20 Years of Software-Reengineering

A Résumé

Harry M. Sneed

ANECON GmbH, Wien &
University of Regensburg

Abstract: This paper is an attempt to define what software reengineering is and what it has accomplished in the light of 20 years of practical application. The paper points out that reengineering is one of the many software maintenance activities, i.e. everything done with software once it has been put to use. Reengineering actions are devoted to improving the technical quality of existing software. By accepting this definition, it is possible to distinguish reengineering from other related activities performed on a software product after its first release such as reverse engineering, redocumentation, evolution and migration.

Keywords: Software Reengineering, Reverse Engineering, Refactoring, Migration, Maintenance, Evolution.

1 The Field of Software Maintenance

Researchers and practitioners have long since come to the conclusion that software maintenance has to be distinguished from software development. The operations performed upon a piece of software after someone starts using it are different from those performed upon it before its first usage. Tasks performed upon a working piece of software are subject to constraints which do not apply to software under development. Software, which is not being used, can be discarded at any time. The only loss occurred is the loss of time and effort which went in to get the software that far.

With software maintenance it is different. Changing the software in any way may disrupt the functioning of the software and prevent the unfortunate one using it from accomplishing the work or play for which he uses that software. Therefore, according to Boehm in [Boe83] the dominant goal of software maintenance is to preserve the continuity of the service provided by the software. This is what distinguishes maintenance from development and what caused the American Nation Standards Bureau to define maintenance as all work done on a software system after it goes into production [NBS85].

This broad definition of maintenance would have sufficed to cover all activities after the initial release including corrections, changes, improvements and enhancements. In fact, Lientz and Swanson used this definition in their study of maintenance types, concluding that maintenance activities can be grouped into four types:

- corrective maintenance activities
- adaptive maintenance activities
- perfective maintenance activities and
- enhancive maintenance activities [LS80].

All of these activities are concerned with changing the code, the test, the design and for the specification of a software artifact. What distinguishes one change from the other is the motivation for that change. If the change is done to correct an error, i.e. to make the software perform, the way it should have done in the first place, such as using a four digit year instead of a two digit year, it is considered corrective maintenance. If the change is carried out to cause the software to perform an existing function in a different way, such as computing a price in Euros instead of Marks, it is considered to be adaptive maintenance. If the change is carried out to make the software run faster, such as removing code from a loop, or to make the software easier to understand, such as nesting the statements, it is considered to be perfective maintenance. If a change is made to insert a new function, which has not been there before, such as a new link to another website, this is considered to be an enhancement, ergo enhancive maintenance.

The distinction between these maintenance types is important for accounting purposes. The user of the software may be unwilling to pay for error corrections, especially if no one can tell him how many errors will occur. He has no control over that. The same applies to perfective maintenance activities. Since the user was not responsible for the technical state of the software, why should he have to pay to have it improved? Normally, a user is only willing to pay for that what he is responsible for, namely adoptions and enhancements, whereby adoptions are often covered by a flat rate – the so called maintenance fee.

2 The Emergence of Software Reengineering

Software Reengineering was looked upon as a subtype of perfective maintenance as pointed out by this author in an article on software renovation at the Bertelsmann A.G. in 1984 [Sne84]. This was one of the few instances when a user was willing to pay for perfective maintenance. An external contractor was supposed to develop an order entry and dispatching system for the user, but failed to finish on time. Only parts of the system could be put into operation. The code – PL/I and COBOL – was in a bad condition and there was no documentation. The user had decided to terminate the relationship with the contractor and to take over the software and finish it himself. The firm of the author was given a fixed price contract with exact specifications of how the code should look and what documents should be produced. The code remained in the original language, but

was reorganized and restructured so that the user programmers could take it over and finish it according to their own conventions, which they did.

At the same time, in the U.S.A. the government administration was under pressure to reduce the costs that occurred after their IT-Systems went into production. These costs were getting totally out of control. It appeared as if there was no end to software projects even after the products of the projects had been in production for years. This high cost of maintenance was believed to be caused by the poor technical state of the software systems. Therefore, the government set out on a campaign to renovate their existing systems. Consultants were hired and one of them, Robert Arnold, wrote the first book on software reengineering [Arn89]. The government also subsidized several software houses who were building source reengineering tools, since it was obvious that reengineering activities had to be automated (Authors Note: This was before the Indians got into the act).

Actually, such tools had been around since the early 1970's. No sooner did Dijkstra provoke the programming community with his landmark article on "Go To's considered harmful" [Dij68] then did enterprising software houses set out to eliminate them. The problem was well defined and could be readily automated transforming direct graphs into trees. For FORTRAN there was the structured Engine and for COBOL Structured Retrofit from Chris Miller [Par82]. These tools did a fairly good job of transforming unstructured control graphs into structured ones, but initially there was little demand for this service. Programmers were not keen on having their code mangled.

Once the government started promoting the development of such tools, many other software houses entered the game such as Language Technology and Arthur Anderson. There emerged a wealth of literature on program restructuring algorithms and software reengineering processes. Many of the owners of the tool developing companies became rich on government subsidies, but the restructured programs were never really accepted by the maintenance programming community. The maintenance programmers had grown up with unstructured code and saw no benefit to them by changing their programs. It only disrupted the mental map they had of their logic and alienated them from their code. [Sne98]. The transformation of source code, whether by a person or a tool, was always looked upon as a risky endeavor.

The basic goal of any reengineering project is to raise the quality of the product. To prove that this is accomplished the project must begin by defining the current quality of the product. It has to be measured and expressed on some metric scale. Only then can one proceed to the next step and define the goals of the project, i.e. how much the quality should be improved. Usually it requires a quality improvement of at least 20% in order to justify the project. After that a plan is created to achieve that goal and the project started. This way the success of the project can be measured. The problem, of course, is in reaching a consensus on what the quality is and how it can be improved [Sne95].

This author has had many experiences dealing with disgruntled old maintenance programmers over the years. Basically they distrust any attempt to improve their code other than by themselves, and they are particularly wary of automated tools. There were

very few instances after the original Bertelsmann project where reengineering of existing programs was required and then only when the programs involved were being taken over by another team as at Thyssen Steel or where the programs were being prepared for migration as at BBC. Most of the other so called "reengineering projects" the author was involved in, were actually migration projects. Somehow these terms got mixed. The author was very careful to use the term "Software Sanierung" for his book on Reengineering in 1991 so as not to have it confused with migration [Sne91a].

3 The Role of Reverse Engineering

The academic world did not become interested in software maintenance until the late 1980's when the academics began to realize that software reengineering was an interesting field of research. Research centers sprung up at several universities in the U.S. including M.I.T., Georgia Tech and Dexter as well as in Canada at Waterloo and Victoria. In Europe the first universities to deal explicitly with Software Maintenance were Durham in England and Naples in Italy. The first European Maintenance Conference was sponsored by the University of Durham in 1988. The first International Maintenance Conference in Europe was sponsored by the University of Naples in 1991. The first Workshop on Reverse Engineering took place in Baltimore in 1993.

The academics brought in a new view of the problem. Until them, the practitioners were mainly concerned with a direct translation of one program structure into another. For the academics, this approach was much too primitive. To be worthy of academic research, it had to become more sophisticated. That meant the original programs should first be transformed into a higher order design language which the human reengineer could then partition and reorder. Only after several consecutive transformations, could the abstract design then be used to regenerate the code. No one in industry would have ever come up with this approach because they knew well that their customers had no time to deal with intermediate languages. They expected the tool to produce a final result.

The best review of this multilateral model was published in 1990 by Cross and Chikofsky [CC90]. When it comes to defining terms in software maintenance no publication is cited more than this one. Based on the research of Rich and Waters at M.I.T. [Wat88] the two authors point out that to reengineer a program, one should start out by extracting the design form the code, redesign the program and then transform the new design back into code. They also recommended extracting the specifications from the design in order to bring the requirement specification inline with the implementation. If this were done, the user might even be able to respecify the system. Two years earlier, this author had published a paper on „Inverse Transformation of Code into Specifications" in which he demonstrated how this could be done [SJ88].

Unfortunately, the stakeholders of the author's research were forced out of business by the collapse of the socialist system, so that the author was never able to deliver the proof of his concept. Instead he was obliged to move into migration where there was a definite need for automated support. However others carried on with this research and developed highly sophisticated tools for presenting program designs. This came to be known as

„Reverse Engineering" which is the same as extracting a plan from a product. It is considered by many to be a prerequisite for reengineering assuming that one must understand a software system in order to be able to revise. However, this depends on the type of revision. If only formal transformations are required such as restructuring the control flow graph, reformatting the code, repartitioning the code into separate modules or removing the IO operations, there is no need for reverse engineering since there is no need to comprehend the system. Reverse Engineering is only needed if one wants to redesign a component or to revise the entire architecture of a system. Then one must indeed understand the old system.

Nevertheless, as it turned out, reverse engineering proved to be a useful technology on its own rights. It can provide documents and views which help the maintenance programmer to make better decisions as to what and how to change the code and data structures. Thus, reverse engineering is mainly useful as a tool for the maintenance programmers to help them perform better. It is seldom considered a separate project unless the owner of a software system wants to redocument it. More over reverse is a prerequisite to impact analysis, which is a highly important method of calculating maintenance task costs and avoiding undesired side effects [Sne01].

4 Why Software Evolution?

The practitioners and benefactors of software maintenance activities including reverse and reengineering could have go on well with the term software maintenance and the four categories of maintenance established by Lientz and Swanson in 1981 forever. However, the academicians were disturbed by the term „maintenance". To them maintenance implied repairing something or putting it into a state in which it should have been in the first place. As such it has a negative connotation. Persons involved in maintenance are considered to be second or even third class developers incapable of conceiving any new solutions, only capable of repairing or modifying the solutions of others. Certainly it could not be applied to perfective and enhancive maintenance where engineers have to improve and enhance an existing system.

Therefore, as Goethe remarks in Faust, if you are at an impasse in the language, just introduce a new notion. The new notion was „Evolution". Evolution became very popular in the 1990's in connection with iterative software development. It had proven to be impractical to try and develop an entire software system at one time. People recognized that building information systems is not like building bridges or even cars. Information systems have to evolve. Tom Gilb had already pointed this out in the late 1980's in his book on „Evolutionary Software Engineering" Gi88]. Ken Beck picked up the idea in the 1990's and made a religion out of it – Extreme Programming [Bec98].

For software technicians working on iterative or evolutionary development projects, the distinction between development and maintenance which had once been so clearly defined by the NBS for accounting purposes began to blur. No sooner was the first component half way executable, then it went into production. The rest was soon to follow. In such a situation, maintenance is running parallel to development and it is

indeed difficult to distinguish between the two. So it follows that everything can be joined together under a common notion such as „Evolution".

Thus, in 2000 the International Journal of Software Maintenance was renamed the Journal of Software Maintenance and Evolution. There was also an attempt to rename the International Conference on Software Maintenance but unfortunately, this proposal was dropped. Since names are only labels whose underlying meaning can be reinterpreted at will, there is no need in changing them unless they are politically incorrect. Maintenance may not be appealing to young developers but it is not politically incorrect.

The advocates of Software Evolution went on to publish a model describing the life cycle of a software product with five phrases:

- conception
- development
- evolution
- maintenance
- retirement [BR00].

Software evolution is seen here as the period after initial development where the product is still growing and being perfected. It is here that reengineering optimization and functional enhancement take place. Later in maintenance only corrections and unavoidable adaptations are made. Otherwise, the product is frozen.

For those involved in the individual activities of maintenance as all work done on a software product after it goes into operation, it really doesn't matter what general notion they are working under. Correcting an error or adapting a function to produce another result is clear to the person doing the job. Restructuring a program or refactoring a class is equally clear provided the goals are defined. It doesn't matter whether it is called maintenance or evolution. The problem with reengineering is that the goals are often not well defined.

5 Software Reengineering and Refactoring

Refactoring was brought into play in 1990's by Martin Fowler in connection with object–oriented development and extreme programming [Fow99]. The GOTO branches of procedural programming gave way to the polymorphic methods calls and the many associations between neighboring classes in object–oriented programming. The deeply nested procedural code gave way to deeply nested class hierarchies and the problems with oversized, uncommented code blocks, meaningless variable names, dynamic type changes, etc. remained.

In effect refactoring is a form of restructuring. One is adding new nodes and corners to a graph and eliminating other. Only here the nodes are classes and methods and the corners are generalizations and associations. Be that as it may, new terms must be found for every new technology, so as not to be associated with the old technology. Refactoring is now considered to be a fact of life in object–orientation, since no one seems able to come up with the right objects the first time. In extreme programming projects it is accepted that developers spend over half of their time refactoring their code which, in the case of restructuring, would never have been accepted, even though the two activities are actually equivalent. They are both reengineering measures with the same goals, namely that of improving the technical quality of the product but being performed on differently structured software.

Today, when you speak with a young developer about software reengineering he will always associate that term with refactoring because he has been conditioned to think in those terms. In effect, he could just as well use the term restructuring to describe what he is doing. The two terms describe the same type of activity. What differ are the way that the activity is carried out and the objects which one is dealing with. With restructuring one is dealing with procedural control flows, with refactoring one is dealing with object / method structures.

6 Software Reengineering and Software Migration

The term "software migration" predates software reengineering by many years. As early as the 1960's user were migrating from one operating system or from one database system to another. They also were converting from one language to another, such as from Assembler to COBOL. There was never any thought on improving the quality of the software. The only goal was to get the software from one environment to another.

Neither the goal not the basic methods have changed since 40 years. Today the goal is still to transport the data and/or programs from one environment to another without affecting the functionality. It may be that the language is changed but the performance should remain the same. There is never a mention of technical quality in the migration contracts. There may be, as a side effect of the architecture and/or language, conversion, a quality improvement, but this is never an explicit goal.

As a migration contractor, the author often made the mistake of trying to combine migration with reengineering. He thought that the user should use this opportunity to improve the internal quality of his software. By doing this, he only jeopardized the real goals of the project and risked acceptance of the results by the responsible programmers. Software academics must face the fact that there is no universally accepted notion of what software quality is. It depends on the culture of the organization. What is good in one organization may be totally unacceptable in the next one. Therefore it is better to leave this topic out of migration projects [Sne99].

If the migrated product performs in the new environment the same functions with the same reliability and the same performance as in the old environment then the migration

is a success. The old and new versions should be functionally equivalent. Improving the quality of the code and/or architecture is another matter, which should be dealt with either before or after the migration project, but never during it. The reasons for this are of an economic nature. Improving the quality of the code introduces additional risks which can lead to unexpected costs. Since almost all migration projects operate on a fixed price which is only paid when the system is running in the new environment a good project leaders will always strive to fulfill the minimum requirement necessary in order to receive payment. There is as yet no accepted method of attaching economic value to quality improvements although there is promising research on this subject [Hah05].

7 The Future of Software Reengineering

When discussing the future of Software Reengineering, we must be careful to distinguish reengineering from the related activities of reverse engineering and migration. Reengineering has its own goal and that is to improve the quality of the software. That is what the term was originally intended for and what it should stick to. This being the case, reengineering projects will continue to be rare. Users have never seen the need to invest in the internal quality of software and there is no indication that they will change in the near future. The relation between internal and external quality and between quality and maintenance costs is not clear to them. For those who have control over the funds there are much more important priorities. Where reengineering is more likely to be financed is by software product vendors who have a vested interest in making their product more flexible and extendable.

Normally, if reengineering is to take place at all, it will be carried out by the maintenance programmers themselves within the context of perfective maintenance. Before making a change they may want to refactor a class or restructure a procedure. If there is a tool available they may use it. It is unlikely that it will be prescribed. This fact implies that reengineering technology must become part of the overall maintenance training. In becoming a certified software maintenance engineer students should be exposed to reengineering techniques and learn to apply them. Reengineering should be part of their job to preserve and improve the quality of their product.

Of course, there will always be occasional reengineering projects in which owners of strategically important software systems will make a one time investment to clean up these systems. For this they need reliable metrics with which they can measure the degree of improvement. Users must be able to relate their investments in reengineering to reductions in maintenance costs. Maintenance personnel must be convinced that better structured or better factored code will make their job easier. Unfortunately, this case has never really been accepted [Art93].

Unfortunately, this is not the case today, at least not in the IT area. Managers can neither readily relate reengineering results to cost savings, nor can maintainers perceive how reengineering helps them to do a better job. The one is a problem of measurement, the other a problem of education. Until these problems are solved, software reengineering

technology will not get the attention it deserves, as opposed to the field of software migration where users are willing to pay any price to port their applications to the latest technical environment. Here it is not the quality of the software which matters but the quality of the environment and the continuity of the service.

This observation confirms the claim of Nicolas Carr that IT doesn't really matter, at least not its quality [Car03]. The only thing that is important is that it functions at all. In the case of IT applications systems, quality is not an issue. Therefore, reengineering is not an issue. Where quality matters, is where the software is itself a final product or part of another product, such as in embedded systems. Here, there will be an increasing need for reengineering technology to improve the competitive position of the product. Such products should be in the focus of reengineering research.

References

[Arn89] Arnold, R.: Software Reengineering – A Tutorial. IEEE Computer Society Press, Los Alamos, 1992.

[Art93] Arthur, L.: Improving Software Quality – does it matter? John Wiley & Sons, New York, 1993, p. 203.

[Bec98] Beck, K.: Extreme Programming explained. Addison-Wesley Press, Reading, 1998.

[BR00] Bennett, K. H.; Rajlich, V. T.: Software Maintenance and Evolution: A Roadmap. In Finkelstein, A. [Hrsg.]: The Future of Software Engineering, Proceedings of the International Conference on Software Engineering, ACM Press, 2000.

[Boe83] Boehm, B. W.: The Economics of Software Maintenance. IEEE Proc. of 1st ICSM, Computer Society Press, Monterey, CA, 1983, p. 9.

[Car03] Carr, N.: The big Switch. Harpers Books, New York, 2007.

[CC90] Chikofsky, E. J.; Cross, J. H.: Reverse Engineering and Design Recovery: A Taxonomy. IEEE Software, Vol. 23, No. 1, 1990, pp. 13-17.

[Dij68] Dijkstra, E.W.: GOTO statement considered harmful. Comm. of ACM, Vol. 11, No. 3, 1968.

[Fow99] Fowler, M.: Refactoring – Improving the Design of existing Code. Addison-Wesley, Reading, 1999.

[Gil88] Gilb, T.: Principles of Software Engineering Management. Addison-Wesley, Wokingham, G.B., 1988.

[Hah05] von Hahn, E.: Werterhaltung von Software. Deutscher Universitätsverlag, Wiesbaden, 2005.

[NBS85] National Bureau of Standards: NBS Special Publication 500-106 - Guidance on Software Maintenance. Martin, R. & Osborne, W., Washington, D.C., 1985.

[LS80] Lientz, B.; Swanson, E.: Software Maintenance Management: A Study of the Maintenance of Computer Application Software. In 487 Data Processing Organizations, Addison Wesley, Reading, MA., 1980.

[Par82] Parikh, G: Techniques of program and system maintenance. Little, Brown & Company, 1982, p. 181.

[Sne84] Sneed, H. M.: Software Renewal – A case study. IEEE Software, Vol. 1, No. 3, July, 1984, p. 56.

[SJ88] Sneed, H. M.; Jandrasics, G.: Inverse Transformation of Code to Specifications. IEEE Proc of 4th ICSM, Computer Society Press, Phoenix, Oct. 1988, p. 302.

[Sne91a] Sneed, H.M.: Software Sanierung. Rudolf Müller Verlag, Köln, 1991.

[Sne91b] Sneed, H.M.: Economics of Software Reengineering. Journal of Software Maintenance, Vol. 3, No. 3, Sept. 1991, p. 163.

[Sne95] Sneed, H.M.: Planning the Reengineering of Legacy Systems. IEEE Software, Vol. 12, No. 1, Jan. 1995, pp. 24-34.

[Sne98] Sneed, H.M.: Human Cognition and how Programming Languages determine how we think. IEEE Proc. pf 6th IWPC, Computer Society Press, Ischia, June 1998, p. 1.

[Sne99] Sneed, H. M.: Risks involved in Reengineering Projects. Proc. of WCRE, IEEE Computer Society Press, Atlanta, Oct. 1999, pp. 204-211.

[Sne01] Sneed, H. M.: Impact Analysis of Maintenance Tasks. Proc. of 17th ICSM, IEEE Computer Society Press, Florence, Nov. 2001, p. 180.

[Wat88] Waters, R.: Program Translation via Abstraction and Reimplementation. IEEE Trans. on Software Engineering, Vol. 14, No. 8, August 1988, p. 1207.