or a statement can be written over several lines [8]. These numbers are simply stated to give the reader some idea of the size of the codebase used in this work.

4.2. Methods

The first thing to do when documenting the source code is to decide how to document it. There are three possible ways to perform the documentation:

- To document on a line-by-line basis thus documenting how the code works.
- To document chunks of code which documents how the code works.
- To document the structure of the code, i.e. how classes and methods are connected.

When considering that the next version of the software will most likely change platform from local application to web application there will probably not be much reuse of the old code, which has more than twelve year old roots. In that light, the first choice seems to be overkill, since the likelihood of the old code to be reused for a (presumed) web based solution is probably very small, and the third choice too shallow, due to the fact that more is needed than just the structure of the code to construct a program with similar functionality, leaving the second choice as the most appropriate candidate.

By documenting chunks of code there will be additional beneficial effects. Not only will the functionality of the code get documented, but also the structure. Another effect will be a reverse engineering of the requirements that lead to the code in the current form.

The next decision is how to perform the analysis of the source code. It could be performed manually or by using some sort of tool. In this case, 'manually' means going through the code 'by hand' and noting how it works. By using some sort of tool like CScout [9], Cbrowser [10] or Source-Navigator [11], the relationships of the classes and methods could be presented graphically. This kind of tool supports editing, searching and browsing the source code. Since the amount of code was only three files and the need to know the connectivity was not important the choice fell on manual analysis. Another reason to perform the analysis manually was that there was no need to learn a new tool since the amount of source code was small and this project was time limited.

A sample of the documented code is provided in Appendix 1.

4.3. File one — PROD A

The first file to be documented was the most complicated one since it contains the calculation of rating. This file was quite extensive (1023 LOC) but it was also this file that had been documented in 1998. This documentation was not complete according to its writer. That the file was previously documented was of great help in the process of documenting it and the other two files. This file had some methods that were identical with some methods in the other two files.

This file also contained an error in the calculation. This error turned out to be minor, after consultation with engineers and product specialists. After consulting with Gsoft about the error, they corrected the error and provided a new version of Sware. This shows that it is important to have the source code reviewed by others in order to eliminate as many errors as possible before releasing the software.

This file also contained a method never used by this file. It was used by the other two files.

The other two files were less complicated in comparison and therefore easier to document. These files did not have any calculation part at all since they were just sorting possible products according to the input.

4.4. File two — PROD B

File two and three were very similar, which is probably due to the fact that the products to which these two files were targeted are quite similar. File two contained eight methods besides those in common for all three files but two methods were never used by this file or any of the other two files. The size of this file was 875 LOC.

4.5. File three — PROD C

File three was the easiest to document due to the fact that there were many similarities to the other two files. File three had eight methods besides those methods that were common for all three files. Of these six common methods there were four that were identical with some methods in File Two. Besides that, this file also contained the same two methods as File Two contained, and like them these two were never used by this file or any of the other two files. The size of this file was 848 LOC.

4.6. Difficulties encountered

During the documentation of the source code some strange things were encountered. First there were methods never used and methods commented out, see Section 4.6.1. Then there were variables created but never used, see Section 4.6.2.

By documenting the code, knowledge about what the user might want from the next generation of the software began to emerge, and that made it somewhat easier to do the requirements engineering.

4.6.1 Methods. File One had a method commented out. This method had a comment which said that it was intended for future use. This makes the actual source code only 998 LOC including blank rows, a difference of 25 LOC.

File Two had four methods commented out. One of these methods was identical with the method for future use in File One. The other three methods that were co-mmented out were duplicates of other methods in File One. The methods in File One were used instead of these methods. Besides these methods, there were parts of the code that were commented out. This changes the actual size of the file to 706 LOC including blank lines and the two methods never used. Without the two methods which are never used there were only 660 LOC, which gives a difference of 215 LOC in total.

File Three had the same four methods commented out as File Two. Even in this file there were parts of the code commented out, besides the methods, which makes the actual size of the code to be 655 LOC including blank lines and the two methods never used. Without the two methods never used there were only 609 LOC, which gives a difference of 239 LOC in total.

When there were methods commented out the code is harder to read since there is a lot of information to scroll by. Reading is made even more difficult when using an editor that does not support colour coding like Microsoft Windows Notepad.

4.6.2. Variables. For instance, there were variables created and initialised with a value from the user input but then the value was never used (see Listing 1). A mail was sent to Gsoft regarding these variables. Gsoft replied rapidly and the explanation was very enlightening:

The source code was based on an implementation of an ORB (Object Request Broker) by a specification of CORBA (Common Object Request Broker Architecture) [12] before there were any commercial implementations of CORBA. This meant that there exist defined types that can be different types at the same time. One example from the source code involves a type called `RValue` and its pointer `HValue`. There is a possibility to ask which type it is and the answer depends on what is stored in the variable. For example, if the variable is assigned an integer value (`tk_int`) then it can also be a `tk_float` or a `tk_string` and then use its type converter (`TFLOAT`, `TSTRING` and so on) on its `HValue` to receive a specifically typed variable.

```
int compareSomeText (HValue TestValue1,
HValue TestValue2){
  char StrTest1[10];
  char StrTest2[10];
  strcpy(StrTest1, (TSTRING) TestValue1);
  strcpy(StrTest2, (TSTRING) TestValue2);
  return strcmp((TSTRING) TestValue1,
(TSTRING) TestValue2);
}
```

Listing 1. Sample of the code where a variable is created and initialised but never used.

After receiving this information the source code was much easier to understand and the documentation process went smoothly. The explanation of the code in Listing 1 is that the debugger cannot read the value of a HValue directly. This means that the programmer has to set up some dummy variables to get around this problem.

4.6.3. The presence of 'unnecessary' methods. During the documentation process it turned out that there were a few methods in each file, that were not used any more or extremely rarely, which were discovered while consulting with Svensson. These methods were intended to allow the use of user assigned values, but there were some difficulties with the current system that led to that those methods become unused. In the next generation of Sware that functionality will be incorporated again.

5. Requirements engineering

5.1. Overview

The discovery of software system's requirements is a long and complicated process that must be considered extremely important in order to develop successful software solution [13].

There are usually two types of requirements that can be addressed; functional and non-functional requirements. A functional requirement has to do with the operation of the software, i.e. behavioural properties. A non-functional requirement has to do with everything apart from the functionality of the software. This can include consideration of performance, usability, reliability or supportability [14, 15, 16]. Sommerville [2] also mentions that the requirements can be classified as domain requirements, which are requirements that come from the application domain and reflects its characteristics. Harwell et al. [17] wrote a paper in which they stated the following about what a requirement is:

"If it mandates that something must be accomplished, transformed, produced, or provided, it is a requirement - period."

The Institute of Electrical and Electronics Engineers (IEEE) [18] has a more formal definition of the term 'requirement' in their Standard 610-12-1990 [19]:

1. *A condition or capability needed by a user to solve a problem or achieve an objective.*
2. *A condition or capability that must be met or possessed by a system or system component to satisfy a contract, standard, specification, or other formally imposed documents.*
3. *A documented representation of a condition or capability as in 1 or 2.*

In order to fulfil this task CompX provided this project with a group which consisted of users of the Sware and engineers. This group (UG, User Group) had experience of the products, both from constructing and selling the products, and their knowledge is therefore important in the process of requirements engineering. The users were able to give constructive criticism of Sware and the engineers helped with what was important to consider in the new program and possible to calculate or use as a sorting criteria.

5.2. The task

Since the focus of the task was on the calculation part only, there were no requirements on areas like hardware or software. That made the task easier, since all energy could be focused on the calculation part and which input to give and how the users wanted to enter the input. The old calculation part did more than filtering the PRODs and calculates the rating for the PRODs that were appropriate for the customer's application, it also sorted the output. Furthermore, the old calculation part differed between the variants A, B and C in that A had the rating calculated, but not B and C. After discussion with Svensson it was decided that this project should also do filtering, calculate the rating (only for PROD A) and sort the output in the calculation part.

5.3. Methods

The requirements analysis implements use cases, business processes or plain text to describe the functions required of the system [20]. Use cases [20, 21] are used in the Unified Modeling Language (UML) [20, 22], developed by Booch and Rumbaugh at Rational Software Corporation [23]. The practical work consists of discussions between the customer and the supplier [20].

Other possible methods to gather requirements are to do surveys or to do one-on-one interviews but these methods seems to be more time-consuming and require that the person conducting them has quite good knowledge about the target (product) area in order to ask relevant questions and not waste time by redoing the work with updated questions.

Therefore the approach taken in this work is to use 'use cases' to describe the flow in the calculation and the alternative paths that the users suggest.

Antón listed a few principles [4] that could help eliminate defects during the elicitation of requirements:

- *Understand the problem before expressing the requirements.*
- *Involve the stakeholders early on and sustain their involvement to validate the requirements.*
- *Ensure that critical requirements are considered.*
- *Give non-functional and quality requirements as much attention as functional requirements.*

By working closely with the UG most of these principles will be fulfilled. Gathering of the requirements [2] will be done by meetings with the UG where discussion can take place. Discussions offer the possibility to comment on pros and cons of ideas that are produced during a meeting. A possible downside with this approach is that the discussion can get out of hand and thereby make the meeting ineffective.

In order to find as many requirements as possible for the calculation, iterative development [24] will be used, meaning that the focus will initially lie on a small part and then adding more and more of the properties of the products. This approach offers the possibility to get a solid base under controlled circumstances, i.e. for each iteration the focus will be on the current problem but consider the whole system as the system grows.

The elicited requirements were documented in a requirement document according to the IEEE standard 830-1998 [25]. This is a wide standard for documenting requirements [2]. Another term for this kind of document is SRS (Software Requirements Specification) [25]. When writing the requirements document one must consider the language used so misinterpretations can be avoided as much as possible. Sommerville [2] talks about the hazards with natural language in the SRS, and one of these hazards is that it can give lack of clarity. It can be difficult to use language in a precise and unambiguous way without making the document wordy and difficult to read. In addition, natural language understanding relies on the specification readers and writers using the same word for the same concept. Another hazard with natural language is that it can be over-flexible for requirements specification.

To facilitate the use of use cases some graphical examples of possible graphical user interfaces (GUI) was developed using Java [26] and NetBeans [27] development environment. The reason for using Java and NetBeans are simple and few. Both are free to use and made the development of the GUI to go fast and smoothly. In Section 6 there are more details about why these GUIs were developed. These GUIs have nothing to do with the product when the project is finished; they were only used to make the communication within the group easier. Macaulay [28] writes that some projects fail due to bad communication or misunderstanding and that was the main reason to why these GUIs were developed.

A sample of a use case for PROD B is provided in Appendix 2.

5.4. Gathering the requirements

This is the most important step in this paper. Since the author does not have the same knowledge about PROD and its variants as the user group it was a steep hill to overcome. Therefore a close cooperation with Svensson took place. He demonstrated the program and how it works today.

In order to gather as many opinions as possible with as few people as possible Svensson had put together the group of users and engineers mentioned earlier. Several of these people were also involved in the rest of the project regarding the next generation of the Sware. This was discernible at the first meeting where the discussion got off-topic very often to discuss other angles towards the next generation of the Sware, i.e. things that did not concern the calculation part, like restrictions for different kinds of users.

At the second meeting everybody could not be present but the focus was never off-topic this time. This might be due to the fact that a statement consisting of the material for discussion in form of use cases and GUI examples from the first meeting, was sent out to everybody, but all had not received the material. These two meetings concerned product type A, but there were two more product types to discuss.

The start of the work with PROD B began with an initial meeting with only three persons: Svensson, an engineer and the author. This meeting concerned how the system works today, and possible new inputs or change in the current inputs. The result of this meeting was transformed into use cases and GUI examples and sent out to the rest of the group before the next meeting.

The next meeting was very fruitful, and discussed pros and cons with different approaches for entering the input, based on the GUI examples. There were suggestions for improvement of the user interface which were noted.

When the third product, PROD C, was in focus there were some changes. It was decided that this product should be incorporated with PROD A's rating and sorting but not within this project. The reason for this decision was that there are some similarities between PROD A and C which make it possible to join these two products into only one configuration program, i.e. the first part of the requirements engineering in this project.

5.5. Validation and verification of the requirements

Validation and verification are not the same thing but they are easily confused. Boehm [29, 30] expressed the difference between them in this way:

- *Verification: To establish the truth of correspondence between a software product and its specification (from the Latin* veritas, *"truth"). Or in other words: Are we building* *the product right?*

- *Validation: To establish the fitness or worth of a software product for its operational mission (from the Latin* valere*, "to be worth"). Or in other words: Are we building the right product?*

Two techniques of system checking and analysis can be used within the verification and validation (V&V) process [29]:

- Software inspections in which requirements documents, design diagrams and the program source code is analysed and checked.
- Software testing in which the software is executed with test data and the output is analysed together with its operational behaviour.

In this case software testing is out of the picture since there will be no program as result and this leaves software inspection as the technique to be used here. Software inspections are known to be an effective way of finding errors [29, 31]. At this stage there were only requirements, but those needed to be verified so that the programmer is able to write the program's functionality correctly. Here a variant of formal inspection [29, 32], which was developed by Michael Fagan [33] at International Business Machines (IBM) [34], will be used. The formal inspection process involves the interaction of the following five elements:

- Well-defined inspection steps.
- Well-defined roles for participants (moderator, recorder, reader author, inspector).
- Formal collection of process and product data.
- The product being inspected.
- A supporting infrastructure.

In this project there was less formality in this process. The inspection steps were clear since there were the requirements to be reviewed and they must be dealt with in a certain order (more or less) for each product type. The roles were narrowed down to moderator, recorder and inspector.

The inspected documents were the use case documents and the requirements documents. The inspection meeting usually dealt with one document per meeting. If a meeting for a document was short a discussion about the next document took place.

The initial inspection was performed by everyone by themselves before the meeting, and then collectively during the meeting, by all together which was carried out as a discussion about the document. During the meeting notes were taken about the opinions stated regarding the document. After the meeting the document was adjusted as agreed and then sent out to everybody again for a last inspection.

6. Prototyping

6.1. Overview

A good explanation of prototyping is given by Lantz [35]:

> "*Prototyping is about people. Users are not often able to articulate what they want an information system to do, and they cannot visualize it from written specifications. Prototyping enables them to see a system, "play" with it and modify it before it is implemented.*"

A prototype is a type of model that appears to be working since it can look and/or behave similarly to the target system. Prototypes are created for the purpose of judging, demonstration, or experimentation. In systems development, prototyping involves building models of some portion of a delivered system quickly and inexpensively to check functionality, performance or fit. Prototyping can be used to verify the validity of the requirements.

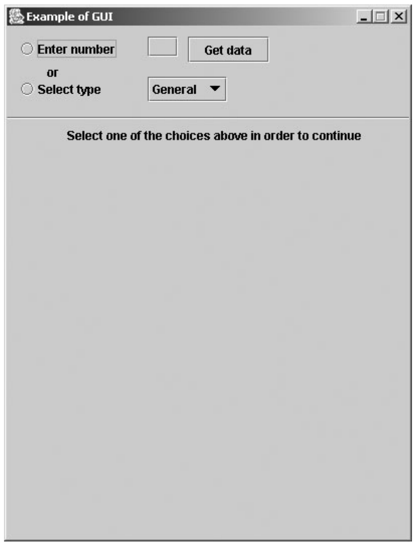
Prototypes can take many forms, from paper illustrations to online working models. The appropriate use of prototyping can greatly enhance the effectiveness of the development process.

6.2. Methods

Prototyping can be performed in several ways, such as a sketch or a working model. A sketch seemed too simple and would probably not convey the message as well as a working model. A working model seemed too complex and its usefulness would probably be outweighed by the development time required. The best solution seemed to be a compromise between these two solutions.

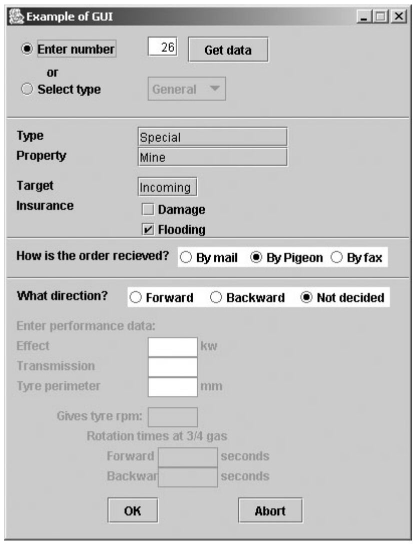
The choice fell on a working model that shows the major parts of the functionality.

In order to develop the prototypes there was a need for a tool. As mentioned earlier, Java and NetBeans was chosen for the prototyping. Java is a programming language from Sun Microsystems [26]. NetBeans [27] is an Integrated Development Environment (IDE) which is a programming environment that has been packaged as an application program, usually consisting of a code editor, a compiler, a debugger, and a



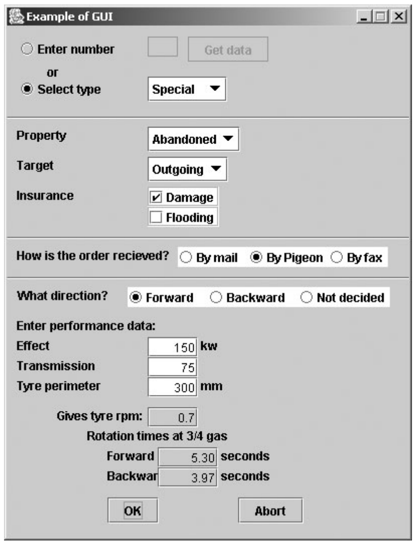
Select one of the choices above in order to continue

Figure 1. The interface showing the first choice and the rest of the form is hidden.



[✓ Enter number] 26
Select type: General
Type: Special
Property: Mine
Target: Incoming
Insurance: ☐ Damage, ✓ Flooding
How is the order recieved? By mail [✓ By Pigeon] By fax
What direction? Forward Backward [✓ Not decided]
Enter performance data:
Effect kw
Transmission
Tyre perimeter mm
Gives tyre rpm:
Rotation times at 3/4 gas
Forward seconds
Backwar seconds

Figure 2. The first form with part of the form inaccessible due to a choice made.



Enter number
[✓ Select type] Special
Property: Abandoned
Target: Outgoing
Insurance: ✓ Damage, ☐ Flooding
How is the order recieved? By mail [✓ By Pigeon] By fax
What direction? [✓ Forward] Backward Not decided
Enter performance data:
Effect 150 kw
Transmission 75
Tyre perimeter 300 mm
Gives tyre rpm: 0.7
Rotation times at 3/4 gas
Forward 5.30 seconds
Backwar 3.97 seconds

Figure 3. The second form with the whole form accessible.

graphical user interface (GUI) builder. Java has an advantage over several other programming languages by the ability of being executed on many different platforms.

A sample of the SRS is provided in Appendix 3.

6.3. Development of the prototypes

The development of the prototypes began after the first meeting with the UG, since the first requirements had then been elicited. By using Java and NetBeans the prototypes could drawn up with a minimum of effort. The first prototype was a crude implementation with all information visible and active. The feedback on the requirements with regards to that prototype was very mixed. Some saw through the faults of the interface and commented the requirements. Others got stuck on the interface; “Why is that information in two places?”

The next version of the prototype had acquired some functionality. Adding the functionality required more work in the NetBeans IDE. The functionality consisted of the additional feature that the interface ‘greyed out’ (made inactive) the part that was not selected. All information was still visible but some got greyed out based on the choices made by the user. Still some users got stuck on the interface.

The third version went further by not showing the second step until the user had made the first choice as Figure 1 shows.

When the user has made his choice, different forms will be shown depending on the choice, see Figure 2 and Figure 3. In the same way other options will be available according to the user’s input. Figure 2 shows how a part of the form is inaccessible due to a choice made. Figure 3 shows that the whole form is accessible since another choice has been made.

The users also wanted colour-coding of the possible input fields for the current choice. Using this feature, it is even simpler for a user to see what input that is needed and where to enter the input.

The ability to be able to go back and change some of the input was another desired feature requested by the users. This enables the user to test different data in an easy way instead of needing to start from the beginning for every possible input combination the user wants to try.

The third prototype was appreciated by the UG. Now the meeting could be 100% focused on the requirements instead of faults in the interface. The other GUI examples followed this approach.

7. Result

Sommerville [3] was right about legacy systems and the risks involved (see Section 1), with replacing the system or keeping it and continuing to update it. In this case there was no specification of the old system. There existed a document that had partially documented the calculation and sorting for one of the product types. The many years of maintenance and extending the software had left traces in the structure of the source code (see Section 7.1).

The author felt helped by the old source code in the requirements engineering process, both by getting a better understanding of the problem, and by being able to retrieve some of the old requirements from the old source code. Also, the author did not feel 'blinkered' after the documentation of the source code, even when feeling helped by the same. This is important in order to be able to perform the requirement engineering in a successful way.

7.1. Documentation of the Source Code

The documentation of the source code went very well even though the software did not have any documentation or comments in the source code and the author is not so familiar with C++. The method of explaining chunks of code, i.e. what those lines of code does and not how it does it, was a good way to explain the functionality of the code without going into technical details all the time.

The three files were very similar to each other. The findings of identical code in the three files indicate that 'cut-and-paste' [36, 37] programming has been used thus adding to the similarity feeling. In addition, the code itself seemed to be well-coded, i.e. only discovered one error in the code and the code was properly intended which helps the readability of the code.

The structure of the code was not as good as the code itself. There have been some efforts to clean up the code by reducing some of the identical methods. One method was not used by File One but by the other two, and in this case the method was kept in File One and commented out in the other two files. This is not as strange as is sounds, since there are other methods in File One that is used from the other two files.

Of these six methods in File Three there was one method that was exactly identical with the namesake method in File Two, and three methods that were identical except for which database that should be used when comparing with the namesake methods in File Two.

The presence of commented out methods and code made the documentation process somewhat slower since there was a lot of code to scroll by. It also added to the 'unstructured' feeling.

The use of dummy variables was not consistent throughout the code, which indicates that all code has not been debugged. As an example, File Two and Three contains the method `compareSomeText` (see Listing 1), which are identical except for which database is used and that File Two has dummy variables.

7.2. Requirements Engineering

Working with the UG during the meetings was a successful technique except for the first meeting, at least for this project. In the first meeting the discussion got off-topic for some time but the overall quality of the meeting was good.

The use of GUIs seemed to have been a successful way of communication, in order to avoid misunderstanding in verbal communication or misinterpretation of the text in the use cases.

The most surprising outcome here was that there was no need to revise the calculation formula, since that was the target for this project. The actual result was the alternative ways to enter the input and that the users wanted a more 'intelligent' user interface that showed only what the user needs according to the user's input so far.

Since there was no revision of the formula for the rating calculation, the existence of the legacy code was very helpful when creating the formula for the rating calculation. To create this formula without the legacy code would probably have taken a long time with much trial and error.

The validation and verification of the requirements went smoothly. This part was simple to perform, since the task at hand was to go through the document and verify that the requirements were written according to what was agreed upon in the meeting. The language in the requirements document contained some technical words from the area of the products. This could be a source of misinterpretation, so explanations were given to all of those words thus reducing the risk of misinterpretation.

If there had been old requirements documents this process would probably have gone much more smoothly since the start of this process would have been the old requirements. From these requirements, one could have selected those that could be used again and then focused on the rest, i.e. avoiding double work. As Salit [38] writes; Requirements are corporate assets, since it takes time to elicit requirements and time is money.

7.3. Prototyping

The prototypes showed that some people look more at the images (screen shots) than at the text. That kind of behaviour was not expected by the author. This shows that communication is easily disturbed by documents and/or prototypes with indistinct qualities.

Therefore the first prototypes must be as good as possible to avoid focusing too much on the layout of the prototype.

The use of use cases with examples of the GUIs was appreciated by the UG, at least after the third version of the prototypes. This lead to better discussion of the requirements since there was a tangible item to discuss.

8. Discussion

The question at hand is whether the author was influenced by the old software in such a way that the requirements engineering process was performed with 'blinkers'. The fact that the author started out by documenting the source code, before the requirements engineering, could have some impact on how open the mind was when the requirements engineering started. In this case it probably did not affect the requirements engineering process, since the author had poor knowledge of the products, and thus had to ask many questions, both based on the knowledge from documenting the source code and trying to see new options.

The author felt helped by the old source code in both understanding the problem and by being able to gather some of the old requirements from that code. Without the source code this work probably would not have gone as well as it did.

8.1. Documentation of the Source Code

Will the documentation that was performed on the source code be enough in other cases? On this issue, it is hard to tell, since this assessment is up to the person that requested the documentation or the purpose of the documentation. In this case it was sufficient.

The code itself seems to be very well written; only lacking comments. The presence of comments makes the code even easier to read for another programmer. How much comments there should be depends mostly on the difficulty of the code. Simple, self-explanatory code will probably not need any comments but more complex code would most likely benefit from comments.

The structure of the code is not very good as indicated by the duplication of methods. The developers should have adjusted the code during the development of the program, but it is easy to criticise afterwards. This could also depend on that fact that cleaning up code costs money, and the company did not want to spend money on that.

'Cut-and-paste programming' is a nightmare for maintenance, since there must be modification in several places, which means that it easy to miss one place and then there is an error in the code. The benefit with this type of programming is that it is fast, but it still is not a good way of programming since it is often error prone and hard to debug [36, 37].

In the original source code, there are several methods that are practically identical except for the database connection and dummy variables. This is probably due to the fact that the software has been upgraded and extended for more than twelve years. Perhaps the cost of correcting this unstructured code is a reason for not performing the correction. If they corrected the code to a structured format where all common methods are grouped in a single file, separated from the product specific code, and adjusted the methods to be able to receive a specified database as an input parameter there would be a need for extensive testing to ensure the functionality of the software, and testing costs time and money. The benefits for doing this, however, are less code, better structure of the code and easier maintenance.

8.2. Requirements Engineering

The requirements engineering component of this project was the most difficult part since the area of products was new to the author and there was a steep learning curve to understand how all the input should be used in order to get a good result.

The purpose with the meetings was to elicit as many requirements as possible. The meetings worked well as a requirement elicitation technique and produced requirements regarding how to enter the input and the data required for the calculation.

It is doubtful that this result could have been produced in another form of elicitation technique like surveys or one-on-one interviews, with regard to the author's knowledge in the area of these products. On the other hand there is, most likely, no 'one way for them all' solution, so the person(s) that will conduct an investigation must compare the pros and cons for different solutions and then choose the one that seems to be the best for the job.

The process could most likely have been faster if the author had had better knowledge about the products and how the data is connected. In this case the author had to put some time and effort into understanding the products before any of the preparations for the meeting could take place. With knowledge about the products the whole process could most likely have started earlier.

The meetings, as method of operation, had some failure in the communication, since all participants did not receive the material that was sent out. A possible reason for this is the mail client used within in the company, that perhaps did not show the attachment clearly enough amongst the other information in the letter inviting the participants to the meeting. Another factor could be an anti-virus program or other protection programs that blocked the attachment for some reason, or plain inattentiveness from the recipient. Therefore there must be a way to ensure that everybody gets the material that is sent out. One possibility might be to include an index to the mail stating the important parts.

That the result was requirements for how data could be entered, and not requirements for an update of the calculation formula, was unexpected. One possible explanation might be if the GUIs somehow could have been inhibiting the creative thinking in the UG. This thought can be countered by the fact that the discussions were lively and active concerning possible inputs whether they were relevant for the current product or not.

The absence of the original requirements containing the formula for the calculation made the construction of the formula go slower, since the formula had to be retrieved from the legacy code. The presence of the old source code was helpful when expressing the formula for the rating in the requirements document, since there was no need to 're-invent the wheel'.

Failing to maintain requirements adds time and cost to every subsequent project [38]. This can be considered proven by this project, and undoubtedly many more, since the process had to start all over again in order to be able to document the requirements for the calculation formula. As mentioned earlier, requirements are corporate assets. The question of why they are not treated like that, only the company can answer.

A difference between this project and many other projects is that this was targeted to a specific customer with a specific need and not a mass market. A software product targeted to the mass market is most likely generic to suit as many potential customers as possible. When this product evolves the functions of the software are improved and/or new functions are added. Some of these functions are demanded by the market and others are probably added just to give more functionality. In this case the product was targeted to a specific customer with specific needs. In this case the software should only contain what is necessary and nothing more in order to keep it simple to use (and in the long run, also easy to maintain).

8.3. Prototyping

The choice of IDE for prototyping does not matter since it is the result that counts. The use of Java/NetBeans could as well have been replaced with another IDE like Microsoft Visual Studio [39] or any of Borland's [40] IDEs. The choice of Java/NetBeans was simply because they were free to use and contained sufficient functionality.

There is no way of telling in advance if the produced prototype is good enough to let the people focus on the requirements instead of focusing on the defects of the prototype. In this case the development of the first prototype maybe should have been put aside until the second meeting was over. One can only guess if delaying of the prototype would have helped the requirement engineering process or not. If the first prototype had not been at the second meeting, the question is if that meeting would have produced what it did. But quite clearly the quality (accuracy to the requirements) is of outmost importance to keep the discussion focused on the right subject.

The use of use cases and GUI examples was obviously a good way to support the discussion of the requirements, even if the text was in the background in comparison with the GUI examples.

9. Conclusion

Requirements are most likely the single most important part of the software development process. With deficient requirements the software will also be deficient and the whole software development process has failed. To reduce the risk of deficient requirements, the developer must be open-minded and look at the problem from several angles in order to elicit the best requirements possible.

The requirements engineering process was performed in a legacy setting, starting with the documentation of the code. The code seemed well coded but with an unstructured streak. Although the legacy code provided some difficulties, the documentation process went well. The requirements engineering process had to start from scratch, since there was no previous documentation to start from. The use of prototypes during the requirements elicitation meetings helped the requirements elicitation process. The use of prototypes helped the communication with the user group. Prototypes helped to get over any language barrier since there was something to point at when explaining something, but prototypes with indistinct qualities did hinder the requirement elicitation process.

A legacy software setting did not seem to make the developer narrow-minded. Instead a legacy software setting can speed up the requirements engineering process, since several factors are known like, in this case, the calculation formula.

Did the legacy software help or hinder in this project? In this project it was helpful to have the legacy code since it speeded up the requirements engineering process due to less duplication of work.

10. Future Work

More of this kind of work must be performed in order to give a generic conclusion about a legacy software's impact on the development of a new version of a software. This further work may also help to answer the question of whether there are cases where a legacy software setting hinders the requirements engineering process.

11. References

[1] S. Rugaber, "A Tool Suite for Evolving Legacy Software", Institute of Electrical and Electronics Engineers, pp. 33-39, URL: http://80-ieeexplore.ieee.org.ezproxy.htu.se/iel5/6437/17178/00792496.pdf?tp=&arnumber=792496&isnumber=17178, 2004-05-01.

[2] I. Sommerville, *Software Engineering*, 6th ed., Addison-Wesley, Harlow, 2001, pp. 97-147.

[3] I. Sommerville, *Software Engineering*, 6th ed., Addison-Wesley, Harlow, 2001, pp. 581-600.

[4] A. I. Antón, "Successful Software Projects Need Requirements Planning", Institute of Electrical and Electronics Engineers, pp. 44-46, URL: http://80-ieeexplore.ieee.org.ezproxy.htu.se/iel5/52/26915/01196319.pdf?tp=&arnumber=1196319&isnumber=26915, 2004-05-07.

[5] C. Clavadetscher, "User Involvement Key to Success", Institute of Electrical and Electronics Engineers, pp. 30-32, URL: http://80-ieeexplore.ieee.org.ezproxy.htu.se/iel3/52/14540/00663781.pdf?tp=&arnumber=663781&isnumber=14540, 2004-05-07.

[6] S. Prata, *C++ programmering*, 2nd ed., Pagina, Stockholm, 1996.

[7] B. Stroustrup, *Programmeringspråket C++*, Addison-Wesley, Harlow, 2000.

[8] I. Sommerville, *Software Engineering*, 6th ed., Addison-Wesley, Harlow, 2001, pp. 511-534.

[9] D. Spinellis, "CScout", URL: http://www.spinellis.g r/cscout/, 2004-05-10.

[10] C. Felaco, "Cbrowser home page", URL: http://cbrowser.sourceforge.net/, 2004-05-10.

[11] M. DeJong et al., "Source-Navigator(TM)", URL: http://sourcenav.sourceforge.net/, 2004-05-10.

[12] Object Management Group, "OMG Management Group", Object Management Group, URL: http://www.omg.org/, 2004-05-04.

[13] Z. Pozgaj, H. Sertic, M. Boban, "Effictive requirements specification as a precondition for successful software development process", Institute of Electrical and Electronics Engineers, pp. 669-674, URL: http://80-ieeexplore.ieee.org.ezproxy.htu.se/iel5/8683/27508/01225420.pdf?tp=&arnumber=1225420&isnumber=27508, 2004-05-12.

[14] Coley Consulting, "What is a Requirement or Specification", Coley Consulting, URL: http://www.coleyconsulting.co.uk/require.htm, 2004-03-29.

[15] A. Abran et al., "SWEBOK", Institute of Electrical and Electronics Engineers, pp. 10-14, URL: http://www.swebok.org/stoneman/version_1.00/SWEBOK_w_correct_copyright_web_site_version.pdf, 2004-05-04.

[16] C. Larman, *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and the Unified Process*, 2nd ed., Prentice-Hall, Upper Saddle River, 2002, pp 41-44.

[17] R. Harwell et al., "What Is A Requirement?", Lockheed Corporation, 1993, URL: http://www.incose. org/rwg/what_is.html, 2004-03-29.

[18] Institute of Electrical and Electronics Engineers, "Welcome to the IEEE", Institute of Electrical and Electronics Engineers, URL: http://www.ieee.org/portal/index.jsp, 2004-05-11.

[19] Institute of Electrical and Electronics Engineers, "Welcome to the IEEE", Institute of Electrical and Electronics Engineers, URL: http://ieeexplore.ieee.org/xpl/tocresult.jsp?isNumber=4148, 2004-05-11.

[20] H-E. Eriksson, M. Penker, *UML Toolkit*, 1st ed., John Wiley and Sons, New York, 1997, pp. 269-294.

[21] C. Larman, *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and the Unified Process*, 2nd ed., Prentice-Hall, Upper Saddle River, 2002, pp 45-82.

[22] Object Management Group, "UML", Object Management Group, URL: http://www.uml.org/, 2004-05-09.

[23] International Business Machines (IBM), “Rational software from IBM”, IBM, URL: http://www-306.ibm.com/software/rational/, 2004-05-09.

[24] C. Larman, *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and the Unified Process*, 2nd ed., Prentice-Hall, Upper Saddle River, 2002, pp. 13-28.

[25] Institute of Electrical and Electronics Engineers, “Welcome to the IEEE”, Institute of Electrical and Electronics Engineers, URL: http://ieeexplore.ieee.org/xpl/tocresult.jsp?isNumber=15571, 2004-05-11.

[26] Sun Microsystems, “Java Technology”, Sun Microsystems, URL: http://java.sun.com/, 2004-04-17.

[27] NetBeans.org, “Welcome to NetBeans”, NetBeans.org, URL: http://www.netbeans.org/, 2004-04-17.

[28] L. A. Macaulay, *Requirements Engineering*, 1st ed., Springer, London, 1996.

[29] I Sommerville, *Software Engineering*, 6th ed., Addison-Wesley, Harlow, 2001, pp. 419-439.

[30] SINTEF Telecom and Informatics, “TIMe manual version 4.0”, SINTEF Telecom and Informatics, URL: http://www.sintef.no/time/ELB40/ELB/VV/VV.pdf, 2004-04-18.

[31] V. A. French, “Applying Software Engineering and Process Improvement To Legacy Defence System Maintenance: An Experience Report”, Institute of Electrical and Electronics Engineers, pp. 337-343 URL: http://80-ieeexplore.ieee.org.ezproxy.htu.se/iel3/4043/11593/00526555.pdf?tp=&arnumber=526555&isnumber=11593, 2004-05-12.

[32] L. Rosenberg, “Software Formal Inspections”, National Aeronautics and Space Administration, URL: http://satc.gsfc.nasa.gov/fi/fipage.html, 2004-04-18.

[33] M. Fagan, "About Michael Fagan", Michael Fagan Associates, URL: http://www.mfagan.com/bio.html, 2004-05-12.

[34] International Business Machines, "IBM United States", International Business Machines, URL: http://www.ibm.com/, 2004-05-12.

[35] K. E. Lantz, “THE PROTOTYPING METHODOLOGY”, URL: http://www.manageknowledge.com/prototyp.html, 2004-05-10.

[36] S. Walters, “Cut And Paste Programming at Perl Design Patterns Wiki”, S. Walters, URL: http://www.perldesignpatterns.com/?CutAndPasteProgramming, 2004-05-09.

[37] S. D. Huston, D. C. Schmidt, “Object-Oriented Frameworks for Network Programming”, Pearson Education, 2004, URL: http://www.awprofessional.com/articles/article.asp?p=31351, 2004-05-09.

[38] R. Salit, “Requirements Are Corporate Assets”, Institute of Electrical and Electronics Engineers, pp. 86-88, URL: http://80-ieeexplore.ieee.org.ezproxy.htu.se/iel5/52/26915/01196327.pdf?tp=&arnumber=1196327&isnumber=26915, 2004-05-01.

[39] Microsoft, “Visual Studio Home”, Microsoft, URL: http://msdn.microsoft.com/vstudio/, 2004-05-12.

[40] Borland, “Borland Products and Software Solutions”, Borland, URL: http://www.borland.com/products/, 2004-05-12.

Appendix 1. Sample of the code documentation

Appendix removed due to secrecy.

Appendix 2. Sample use case for PROD B

Appendix removed due to secrecy.

Appendix 3. Sample of the Software Requirements Specification

Appendix removed due to secrecy.