REENGINEERING
and
REENGINEERING PATTERNS

Daniel Gjörwell
dgl99001@student.mdh.se

Staffan Haglund
shd99004@student.mdh.se

Daniel Sandell
dsl99001@student.mdh.se

The Department for Computer Science and Engineering
Mälardalens Högskola

2002-02-24
Västerås

ABSTRACT

Reengineering and Reengineering patterns is a relatively new concept that has begun to make an impact on the software engineering sociaty. By shifting resources towards the restructuring of old legacy software systems rather than focusing on new software development the businesses would and have been able to save precious time and resources. Even though the benefits of this approach is clear, the difficulties that follows with the concept, together with inexperience and the lacking of appropriate tools make the reengineering of software systems a difficult but interesting subject.

3. THE HISTORY OF THE REENGINEERING CONCEPT

In early years of the information revolution the need for reengineering was not acknowledged by the wider community. Instead, attention was directed towards the discovery of new ways of creating both better hardware and better software. Methodologies that were concerned with how to engineer the development of new systems where published, cheered and disposed in an ever increasing rate. Almost no attention were given to the old systems that were getting more and more outdated. In addition, the businesses where changing rapidly and with them came the need for appropriate information software. This became known as 'software shortage'; since there always seemed like newly devoloped software where not good enough to accommodate the business needs.

Then, in the early 90's the focus of system devolopment changed very rapidly from the development of new software to the reengineering of old 'legacy' systems (systems developed over time and in need of maintainance). The fact is, that so much attention where given to reengineering that entire businesses where cought up in the excitement and had their entire business structure reorganized according to the newly developed reengineering methodologies and patterns that had emerged. Reengineering was the word of the day and the reengineering consultants where having a field day [BPR 99].

But soon it became appearant that the reengineering of both business and it's software where not as easy as the consultants had first believed. Over half of the reengineering processes of the time failed, mostly due to inexperience and lack of customer involvement. With these failures followed huge costs for the companies and soon the reengineering boom was over, and with it the interest in reengineering devolopment [BPR 99].

Still, almost 80% of a business information system budget costs comes from the maintainance of old legacy systems. That means only 20% of the total cost can be related to the development of new systems [SR]. This shows that reengineering, the reorganization and redesign of a system (or business), is very important; since if these costs can be reduced, much will be gained for the software user. This fact has contributed to the return of an active reengineering research community, and as we enter the 21st century we see that reengineering is once again marching forward into the frontlines of software engineering.

4. SHORT INTRODUCTION TO REENGINEERING

Reengineering concerns the examination of the design and implementation of an existing legacy system and applying different techniques and methods to redesign and reshape that system into hopefully better and more suitable software.

This process is by no means an easy task, since legacy systems may have come a long way from the state in which it was first concieved and implemented. Updates and the adding of new functionality may cause a lack of proper and updated documentation, especially if the system is entrusted to people not skilled in the reengineering way of thinking.
Also, maintainability may suffer as the source code becomes flooded with new ways of communication; and the breaking of encapsulation of objects in favour for an easy way to some immiediate goal may lead to code so complex, it is almost unmaintainable .

The reengineering team must be able to see through the dense jungle that often characterize a legacy system, and find those parts that still has a meaning in the final system. Taking such decisions must be done with great care and great thought, since the reengineering process is about taking care of the end users current and future needs; as well as ensuring a quick and easy transition to the new system. If the entire system is redesigned and implemented then we don't have a reengineering process, but a pure software engineering process with an end product probably having the need for extensive user training.
You could therefore say that when reengineering, you try to change as little as possible while still ensuring that fundamental aspects like maintainability, userability, functionality and other requirements are still addressed in a proper way.
This gives us our main paradigm in reengineering:

"Change, but change as little as possible."

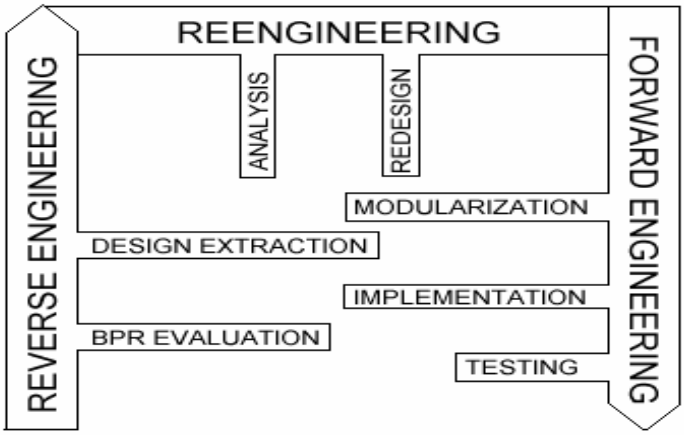

Figure 1: The Phases of the Reengineering Process

In order to get a more comprehensive view on the Reengineering of legacy systems we will have to look at the parts that define the reengineering process: the reverse engineering phase, which is concerned with the extraction of elements and information from an existing system. The phase that concerns the analysis and redesign takes the data obtained in the reverse engineering phase and tries to identify the aspects that may be reused and the parts that need to be replaced or redesigned. This phase is also concerned with the designing the final system. Finally we have the forward engineering phase, which addresses the implementation of the reengineered system other aspects vital to the extension of the software life-cycle [REFITS 94].

5. REVERSE ENGINEERING

5.1 Introduction

Reverse engineering were at the beginning used for analysing hardware to discover its design but is nowadays equally used in software reengineering. Reverse engineering generally seeks to recover information from a low level of abstraction to a higher level, such as finding design information from source code.

Reverse engineering is the first stage in the reengineering process to explore the software system at hand, in an effort to document the systems advantages and flaws and to find reusable components from the software. If reverse engineering is used alone then it does not involve modifying the software system.[FAMOOS99]

Here are some advantages and problems with using reverse engineering according to [WHRES96].

5.2 Advantages

It can both mean the development of never existing design documents and also the recovery of information that has been lost over the years. Recovering information from old software systems is very important, particularly if the systems is to be maintained by maintenance engineers who did not develop the system. Such a team would really be having an almost impossible task ahead of them if the reverse engineering phase was not a part of the reengineering process.

In software reuse, one of the key issues is the definition and development of reusable elements such as objects, software components and parts of the system documentation. With reverse engineering it is possible to provide different kinds of system documentation and also identifying reusable component that exist in the system.

The major goal of using reverse engineering is quality improvements of the system but also the minimization of expenditures and software reuse facilitation.

5.3 Problems

Here are some problems that have to be solved when using reverse engineering:

One problem is that the source code can be poorly structured and design specifications missing or be incomplete. Documentation of the system could be out of date or even not exist at all. Other problems are the incorporating of modules and the module complexity.

When a system has been developed with inconsistent principles it often makes it difficult to work with. When reverse engineering, a large amount of source code usually has to be examined, and based on insufficient knowledge about the program or even without any system documentation at all it may need extensive research of the system structure.

There are efficient tools that supports the reverse engineering process but they suffer from the fact that they are unable to extract sufficient information from the available sources (like source code). So the best way to solve these problems is to combine automatic tool assistance with human expertise support when the tools reach their limits.

6. ANALYSIS AND REDESIGN

6.1 Introduction

This is the second phase of a reengineering process [BPSARLSS94]. The phase is closely connected with the usual analysis and design phases seen in ordinary software engineering projects; but with the difference that in a reengineering process, the old system already has a design structure and is therefor putting its own features and constraints on the new design.

6.2 Analysis

The analysis of a legacy system is in our opinion the most important part of the reengineering process. This is due to the fact that without a proper analysis and thereby a proper understanding of the software you will not be able to complete your project with a satisfactory result. The extrapolation of the costumer and/or user need is vital to get a clear picture of the final product and the benefits and drawbacks of that software.

In the analysis, the reengineering team will begin by identifying these needs and reshape these into highly specified requirements. Getting these requirements right from the beginning of the project is important, as every step in a reengineering chain usually causes the costs to increase [FAMOOS99]. This is a common problem in software engineering and the reengineering model is no exception.

There is also the fact that some projects doesn't get enough benefits from the reengineering to be cost-effective. Most methodologies used in reengineering projects have a cost versus benefit stage in which the team analyse this aspect of the project [FAMOOS99][RM]. If it turns out that the changes is either too small to make a difference in the overall system or too big to implement in a reasonable amount of time or that the cost of an extensive change in implementation would be too high, then the team may actually need to reanalyse the project once again. Either it will turn out to be a case where the project actually has little or no use; or the team may have missed some information in the previos stages that would cause the scale to tip in favour of a continued reengineering process. According to some sources, a cost vs. benefit stage is almost a must-have in a reengineering process [RM]. This can also be seen in the Standard Methodologies chapter later in this report, where the cost/benefit step is used extensively.

Like in ordinary software analysis the results must be carefully monitored and documentated. This is especially important if the analysis has revealed new functionality requirements. A well documented analysis does not only contribute to an easier design and implementation of the new system but also provides useful data to a development team in the future.

After the requirements have been specified, the team may look at the redesign of the legacy system. Redesigning a legacy system means that the team will try to change that system according to the analysis results.

6.3 Redesign

Redesign is the application of the analysis results on an existing design, with the addition of new design elements to incorporate these changes into the system functionality. Redesign has much in common with the design phase of an ordinary software engineering project. The difference lies in that the design of the reengineered system must consider the old design elements and how these will be affected by a design change.

Redesigning is sometimes difficult since it may be a serious conflict between the new design structure and the original design. If the designs proves to be too incompatible, then it comes down to choosing whether or not to change the entire design or try a different approach (remember paradigm 1). If the designs on the other hand fit together then the implementation of the new software will probably be easy and the time needed to get the software delivered will be shorter. In either case, the design of the new system must be very careful and well documented, just as in the analysis stage. This is especially try if the changes includes changes to interfaces, both towards other software parts or other forms of end-users [FAMOOS99].

If the design follows the object-oriented approach, careful consideration must be placed on the classes and their interaction as well as the interaction between the application and the end-user. The team should generally try to minimize changes to the interfaces. Since such changes may cause the software to be incompatible with the environment; thereby causing a need for updates on those systems as well [FAMOOS99]. Such design will increase the cost for the costumer and lead to disadvantages for that business in the long term.

Another reason to limit the changes to interfaces are that in our view; human end-users are not keen on changes, especially if the interfaces has changed very little over a long period of time. Long educating times can quickly reduce both the user interest in how the software should be used, as well as causing them to abandon the system entirely (maybe even in favour for the old one).

7. FORWARD ENGINEERING

7.1 Introduction

Forward engineering is the same as the usual process of software engineering, but it have another name so that it can be more precisely seperated from reverse engineering and reengineering. It uses the new design that have been made during the analys and design phase to progressively move from high level design to low level design and implementation.

Forward engineering can be done in three steps [BPSARLSS94]:

7.2 Modularization

It is a process to splitt data, construct application layers and to produce groups of classes that facilitates the same functionality in order to produce reusable and exchangable elements [BPSARLSS99]. The results of modularization are reduced program size and reduced complexity. And if a program is not so big and not so complex it also means that it will be easier to maintain the program.

7.3 Implementation

During the implementation of the program it is important not to build too big or too small a class. One basic principle is that a class should implement one single concept. If a class implements more than one concepts, then the class probably have low cohesion measurements because often these concepts could have been implemented separately. If a class does not implement one single concept by itself, it means that the concept is implemented between several classes and is probably tightly coupled to these [FAMOOS99] . The forward engineering must conduct it's own low-level design in such a way that the cohesion and the coupling is kept on a level that enables reuse and maintainability.

7.4 Testing

Testing is made to find errors that possibly could occur during program execution and also to improve things that already works (optimization). Some test scenarios can be planned in advance, usually for those parts that are fundamental for the program. Such test scenarios are usually designed and implemented during the later phases of the redesign step or early in the forward engineering phase.

Test persons can also be used to test the program, the persons who tests the program can have different experience of working with that kind of software and get different test results. These results can be used to improve the user interface and to find errors that the engineers did not find in their original tests.

8. DIFFERENCES BETWEEN DESIGN PATTERNS AND REENGINEERING PATTERNS

8.1 Design patterns

What is a design pattern? A design pattern describes a common problem that we might face within a software system, and also the essentials of the solution to that problem. The description of the solution should be in such a way that one can use the solution again and again.

A pattern has four essential parts:

- Name
  Short and descriptive name to describe the design pattern.

- Problem
  Describes when to apply the pattern, and explains the problem to be solved.

- Solution
  This is a description of all the elements that the design solution is made up of, as well as their relationships, responsibilities andcollaborations.

- Consequences
  The results and trade-offs, costs and benefits, etc.

A design pattern is always on a high abstraction level. It should not go into small details regarding the solution, such as; whether you are using a stack, queue or other very specific parts of a software system. Instead, they describe the classes and instances, what their roles are and how they collaborate, and their responsibilities [FAMOOS99].

With the four parts mentioned above, a design pattern is described. The Problem part describes when a particular pattern should be applied. In Solutions, we will find a description of the design, of its elements and how these elements interact, their responsibilities, relationships and collaborations. A design pattern also has to show the consequences of using it in an application. Knowing consequences, the results and trade-offs for example, is very important when it comes to selecting a pattern. The solution alone is not sufficient, if it brings other problems instead.

8.2 Reengineering patterns

Reengineering patterns are patterns that describes how to change a legacy system into a new, refactored system that fits current conditions and requirements. The main goal with reengineering patterns is to offer a solution for reengineering problems. They are also on a specific level of abstraction; they describe a process of reengineering without proposing a complete methodology, and they can sometimes suggest a type of tool that one could use.

The idea behind reengineering patterns is that a developer must diagnose a problem, see all available options and choose a particular course of action. Generally the pattern format has been defined with some of the properties below in mind:

Focus on reengineering process.
Of course a reengineering pattern should focus on the reengineering process. For example, how can we define the problems that has arised with the current system, and how do we define them? What may be the pitfalls in transforming a system when this pattern is applied?

Easy navigation
The reengineer should be able to determine a patterns applicability rather quickly. Therefore a pattern should give clear and precise information about it's intended use.

Separate out tool and language dependent issues.
Since the pattern must be applicable in the most general way, and therefor should not specify what tools or languages to use. The tools and the languages that one will use will probably be subject to change, whereas the pattern, as it is a general solution, can and should, remain the same.

Standard terminology and notation.
As we want to separate out the tool and language dependancy, the use of a language neutral notation and terminology is mandatory.

The structure of a reengineering pattern consists of some essential elements, and this list should show both similarities as well as differences between reengineering patterns and design patterns.

- Pattern name
  It's very important with a short, clear and descriptive name. The name should be based on the operation that the pattern is sed for performing, since this is the most natural way to use it in discussions about the pattern.

- Intent
  Here there's a description of the reengineering process, the results and why it is desirable.

- Applicability
  Describes when a particular pattern is applicable and when it's not. Also listed here are symptoms, reengineering goals and related patterns. These are symptoms experienced during the reuse, maintenance and changing of the system.

- Motivation
  Here is an illustration of the pattern in work; descriptions of the legacy system and its structure as well as the refactored system and the relation between them. This is done through the use of a concrete example which will contribute to a greater understanding of the rather abstract presentation of the problem which follows in the structure and process sections.

- Structure
  Descriptions of the structure before and after reengineering. The participants and the collaborations between them are here identified. The structure sections also includes a discussion over any disadvantages and advantages in refactoring the system from the current structure into the new target structure.

- Process
  - The detection
    Describes methods and tools to detect that the code is suffering from the suspected problem and that the process can help to alleviate the problem.

  - The recipe
    How to perform the reengineering and possible variants.

  - The difficulties
    Optional. Situations where reengineering may not be feasible.

- Discussion
  In this section, we discuss the legacy system and the refactored system, things like how much will it cost, what are the trade-offs, what do we gain, how big is the problem, etc.

8.3 Conclusion

Although they share characteristics, reengineering and design patterns are different.

Similarities are, for example:

- Name
- Problem description
- Description of the solution
- Consequences and discussion

The difference lies in what kinds of problems the patterns are made to solve.
Design patterns describe how to solve a general and common problem in a certain way, the process of which one should use for the final solution to be reached.
Reengineering patterns instead describe how to refactor an already existing system, so that it can meet current needs and requirements. Since a design pattern is used to solve a problem when constructing new systems, they are sub sets of reengineering patterns. A reengineering pattern is often much more vast than design patterns, since the design phase – where the construction of the new target system is being carried out – is only a part of the whole reengineering process.

9. BENEFITS OF THE REENGINEERING PATTERN APPROACH

9.1 Introduction

The use of reengineering patterns have several benefits which addresses different parts of the reengineering process, from the first glance of the system at hand to the final line of code in the replacing software. Of course, all patterns are not equally good but specialize in a few aspects depending on the phase for which they are designed. In this part of the report we define what the important aspects of the reengineering process are and how the reengineering patterns can address these different aspects.

9.2 Project Understanding

Some reengineering patterns address the important part of the process that concerns the understanding of the software to be reengineered. In order to build a better system, we must first understand the original one [FAMOOS99].

The understanding is vital in the opening phases of the reengineering process, as they usually provide information that is crucial to the later phases. Some of these patterns are concerned with the code examination, reading the source code of the project in hand and then identifying the parts that can be used in the new projects as well as the parts that may or must be replaced in the final product.

Other patterns address the documentation that already exists in the project. Looking at an existing documentation can quickly give you an overview of the system, the structure and development procedure that the original project development used.

As you can see, the use of these patterns in the beginning can give you a sense of what needs to be done within a reasonable amount of time (some patterns require less than a days work in order to give satisfactory output for use in the later phases) [FAMOOS99].
We will now look at three groups that are concerned the initial project understanding phase and all have patterns associated with them. The groups are BPR, First Contact and Extract Architecture.

BPR (Business Process Reengineering)

This is the actual cause of the whole reengineering process to even exist. Firstly, we must add that the BPR is in itself an entirely own reengineering process in its own right and has very little to do with the reengineering of software systems. Therefore we leave the BPR process and BPR patterns to other research teams to examine. But BPR is important to software reengineering in one way though. Since BPR is actually the reengineering of an entire business and a new business usually means new software needs [BPSARLSS94][FAMOOS99]. Therefore a reengineering team concerned with the update of a legacy system needs to examine the variables that has changed since the original system was developed (and updated). The use of patterns may give you an initial understanding, not of the project that you are to redesign, but of the actual design of the business for which the software is intended [BPSARLSS94].

Usually, these aspects of the business should be considered:

- What are the current business requirements?
- What are the objectives of the business?
- Are the requirements and/or the objectives likely to change?

Such things are important to grasp if the team are to make a successful implementation of a new, improved system.
Again, patterns that are concerned with the examination of business changes and requirements analysis may aid you and your team in your effort to make the system more capable of coping with new business demands [BPSARLSS99].

First Contact
When you start to look at a project you are faced with a number of difficulties. First of all the share scale of the project can be intimidating. Therefore there is a risk that you don't find a way to begin your reengineering process [FAMOOS99]. This may lead to slow progress in the beginning, which in turn can cause an excessive workload later in the project. Such workloads may in turn cause the delivery of the new product to be delayed and even cancelled. Also, you will probably encounter colleagues that are sceptic in your ability of both coping with the reengineering as well as the real need of such a process. If they are in a steering position they may not trust your ability if you cannot show them good progress [FAMOOS99]. Getting a quick understanding can impress these colleagues and convince them of the need for a new product and that you are the person to entrust with the task.

Extract Architecture
After you have gotten a quick understanding of the project, you may wish to come to terms with how you can change the existing project. In order to do this you can try to extract information from available documentation and other information resources like databases and by examining the source code.

Looking a database may give you a clear view of how a project handles data and data manipulation, which can help you redesign that aspect of the project and look for tools that simplify the control of the database.

By looking at the source code you can guess on objects and classes that may be useful in the new product. These approaches are covered by the 'Check the Database' and the 'Guess Objects' patterns. Guessing objects are of course mostly useful in an object-oriented project development approach but may also be used in a non-object-oriented implementation. Building a project that starts with the identification of classes and relations can later be translated into a non-object-oriented source; probably enjoying a higher level of encapsulation than would have been the case if the project had been done in a pure non-object-oriented way [FAMOOS99].

9.3 Resource Efficiency
Since large 'legacy' systems, object-oriented or not, is a huge task, they demands a great deal of resources, both in manpower and in other forms, like tools and time consumption. Since resources are expensive, especially if the resources needed include large amount of time. The use of reengineering patterns can reduce the resources needed dramatically.

As we already seen, the use of patterns in the understanding of the original project reduced time consumption in the beginning of a reengineering process; other patterns use more or less resources and tools.

Patterns that deal with resource effiencey are usually concerned with the extrapolation of 'hot spots' or 'areas of focus' [FAMOOS99]. These areas of focus are such that they are more important to examine and to redesign than other parts of the system. Since resources are so

important, the concentration of resources to these vital areas can greatly increase the chance that the project will be completed in a satisfactory way. Areas of focus may in turn be devided into several catagories, depending on the time aspects, the complexity and the functionality of that area. Projects that are concerned with improving the userability of a system may rank the complexity as the most important area of focus, while a project concerned with the correction of flaws and bugs within the system may consider the functionality the top priority.
The use of patterns in deciding on hot spots or areas of focus may give you the upper hand when it comes to manage the resource constraints [FAMOOS99].

9.4 Late Reengineering

In the later phases of the reengineering process, the emphasis depends on the fact whether the project will end without any change in the source code or not. If the reengineering team has found out that the system under investigation has very little or no need for a continued reengineering and refactoring, the process may end at this state.

However, if the team decided to make critical changes to the system, then new patterns may be applied to help the team organize and understand the how and when to make the necessary changes. The fact that the a legacy system is usually so complex, the understanding part of the reengineering process once again proves to be very important for the success of a reengineering project. The ‘ Refactor to Understand’-pattern [FAMOOS99] address the fact that by renaming and rebuilding parts or all of the source code in the project you may actually find out the best way to implement the new functionality and by that saving time and effort. This is usually a good pattern to use in conjunction with the ‘Read All the Code in One Hour’-pattern since you would then already have an idea of how the original code was structured.

After the functionality has been added to your new software system, the testing of these changes is also very vital to a successful completion of the reengineering project.
Many patterns are concerned with the testing of new functionality, not only reengineering patterns but design patterns in a more general term. But the testing of a reengineered legacy system proves to be more difficult than that of a system that you have concieved entirely by yourself. The difficulty lies in the fact that some parts of the system may not be entirely understood, both in terms of structure and communication with other parts [FAMOOS99]. Designing good test scenarios and test programs may call for the use of well defined strategies and methodologies in the form of patterns, both reengineering patterns as well as ordinary forward engineering design patterns.

10. PATTERN NAVIGATION

There are several techniques which can be used to identify patterns. Here are some of these techniques [ICESR99].

- Study a project and use some of these four techniques.
  1. As the project proceed; try to be involved in the informal discussions.
  2. Attend meetings concerning the project.
  3. Ask questions to senior designers about the strategy they are using in the reengineering system and why they are using that strategy.
  4. As the project proceeds interview both senior decision makers and junior engineers at various stages of the project.

  Because it is difficult to take people away from their work to be interviewed, the first two techniques are often the most useful.

- When you are studying a project, try to observe problems that appears and the tactics the team use to solve it. At which areas have the team deviate from the strategy as they planned in advance.

- Interview reengineers as they have several years of experience and ask questions about how they did to the identify patterns that they using.

- Study work on reengineering projects that have been published.

Pattern navigation shows an overview of different patterns. The navigation is based on forces that are important in reengineering. The patterns are listed in a table for each navigation and in the table it shows what forces a pattern covers.
There are different types of forces, for example according to [FAMOOS99] the flexibility and understandability force which we will study more closely. Other types of forces include reuseability, effort, scalability, parsing effort and global impact of a pattern.

Flexibility
Try to transform inheritance relationship into a component relationship, because an inheritance relationship can only be changed statically but an component relationship can be changed dynamically. Using composition instead of inheritance will increase the flexibility of the system.

Another way to improve the flexibility of the system is to detect and eliminate dependencies between packages of a system that are not allowed according to the designated system architecture. If these architecture breaking dependencies are not removed they could prohibit the exploitation of the architecture's advantages and it could also cause problem at maintenance work.

Understandability
The understandability increases if breaking a single complex class into hierarchy of smaller but more specialised classes. If the classes are separated it simplifies understanding how the hierarchy could be extended.

11. TOOLS

Reengineering is a huge task. In order to reengineer a system with a successful outcome, tools are very important, since they assist the reengineers to handle the usually vast amount of data in a large legacy system. Tools will help the reengineering process by solving or assisting in solving different problems, and maybe most importantly, to save time.

There are many different kinds of tools, all which will assist the reengineers in various phases in the reengineering process. FAMOOS describes a few tool prototypes that were made withint and for this project. These tools will support them and developers in the reengineering process with various tasks such as visualisation and system reorganisation.

Examples [FAMOOS99]:

- **Name:** GOOSE
  Description: A tool providing automated support for problem detection.
  The reengineering of modern object oriented systems is difficult and costly, due to the size and complexity of these systems. It is an enormous task to read through all the source code in order to learn about the system, so there is a need for automation provided by tools to gain knowledge about the system. This will give the developers more time to focus on the reengineering itself.

- **Name:** DUPLOC
  Description: A tool for detecting duplicated code.
  Duplicated code means, among other things, duplicated errors and general code bloat. Software with duplicated code is also much harder to change. It's important to find duplicated code, however it is a rather cumbersome task and therefor it is desirable to produce tools that can provide automated assistance thereof, and DUPLOC is such a tool.

- **Name:** CodeCrawler
  Description: A tool that supports reverse engineering of large objectoriented projects by combining visualisation with metrics.

The US Air Force Software Technology Support Center (STSC) also presents some various tools for system reengineering. All these tools are also divided into many categories, to clarify the purpose of each tool. These categories are:

- Business Process Reengineering
- Data name rationalization
- Data reengineering
- Forward reengineering
- Object module recovery
- Redocumentation
- Reformatting
- Restructuring
- Retargeting
- Reverse engineering
- Slicing
- Source code translating

All these categories are of course interesting as they all are at some point useful and important in reengineering processes. A brief description of some categories and the tools will now follow.

Reverse Engineering
The tools in this category are made to assist in the first phase of the whole reengineering process, to reverse engineer the legacy system. Many of these tools can read the existing source code and then present it in a way that gives the reengineer a clear overview of the system, such as flowcharts and different types of diagrammes.

Examples [STSC99]:

- **Tool Name:** AutoAnalyzer
  Vendor: Advanced Software Automation, Inc.
  Description: AutoAnalyzer provides a testing and software maintenance environment for software engineers. Builds an interactive structure chart of source code, traces use of global variables, measures and displays test coverage analysis, performance information.

- **Tool Name:** C Design and Documentation Language
  Vendor: Software Systems Design, Inc.
  Description: CDADL helps programming mainly by improving design quality and designer productivity. It analyzes the pseudo-code and executable C code to find errors and to make a printed output report which simplifies the design.

Forward Engineering
Tools under this category are tools to assist in the final phase of reengineering.

Examples [STSC99]

- **Tool Name:** ARIS
  Vendor: Software Systems Design, Inc.
  Description: ARIS is CASE tool which produces Ada code representing a Top Level Ada design.

- **Tool Name:** Auto-G Case Toolset
  Vendor: RJO Enterprises
  Description: The Auto-G Ada Translator translates G&T Design Language (G&TDL) to Ada. It is a complete lifecycle computer aided system engineering toolset.

ReThree-C++ [PBIGGS96]
This is an integrated reverse engineering, redocumentation and reuse tool set, created by Pete Biggs at Brigham Young University. ReThree-C++ can extract information from source code, and create a repository of C++ classes for later retrieval. ReThree-C++ can be divided into three main functions:

1. **Reverse engineering of C++ source code**
2. **Documenting C++ source code.**
3. **Building, maintaining and searching a reuse repository** of C++ classes which can be re-used in later applications.

This tool can automatically reverse engineer C++ code, in order to make a visual presentation of the class hierarchy in OMT object modelformat. Since it is difficult to find software that can view OMT, is free and actually works, we have not been able to view the OMT.

ReThree-C++ can also document the source code. It uses the comments in the source code to generate documentation on the software and saves it in .rtf format. The quality of the produced documentation will therefor be dependant on the developer who wrote the source code. After testing this feature a few times, we found that it worked very well without any problems. This gave us a glimpse of how reengineering tools work and how they can be used in the process.

12. STANDARD METHODOLOGIES

This chapter has been selected to give you an overview of the methodologies that are in use in reengineering projects (both in software reengineering and in BPR). The methodologies are all at a high-level but will follow the general outlook of the reengineering process that we have described so far. We will also look at benefits and drawbacks of each methodology. The names of each methodology has been choosen in a way that they reflect the overall emphasize of that methodology.

Method 1: Continous Improvement
1. Describe the project
2. Create visions, values and objectives
3. Redesign business processes and tools
4. Evaluate concept and benefits
5. Plan for implementation
6. Implement the solution
7. Transition to continous process improvement.

This methodology, as described in the Reengineering (BPR) Methodologies put the emphasize on the continuation of the project after its completion. This is good since it will ensure that the project can stay up-to-date during longer periods of time and thereby reducing the risk of further reengineering. According to some scholars [RM], this method does not take enough time to consider the user needs and thereby increasing the risk of the project beeing unsatisfactory when completed. A project that is easy to change but does not meet the requirements are likely to be discarded since the business management will be less likely to grant another reengineering process. Finally, the methodology has no cost versus benefit step. This is higly undesirable, since the implementation of the new system may prove to be to costly when compared to what is actual gained.

Method 2: Diagnose

1. Define the project
2. Document as-is processes (diagnose system)
3. Redesign business processes
4. Develop a cost/benefit analysis
5. Plan and Implement
6. Evaluate performance

This methodology has, according to [RM], a too time-consuming structure. The documentation of an entire legacy system as-is means a lot of time is spent on the original system and not on how to find ways to meet the new requirements. Of course, it will give the reengineering team a good base on which to plan further reengineering tasks but as the time goes by, the frustration with the inadequate legacy system becomes ever more appearent. It can also means that the team may get hampered in their ability to think in different directions as they get to familiar with the old way of doing things. This is one of the major drawbacks according to the authors [RM].
A good thing is the fourth step, as it deals with a comparison of costs versus benefit. A system may not be worth to reengineer if the costs overshadows the benefits. This step should be carefully examined and after it has been done, the customer must be given the chance to decide on further development lies within his or her interest.

Method 3: Design by learning

1. Create project definition
2. Learn from others (customers, associates, testing, technology)
3. Create vision and design new model
4. Develop to enable good architechture and models.
5. Perform an cost versus benefit analysis
6. Define process, system and training requirements.
7. Plan implementation
8. Develop solutions
9. Implement solutions and measure performance

This a very strong methodology in general. It lacks however the planning for continous improvement which charachterize the first methodology. Not taking this into account may cause the project to be unflexible and hard to update. On the positive side is the careful study of other oppinions, especially the customer part which will give a good insight in what way the system has to change in order to meet the demands and what parts is considered to be adequate enough to keep. This is a clear advantage to the second methodology because it will give good insight into the project without staring at a lot of processes and source code.
The methodology also has a cost/benefit step which also increases its userability.

Method 4: Best-Fit

1. Define project and identify team resources
2. Brainstrom new processes and technologies
3. Analyze and prioritize opportunities (benefit analysis)
4. Select "best" opportunity and design solution
5. Develop to enable the use of tools and other processes.
6. Plan transition
7. Implement solution
8. Measure results

According to the authors [RM] this methodology may be faster than the previous three but it lacks the ability to produce long-lasting products. Although it has a transition step that accommodate for further improvement the fact that the project team has little contact with the costumer and the end-user may cause the whole project to miss its target. No matter how good your ideas are it doesn't help the end-user if they don't provide what that user needs. Maybe it will turn out that the "best-fit" solution is actually only the best in a set of bad solution. Also, the time spend on brainstorming without proper knowledge about the requirements may mean that the process may be forced to return to previous steps to add required functionality that the first solution didn't have. Since such iterative project development ususally means higher costs, especially if the flaws are detected late in the development process, the new system may come out with a much higher price tag than first anticipated.

13. SUMMARY AND NOTES

This small chapter will serve as our conclusion of this report. We believe that the concept of reengineering will prove to be useful and become increasingly important as old software systems, both non-object-oriented and object-oriented, will render obsolete as the flow of information increase. Considering these swift changes, and the massive amounts of resources to keep a software system up-to-date, we believe that new methods and tools will be developed, so that the systems may remain above ground instead of six feet under.

REFERENCES

[BPR99] Jan. K. Collins, *Business Process Reengineering: A USC Perspective,* University of South Carolina 1999.
http://www.research.sc.edu/research/bereview/be46_2/burseeng.htm

[SR] Shim Enterprise, *Software Reengineering*, Shim Enterprise.
http://shiminc.com/reengine.htm

[REFITS94] Tamara J. Taylor, Frank Sparks, *Building a Phased, Structured Approach to Reengineering Legacy Software Systems***,** September 1994.
http://www.stsc.hill.af.mil/crosstalk/1994/sep/xt94d09h.asp

[FAMOOS99] Holger Bär et al, *The FAMOOS Object-Oriented Reengineering Handbook*, 15-10-1999. http://www.iam.unibe.ch/~famoos/handbook

[WHRE96] René R. Klötsch, *Reverse Engineering: Why and How to Reverse Engineer Software,* April 1996.
http://www.infosys.tuwien.ac.at/Projects/CORET

[RM] Jeff Hiatt, *Reengineering (BPR) Methodologies*, 1996.
http://www.prosci.com/mod1.htm

[ICESR99] Rick Dewar et al, *Identifying and communicating expertise in systems reengineering: a patterns approach,* June 1, 1999.
http://www.reengineering.ed.ac.uk/

[STSC99] Karen Rasmussen, *Tool Lists from the STSC's Reengineering Tools Database,* U.S. Airforce Software Technology Support Center, 1999.
http://www.stsc.hill.af.mil/reng

[PBIGGS96] Pete Biggs, *ReThree C++ - A Reverse Engineering, ReDocumentation and Reuse Tool for C++,* Brigham Young University 1996.
http://www.students.cs.byu.edu/~pbiggs/re3-cpp.html