Iterative Reengineering to compensate for Quick-Fix Maintenance

Filippo Lanubile, Giuseppe Visaggio

Dipartimento di Informatica - University of Bari, Italy

Abstract

A typical approach to software maintenance is analyzing just the source code, applying some patches, releasing the new version, and then updating the documentation. This quick-fix approach usually leads to documentation not aligned with the current system and degrades the original system structure, thus rendering the evolution of the system costly and error-prone. Although there are alternative maintenance models which avoid these problems, by analyzing and updating the system documentation first, the quick-fix approach continues to be popular because of the time pressure for new releases and the resistance to change of maintenance programmers.

In this paper, we propose an iterative reengineering model which can be run each time the maintainability and reliability of a software system degrade under a tolerance level. The reengineering process, applied after a number of modifications, can result in renovation of the current system or, simply, in realignment of the documentation. In this context, reengineering is no longer a one-shot process but becomes an ordinary process which runs concurrently with the quick-fix maintenance process. The results obtained with an industrial case study are presented and the lessons learned are discussed.

1: Introduction

In the last few years, the contribution of researchers and practitioners in the reverse engineering field has been very rich of proposals and applications in all the key objectives pointed out in [5]: coping with complexity, generating alternate views, recovering lost information, detecting side effects and analyzing quality, synthesizing higher abstractions, facilitating reuse, and populating a repository or knowledge base. Many research studies and industrial experiences have also been published on reengineering old systems in a new form to decrease maintenance costs, reduce software errors, or upgrade to a new hardware [2]. However, very little has been written regarding what happens in the maintenance process after a software system has been reengineered or simply reverse engineered.

Consider the three maintenance process models, proposed by Basili in [3]: quick-fix model, iterative-enhancement model, and full-reuse model. All three models assume that the existing system has a complete and consistent documentation from requirements to code. This assumption can be realistic if we consider the configuration of a system after that a reverse engineering process has been applied. The quick-fix model, shown in Figure 1, starts modifying the existing source code, and then test the new version and modify the system documentation. It represents an abstraction of the common approach to handling software maintenance where all changes stem directly from the implementation abstraction level. The iterative-enhancement model, shown in Figure 2, starts with an analysis of the existing system's documents. The highest level artifact which is affected by the request of change is then modified and the modification is propagated downward through the lower abstraction levels. At each step, the system is redeveloped based on the analysis of the existing system. The full-reuse model, shown in Figure 3, differ from the iterative-enhancement model, because it assumes that there exists a repository of software artifacts from the current and earlier versions of the subject system or similar systems. It starts with the analysis of the requirements for the new system and rebuilds the system by reusing whatever existing document or component is applicable, or developing them when necessary.

The full-reuse model promotes the development of reusable artifacts and encourages their reuse also in modification tasks, while the iterative-enhancement model try to accomplish changes by adapting the existing system. Both models differ from the quick-fix approach because the analysis and update of documentation is performed before the modification of the code. This allows maintainer programmers to have a control on the increasing entropy as the system evolve [7].

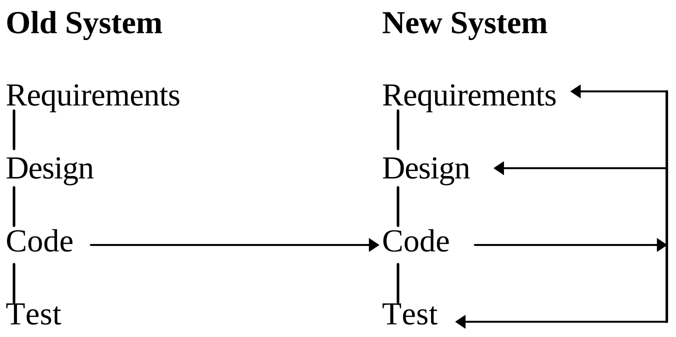

Figure 1. The quick-fix model

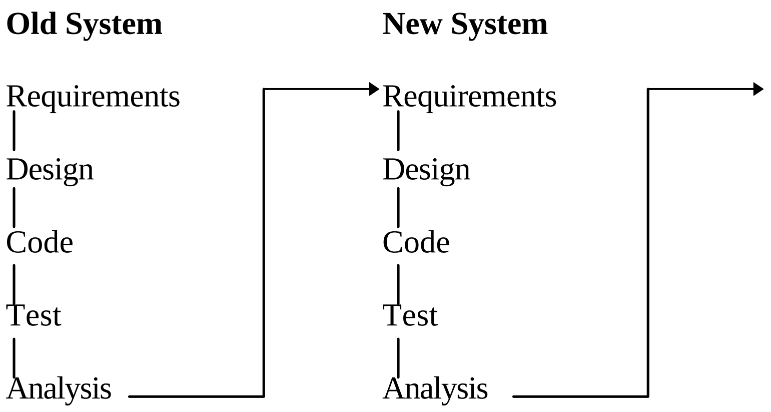

Figure 2. The iterative-enhancement model

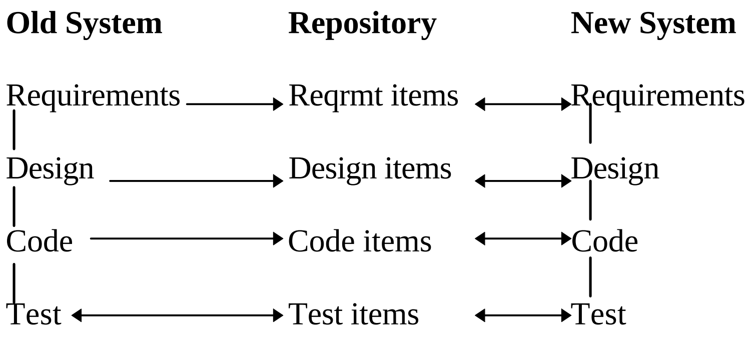

Figure 3. The full-reuse model

However, as the same Basili points out, most maintenance gets done using the quick-fix model, since the time pressure hangs over long-term cost savings due to preserved technical quality. Another cause of this persisting popularity is that old habits are hard to die. Maintenance programmers are often application experts which know well the application but do not like to document, or inexperienced programmers without maintenance training, or personnel which suffer the maintenance work. In all these cases, it is difficult to follow a maintenance process different from the quick-fix approach. Unfortunately, with the quick-fix model, documentation is not often updated as the system is modified, and after a number of patches the system structure becomes lost and the system itself no more modifiable.

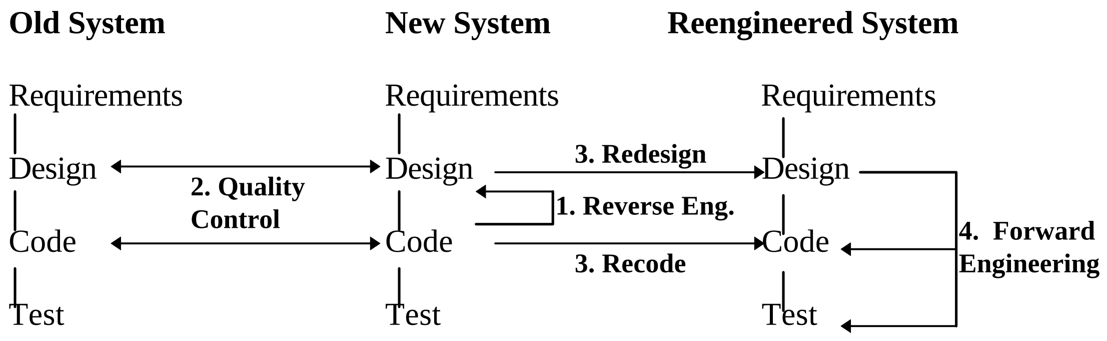

Figure 4. The iterative reengineering model

In this paper, we propose a reengineering model which can be performed to compensate for the drawbacks of the quick-fix model, when no alternative maintenance model can be followed by the maintenance organization. The reengineering model is iterative because it can be reapplied each time the system documentation needs to be realigned and the evolution of the system has become difficult and error-ridden.

The following section presents the iterative reengineering model and its relationships with the quick-fix maintenance process. The third section describes an industrial case study and the results achieved. Finally, conclusions present the lessons learned and point out the future work.

2. The Iterative Reengineering Model

The reengineering process and the quick-fix maintenance process work together as concurrent processes which share the existing system. The iteration rate of the reengineering process is not tied to the modification rate but to the effect of modifications to the system evolution. Each time the cost of changes becomes intolerable and the reliability decreases under an acceptable threshold, an iteration of the reengineering process starts.

The current version of the reengineering model deals only with changes which do not affect the conceptual characteristics of original requirements. This assumption is realistic because the advantages of the quick-fix approach consist in a timely response to system failures, technological changes, or expanding needs of the users which modify the form of presenting information but not the essence of the problem. In other words, the iterative reengineering model is not compatible with requirement changes, which should be performed using an alternative maintenance approach such as the iterative-enhancement model or the full-reuse model.

Figure 4 demonstrates the flow of change from the old and new systems to the reengineered system. The model is decomposed in four stages: (1) reverse engineering, (2) tracking quality, (3) redesign/recode, and (4) forward engineering. The model is coherent with the *rework* strategy of software reengineering [4] which incorporates the principles of abstraction, alteration, and refinement.

The first step, *reverse engineering*, is used to rebuild the system representation at the design level. Diagram generators create charts from code to describe the control and data flow, call hierarchies, and data structures. These diagrams combined with existing design documentation, personnel experience and domain knowledge make it possible to reproduce all the information required at the design abstraction level. A number of software technologies are today available, including traditional CASE tools themselves, which incorporate reverse engineering capabilities both for data and functions.

The second step, *quality control*, is the evaluation of software components against predetermined critical values of metrics or the tracking of the quality of a component over successive versions [8]. For example, if the cyclomatic complexity of a module in the new system grows with respect to that of the previous version in the old system, it could indicate a degradation in quality and so determine where redesign/recode efforts should be concentrated. Since for large-scale systems, it is impracticable performing metric analysis by hand, metric collection tools are required to parse the source code or the design repository and generate metric values. We use a broader interpretation of quality control, which is not only metric-driven. Old design documents are compared with the new rebuilt design documents to verify that design changes have not resulted in a worsening of data normalization, cohesion, coupling, or information hiding. New code is checked to verify that the changes are conforming to the coding standards used in the old code. This comparison requires that the system is under configuration management.

The quality control step guides the activation of the next steps according to the following procedure.

```
If the design quality of the new system is worst than the
design quality of the old system
                    Activate Redesign
Else If the code quality of the new system is worst than
the code quality of the old system
                    Activate Recode
Otherwise           Exit
```

In the last case, when the system quality is proven to be unchanged both at design and code level, the reengineering process stops, resulting in a reverse engineering process which realigns the system documentation.

The third step, redesign/recode, makes changes which improve the quality of the system. Redesign modifies design characteristics like database schemes, data structures, module decomposition, and module interfaces. Recode comprises changes of implementation characteristics like renaming program items, control flow restructuring, and including meaningful comments. Both redesign and recode change the system representation without changing the level of abstraction. Many commercial recode tools are available for converting unstructured programs into structured programs, or improving the format of the source code.

The fourth and last step, forward engineering, is performed to upgrade those parts of the systems which have been redesigned, and to ensure that the system still works. Lower CASE tools and testing tools can be used to support these activities.

3. Experiencing the iterative reengineering model

The experience was performed on a banking information system, more than fifteen years old, made up of COBOL programs which run on a mainframe. A previous reengineering project, described in [1], had recovered lost requirements and design information, ending with a renovation of the degenerated parts.

This second iteration of the reengineering process concerned four subsystems which had been under quick-fix maintenance for six months after the first reengineering. The only available information was that the patches had not affected any characteristic of requirements. Four programmers were involved for a period from four to six months, each reengineering a separate subsystem. Data structures were rebuilt from COBOL File Sections, using a data modeling tool with reverse engineering capabilities developed by Basica but not yet on the market. Quality control revealed no modifications to the data model. Other design documentation was reconstructed from COBOL Procedure Divisions using ViaSoft’s ESW. Figure 5 shows the call graph of a sample program.

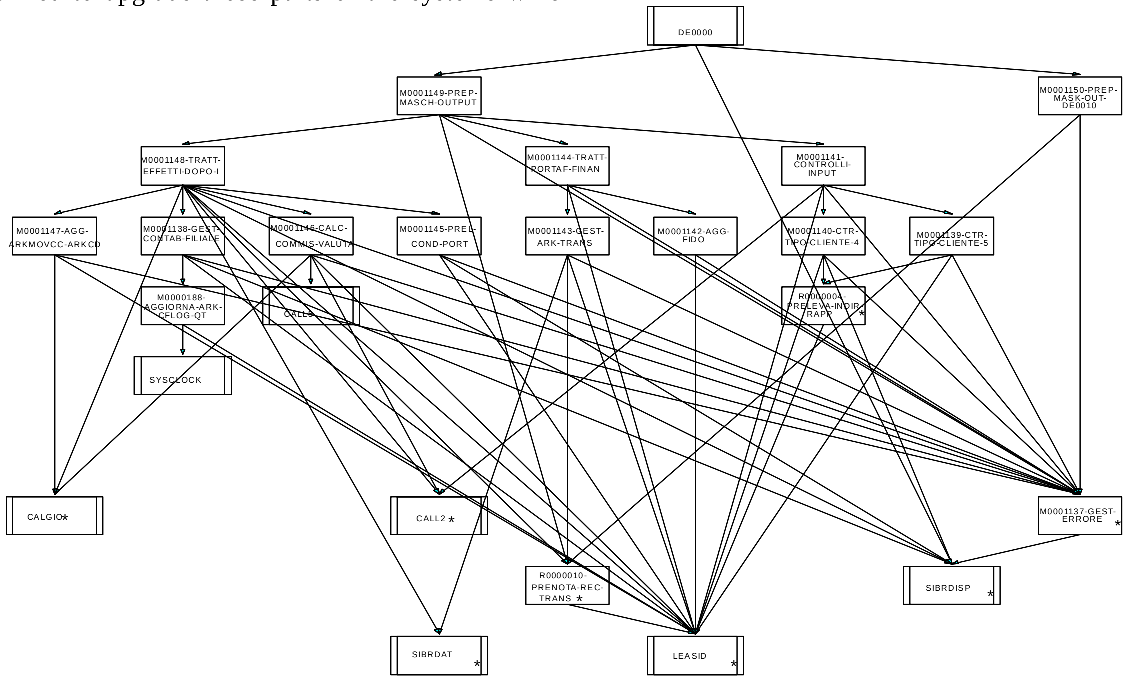
DE0000
M0001149-PREP-MASCH-OUTPUT
M0001150-PREP-MASK-OUT-DE0010
M0001148-TRATT-EFFETTI-DOPO-I
M0001144-TRATT-PORTAF-FINAN
M0001141-CONTROLLI-INPUT
M0001147-AGG-ARKMOVCC-ARKCD
M0001138-GEST-CONTAB-FILIALE
M0001146-CALC-COMMIS-VALUTA
M0001145-PREL-COND-PORT
M0001143-GEST-ARK-TRANS
M0001142-AGG-FIDO
M0001140-CTR-TIPO-CLIENTE-4
M0001139-CTR-TIPO-CLIENTE-5
M0000188-AGGIORNA-ARK-CFLOG-QT
CALL5
R0000004-PRELEVA-INDIR-RAPP *
SYSCLOCK
CALGIO *
CALL2 *
M0001137-GEST-ERRORE *
R0000010-PRENOTA-REC-TRANS *
SIBRDISP *
SIBRDAT *
LEASID *

Figure 5. Call graph of a sample program

	Old System			New System		
Modules	LOC	v(G)	V	LOC	v(G)	V
M0001140	274	62	12735	279	56	11637
M0001141	187	32	8217	201	33	8334
M0001142	54	25	5496	54	25	5496
M0001143	59	6	2296	59	6	2296
M0001144	54	6	3272	54	6	3272
M0001145	59	12	3480	59	12	3480
M0001146	147	35	5952	153	33	5648
M0001147	212	36	8415	212	36	8415
M0001148	242	37	17990	223	38	17690
M0001149	36	9	2336	41	8	2008
M0001150	42	2	2023	42	2	2023

Table 1. Metric values per module and version

	Before		After	
	LOC	Modules	LOC	Modules
Subsystem 1	290 000	5 783	314 823	5 724
Subsystem 2	273 525	5 675	281 152	5 623
Subsystem 3	285 732	5 729	238 140	5 670
Subsystem 4	274 315	5 266	267 138	5 238
Total	*1 123 572*	*22 453*	*1 101 253*	*22 255*

Table 2. Size before and after the second iteration

Quality control on functional components was performed by tracking changes for their effect on three software metrics produced by the ViaSoft tool.

Table 1 shows the metric values, including LOC, cyclomatic complexity and Halstead's volume, related to a subset of modules. Modules with the same metric pattern (M000142-M0001145, M000147, M0001150) were considered unchanged. Modules with higher control-flow complexity values (M000141 and M0001148) were inspected to discover weakness points at the design and code level abstraction. Reading comments in the module headings, a number of new modules were discovered, which had been introduced by duplicating some existing module or coding a function already present. Thus, redesign was mainly aimed at eliminating redundant components. During the forward engineering step, testing revealed undiscovered failures and corrective changes were then applied, using an iterative-enhancement model.

Table 2 outlines the differences in size before and after this second iteration of the reengineering process. While the first two subsystems grew in size to meet the expanding needs of users which required new reports or different forms of reports, the decreased size of the last two subsystems is the effect of having cut off the superfluous parts, a sign of system degradation.

Table 3 reports effort and productivity data of this second iteration of the reengineering process. Data were collected from report forms which were filled and collected on a daily basis. The work-day report form was the same used in [6] to analyze data of the first iteration. Effort is measured in person-hours with a precision of one half hour, while productivity is measured as the number of lines of code revised per programmer-hours of effort. Data of the two examination steps, (1) reverse engineering and (2) quality control, are combined together and compared with data of the two change steps, (3) redesign/recode and (4) forward engineering.

	Effort (Person Hours)			Productivity (LOC/PH)		Productivity Ratio
	Steps (1)+(2)	Steps (3)+(4)	Steps (1)+(2)+ (3)+(4)	Steps (1)+(2)	Steps (3)+(4)	Steps (1)+(2) / (3)+(4)
Subsystem 1	197.5	561.5	759.0	1594	561	2.84
Subsystem 2	165.5	770.5	936.0	1699	365	4.65
Subsystem 3	159.0	498.5	657.5	1498	478	3.13
Subsystem 4	167.5	500.0	667.5	1595	534	2.99
Total	*689.5*	*2330.0*	*3019.5*	*1630*	*482*	*3.38*

Table 3. Efforts and productivity measures per subsystem and model step

There are no significant differences, both in effort and productivity, among the four subsystems, while the examination and change steps have different performances at the 0.001 significant level. The productivity in reconstructing documentation and analyzing system quality is more than double that in changing the existing system. This ratio is due to a difference weight of software technologies in supporting the steps of the reengineering model. The changing steps are more labor-intensive than the examination steps.

4. Conclusions

We presented an iterative reengineering model which can be used in organizations where the quick-fix approach to software maintenance is a current practice. The iteration is caused by the periodic need for reconstructing software documentation and preventing software system from degradation.

Lessons learned from running the second iteration of the reengineering process are the following.

- Quick-fix maintenance can rapidly deteriorate the maintainability and reliability of a system: iterative reengineering can be a solution for those software organizations which are not able to adopt more mature maintenance models, because of time or lack of discipline.
- When software developers retain maintenance responsibility, the reengineering process can be managed by a separate reengineering staff or a separate reengineering organization.
- Reconstructing software documentation accounts for less than one fourth of the overall reengineering costs: we need to improve the productivity of change phases using a common repository which integrates the multiple tools used for reverse and forward engineering.
- Econometric reengineering models [9] which decide whether to replace, redevelop, reengineer, or continuing maintenance, should take into account the realignment of documentation as a separate option, so that software managers can decide if altering the existing system, after having rebuilt documentation, can be worthwhile.

Future work is aimed to give an answer to the open issues, and to continue the collection of process and product data for further analyses.

Acknowledgements

We would like to thank Gregorio Fatone for his help in collecting and analyzing the data. This work was partially supported by national funds MURST 40%, project “Methods for producing reusable components from existing software”, and MURST 60%, project “Maintenance of heterogeneous systems”.

References

[1] F. Abbattista, G. M. G. Fatone, F. Lanubile, G. Visaggio, “Analyzing the application of a reverse engineering proxess to a real situation”, *Proceedings of the 3rd Workshop on Program Comprehension*, Washington, D.C., November 1994, pp.62-71.

[2] R. S. Arnold, *Software Reengineering*, IEEE Computer Society Press, Los Alamitos, California, 1993.

[3] V. R. Basili, “Viewing maintenance as reuse-oriented software development”, *IEEE Software*, January 1990, pp.19-25.

[4] E. J. Byrne, “A conceptual foundation for software re-engineering”, *Proceedings of the Conference on Software Maintenance*, Orlando, Florida, November 1992, pp.226-235.

[5] E. J. Chikofsky, and J. H. Crossburn, “Reverse engineering and design recovery: a taxonomy”, *IEEE Software*, January 1990, pp.13-17.

[6] P. Fiore, F. Lanubile, and G. Visaggio, “Analyzing empirical data from a reverse engineering project”, *2nd Working Conference on Reverse Engineering*, Toronto, Ontario, Canada, July 1995.

[7] M. M. Lehman, “Programs, life cycles, and laws of software evolution”, *Proceedings of the IEEE*, vol.68, no.9, September 1980, pp.1060-1076.

[8] N. F. Schneidewind, “Methodology for validating software metrics”, *IEEE Transactions on Software Engineering*, vol.18, no.5, May 1992, pp.410-422.

[9] H. M. Sneed, “Economics of software re-engineering”, *Software Maintenance: Research and Practice*, vol.3, no.3, 1991, pp.163-182.