The Role of Program Structure in Software Maintenance

Jaana Lindroos

Helsinki, 27th of February 2005

Seminar on Software Maintenance

Department of Computer Science

UNIVERSITY OF HELSINKI

The Role of Program Structure in Software Maintenance

Jaana Lindroos
Seminar on Software Maintenance
Department of Computer Science
UNIVERSITY OF HELSINKI
27th of February 2005, 8 pages

Abstract

It has been mentioned that application should be designed for maintenance [AlC98]. Maintainability is defined as the ease with which systems can be understood and modified [GiS89]. Maintenance is required for old legacy systems as well as new developed systems, which for example use object-oriented (OO) languages. It is well known that software maintenance consumes the majority of the costs of a software system during its entire life; estimates vary between 50 and 70 % [DoL97].

For software systems that are under development the target is to reduce maintenance of software system already in early phase of software development. In that process measuring program structure can be used as a guideline for producing better code that contains fewer errors after the software system is released to the market. Old legacy systems that are in maintenance phase can also utilize software metrics that measures programs structure. In that case the target is to provide early estimates for allocating resources and estimating costs for maintenance tasks.

Maintenance is a very challenging and time consuming task and researchers have tried to cope the problem with different kinds of empirical and theoretical studies by trying to solve the problem by using program structure metrics in different phases of software development and maintenance processes. So far the results of those studies are promising, but there is still lack of common and widely used good practices that could be shared easily to different kinds of environments and languages used.

ACM Computing Classification System 1998:
D.1.5 [object-oriented programming]
D.2.2 [design tools and techniques]
D.2.7 [Distribution, Maintenance, and Enhancement]
D.2.8 [Metrics]
D.2.9 [management: software quality assurance]

Keywords: software maintenance, predicting software maintenance performance, software metrics, object-oriented software metrics, software quality

Contents

1 Introduction

Development and maintenance of the software product is a continuous process [TjL01]. Maintainability is defined as the ease with which systems can be understood and modified [GiS89]. In past studies, it has been operationalized as "number of lines of code changed", time (required to make changes) and accuracy, and "time to understand, develop, and implement modification" [BVT03]. For developers, it is crucial to understand the structure of a system before attempting to modify it [MMC99].

It is well known that software maintenance consumes the majority of the costs of a software system during its entire life; estimates vary between 50 and 70 % [DoL97]. Effort estimation is a valuable asset to managers in planning maintenance activities and performing cost/benefit analysis. Early estimates and accurate evaluations permit to significantly reduce project risks and to improve resource scheduling. Software project costs are essentially human resource costs and this entails that the effort (i.e. man-days needed for system maintenance) should be kept under severe control. [DDS02]

The Object-Oriented (OO) paradigm has become increasingly popular in recent years. The benefits of object-oriented software development are now widely recognized [AlC98]. The insertion of OO technology in the software industry, however, has created new challenges for companies that use product metrics as a tool for monitoring, controlling, and improving the way they develop and maintain software [BBM96]. Researchers agree that, although maintenance may turn out to be easier for OO systems, it is unlikely that the maintenance burden will completely disappear [BVT03].

The maintenance is also required for old legacy software systems. Changes may be driven by market pressure, adaptation to new environments or situations, or improvement needs. During its life cycle a legacy system is continuously subject to ordinary maintenance that includes the interventions that must be reactively implemented to maintain the system in operation, i.e. corrective maintenance and smaller forms of adaptive and preventive maintenance. Periodically, legacy systems are subject to extraordinary maintenance; examples include migration and massive adaptive maintenance [DPP02].

The rest of the paper is structured as follows. Chapter 2 explains how program structure can be measured. In Chapter 3 one empirical study about the role of program structure and maintenance is described. Chapter 4 contains discussion about current and future situation in the role of program structure and maintenance. Chapter 5 concludes the paper and the next Chapter contains references.

2 Measuring Program Structure

Program structure is the way the source code is designed and developed in the software system. The more complex a system is the more difficult it is to understand, and therefore to maintain [GiS89]. Empirical research shows that complex programs require more corrective maintenance throughout their lives [Gre84, VeW83]. Complex programs contain more errors, and errors are more difficult to undercover during testing. Consequently, they remain undetected until late in the software development life cycle.

About half of a software maintainer’s time is spent trying to understand programs. Unfortunately, we cannot measure program understandability directly. Complexity metrics, however, can give some clues as to components that may incur high maintenance costs [HTO94]. The metrics help the engineer to recognize parts of the software that might need modifications and re-implementation. The decision of changes to be made should not rely only on the metric values [SyY99]. The metrics are guidelines and not rules and they should be used to support the desired motivations.

For example in object-oriented languages the intent is to encourage more reuse through better use of abstractions and division of responsibilities, better designs through detection and correction of anomalies. Positive incentives, improvement training and mentoring, and effective design reviews support probability of achieving better results of using object-oriented metrics. [LoK94] This can be done in design and implementation phase of software project in order to ease up upcoming maintenance tasks.

Object-oriented development requires not only different approaches to design and implementation; it also requires different approaches to software metrics. Metrics for object-oriented system are still a relatively new field of study. The traditional metrics such as lines of code [IEE93] and Cyclomatic complexity [McC76, WEY88] have become standards for traditional procedural programs [LIK00, AlC98].

The metrics for object-oriented systems are different due to the different approach in program paradigm and in object-oriented language itself. An object-oriented program paradigm uses localization, encapsulation, information hiding, inheritance, object abstraction and polymorphism, and has different program structure than in procedural languages [LIK00].

If we are going to improve the object-oriented software we develop, we must measure our designs by well-defined standards. Thresholds are affected by many factors, including the state of the software (prototype, first release, third reuse and so on) and your local project experiences. The language used and different coding styles affect some of the metrics. This is primarily handled with different threshold values for the metrics, which indicate heuristic ranges of better and worse values. For example, C++ tends to have larger method sizes than Smalltalk. Thresholds are not absolute laws of nature. They are heuristics and should be treated as such. Possible problems in our system designs can be detected during the development process. [LoK94]

Software should be designed for maintenance [AlC98]. The design evaluation step is an integral part of achieving a high quality design. The metrics should help in improving the total quality of the end product, which means that quality problems could be resolved as early as possible in the development process. It is a well-known fact that the earlier the problems can be resolved the less it costs to the project in terms of time-to-market, quality and maintenance.

Another way of measuring program structures in research area are old legacy systems that are mainly on maintenance phase. The objectives for pure maintenance projects in legacy systems are first of all to get maintainers to understand the software structure they are going to maintain. For developers, it is crucial to understand the structure of a system before attempting to modify it [MMC99]. On the other hand many legacy systems are so large and so complex that they cannot be comprehended in their entirety no matter what forms of representation is used. Yet these programs must be maintained. Erdös and Sneed claim in their study that it is not necessary to comprehend a program in order to maintain it. [ErS98] One of the main targets in legacy system maintenance work is early estimates that can help in resource allocation and estimating time and costs for the project. That can be done using different kind of metrics that measures for example program complexity [DPP01, DPP02, DDS02].

Even though many empirical research studies conclude that less complexity programs reduce the maintenance work, the writer of this paper couldn't find any evident of those conclusions in empirical studies. There seems to be clear division in research area of studying the programs that are in development phase and the programs that are old legacy systems. In development phase of programs the results of the empirical studies will end to the phase where the program is released to the customer. According to my opinion the researches should continue their studies to the maintenance phase, where the results of metrics and estimations could really be validated. The reason for a lack of continuation in following the projects from development phase till maintenance phase might be that empirical researches in real world situations would probably take years until some results could be available. In this study it was found out that there are two different research areas, first for new programs that are under development, and second for old legacy systems and it is very difficult to find connectivity between these two different research areas.

There exists also some criticism that predicting maintenance costs based solely on characteristics of the developed software is not enough even though the approach is very popular among researchers working in the metric field, since software can be measured and the metrics related to effort in correcting or changing the code. The research is useful in helping to identify cost drivers but according to Sneed it is much too limited in scope. [Sne04]

3 Empirical Study

De Lucia, Di Penta, Stefanucci and Venturi have performed a case study, where they present an approach for an early effort estimation based on the knowledge of a fraction of the programs composing the work-packet. The proposed approach has been applied to a large massive maintenance project performed by a major international software enterprise, namely EDS Italia Software (EDS SC), in conducting massive adaptive maintenance projects with a close deadline. The researchers' aim was to analyze and assess the stability of the process and to provide indications for future projects. [DDS02]

The assessment was performed on only a subset of the software system in order to reduce time and costs. The researchers' used regression analysis to build effort estimation models validated against real data collected from a large Y2K remediation project. The resulting model allows estimating the costs of a project conducted according to the adopted massive maintenance process. [DDS02]

The analyzed project process was similar to earlier processes where EDS SC has conducted several Y2K and EURO conversion projects. EDS SC has adopted a process based on a preliminary assessment and inventory phase, aiming to decompose an application portfolio into loosely coupled sets of cohesive components called *work-packets* (WPs) that can be independently and incrementally modified and delivered. Overall, the massive adaptive maintenance process used by EDS SC was based on a waterfall model composed of six phases: portfolio inventory and assessment, analysis, design, implementation, testing and delivery-installation. More details on the process can be found in [DPP01].

The analyzed project consisted about 40,000 software components from which about 15,000 components were modified during maintenance process. The guideline for researchers' work was composed of three steps:

- Collection of the information (applications, software platform, programs and their location);
- Static analysis of the objects of each application (executed through automatic tools); and
- Analysis of the results for each application.

The analysis of the results focused on the identification and interpretation of some maintainability indicators grouped in the following categories:

- Size (number of programs, lines of code);

- Complexity (Cyclomatic complexity, Software Science Volume, control variables);
- Structure (control flow knots), and;
- Other (unused copybooks, number of go to statement).

The project was used to identify the relationships between the costs, as identified by the effort required to modify and make Y2K compliant a WP, and the size of the project, measured through the metrics shown in Table 1. Data were collected from a sample subset composed of 14 WPs.

Metric	Description
SC	Number of software code components of the WP
LOC	Lines of code, including comments, blanks, and declarative lines[1]
CYC	McCabe Cyclomatic complexity [McC76]
CVAR	Number of control variables
VOL	Halstead Software Science Volume [Hal77]
UPL	Number of logical branch not used
Effort	Actual effort of the WP measured as man-days

Table 1. Used Metrics

For each work packet the effort spent, the number of source files maintained, and the mean values of the metrics were calculated. It was supposed that the maintainer has available data about a given percentage of programs of each WP, to be used to estimate the effort to maintain the WP itself.

The researchers' chose only an appropriate set of metrics to model the effort. In order to do this they used correlation analysis. The chosen metrics that they took into account in their model were:

- The number of programs composing each WP (SC)
- The mean value of, at least, one dimensional metric, say $\overline{\text{LOC}}$; and
- The mean value of, at least a structural metric, say Cyclomatic complexity $\overline{\text{(CYC)}}$.

In other words, the idea was to predict the effort on the *whole*, given some characteristics (i.e., metrics) from on of its *part*, and what percentage of the whole each *part* represents, assuming that the *part* was extracted so to have similar characteristics of the whole. Therefore, a multivariate model was chosen among the different possible regression models considered. The model was built computing mean values of selected set of metrics on all the programs composing each WP. [DDS02]

After the values in the model were calculated the researchers' examined the Mean Magnitude Relative Error (MMRE) rate for each examined WPs (14 in this case). The average prediction error occurred was about 35 %. According to researchers considering 10 % of the programs, the average MMRE was 47,5 %, wasn't excellent for effort prediction, but not so bad considering errors occurred (60 %) in a study "Assessing Massive Maintenance Processes: an Empirical Study" by De Lucia, Pannella, Pompella and Stefanucci [DPP01]. According to my opinion it doesn't make your results any better, if some other studies have got even worse results.

The researchers we also satisfied with their effort prediction model by claiming that the model works well even if the sample extracted is small, given that:

- A sufficiently larger data set (coming from similar projects, or from previously maintained WPs) were used to build the model; and

[1] Note that the LOC used here is not defined according to LOC definition in IEEE standard [IEE93].

- The sample were uniformly extracted, so that, as said the *part* exhibits similar characteristics of the *whole*. [DDS02]

According to my opinion especially the second bullet limits the usage of the model quite much. In practice there might not be so many programs that can be divided into parts that represent uniformly the whole program. That is because nowadays the programs are quite complex and they are divided to many subsystems that might use different programming languages and different kind of application interfaces to other programs.

The researchers conclude the empirical study by saying that the metrics used in the model are early and easily available at the beginning of the operative phase of the maintenance project through the aids of code analysis tools. They also claim that the model they proposed can provide an effort estimate very early in the process, allowing the managers to make reasonably the first staffing, cost, and resources allocation decisions without the need of analyzing the entire software system. [DDS02]

4 Discussion

As mentioned earlier in the Chapter 2 software should be designed for maintenance [AlC98]. Metric data provides quick feedback for software designers and managers already in design and implementation phase of the project. By analysing the collected data, complexity and design quality as well as some other properties of the final software can be predicted. If appropriately used, it can lead to a significant reduction in costs of the overall implementation and improvements in quality of the final product. The improved quality, in turn reduces future maintenance efforts. However, information available in the early design phase is often inaccurate and insufficient. In fact, many useful metrics cannot be used during the early design phase. Because of this, information of software design presented by metric data needs to be revisited and acquired in various phases of the software life cycle. [YSM02]

There is a definite need for further work in this (cost modeling of maintenance) field. Costs, which occur after a system has gone into production, need to become predictable. If not, user organizations will never be able to control their costs [Sne04]. Maintenance complexity (and costs) will increase exponentially as the number of independent COTS (commercial off-the-shelf product) packages integrated into a system increases [RBB03].

A large system will have different complexities at different levels – at component level, at the subsystem level and at the system level. The maintenance operation may lead to an increase in the number of components and subsystems, but it should not lead to an increase in their interactions. They should remain at least at the same level, if not decrease as a result of reengineering efforts. This applies only to static systems. Dynamic systems keep growing until they reach some upper bound of complexity or until they have fulfilled all the requirements of all users. [SnB03]

It is unlikely that universally valid program structure metrics and models could be devised, so that they would suit for all languages in all development environments and for different kind of application domains. But many researches studied while writing this paper were lacking of ability to generalize the results even for particular language or for particular environment or for small and large programs [GiS89, BBM96, ErS98, DPP01, TjL01, DPP02, DDS02, YSM02, BVT03, SnB03]. Instead, some of the researches specifically concluded that because the research was done only in one or couple of (real or testing) environment the results might not be applicable to any other situation. Making a critical conclusion of the above: every organization should have their own metrics for measuring the structure of the programs for easing up the maintenance work.

Sneed is also concern about lack of real empirical results and research in the area of software maintenance. According to his opinion the research community must come to grips with life cycle cost management and provide well founded models based on empirical studies of real world maintenance operations. Only by building up a statistical database will that be possible. Therefore, the first step is to establish corporate or even national metric databases from which researchers can obtain the data they need to create their prediction models. To this end there must be a joint venture on the part of business and government to support such research. [Sne04]

5 Conclusions

This paper introduces the role of program structure in software maintenance. Program structure is the way the source code is designed and developed in the software system. Unfortunately, we do not have reliable metrics to accurately measure the maintainability of a software system. Complexity metrics, however, can give some clues as to components, which may incur high maintenance costs [HTO94]. The more complex a system is the more difficult it is to understand, and therefore to maintain [GiS89].

One empirical study is introduced in more detailed [DDS02] in Chapter 2 and some other studies are handled only briefly. In general empirical researches study the maintenance problem only in one point of view, for example by studying programs that are still in development phase or by studying only programs that are old legacy systems. Some other viewpoints are related to program size and programming languages.

Discussion chapter tries to cover some issues that are generally discussed in scientific articles related to role of program structure in software maintenance. As a conclusion there is a concern about lack of real empirical results and research in the area of software maintenance [Sne04].

References

[AlC98] Alkadi Ghassan, Carver Doris L.: Application of Metrics to Object-Oriented Designs, Proceedings of IEEE Aerospace Conference, Volume 4, pages 159 - 163, March 1998.

[BBM96] Victor R. Basili, Fellow, IEEE, Lionel C. Briand, Walcélio L. Melo, Member IEEE Computer Society: A Validation of Object-Oriented Design Metrics as Quality Indicators, IEEE Transactions on Software Engineering, Volume 22, Number 10, pages 751 - 761, October 1996.

[BMB99] Lionel C. Briand, Sandro Morasca, Member, IEEE Computer Society, Victor R. Basili, Fellow, IEEE: Defining and Validating Measures for Object-Based High-Level Design, Volume 25, Number 5, pages 722 - 743, September - October 1999.

[BVT03] Rajendra K. Bandi, Vijay K. Vaishnavi, Fellow, IEEE, Daniel E. Turk, Member, IEEE: Predicting Maintenance Performance Using Object-Oriented Design Complexity Metrics, IEEE Transactions on Software Engineering, Volume 29, Number 1, pages 77 - 87, January 2003.

[DDS02] Andrea De Lucia, Massimiliano Di Penta, Silvio Sttefanucci, Gabriele Venturi: Early Effort Estimation of Massive Maintenance Processes, IEEE Proceedings of the International Conference on Software Maintenance (ICSM'02), pages 234 - 237, October 2002.

[DoL97] Douce C., Layzell P. J.: Maintenance of Object-Oriented C++ Software: A Protocol Study, IEEE Computer Society 2nd International Workshop on Empirical Studies of Software Maintenance (WESS'97), pages 115 - 119, October 1997.

[DPP01] Andrea De Lucia, Antonello Pannella, Eugenio Pompella, Silvio Stefanucci: Assessing Massive Maintenance Process: an Empirical Study, IEEE Proceedings of International Conference on Software Maintenance, IEEE CS Press, pages 451 - 458, November 2001.

[DPP02] Andrea De Lucia, Antonello Pannella, Eugenio Pompella, Silvio Stefanucci: Empirical Analysis of Massive Maintenance Processes, IEEE Proceedings of the 6th European conference on Software Maintenance and Reengineering (CSMR'02), pages 5 - 14, March 2002.

[ErS98] Erdös Katalin, Sneed Harry M.: Partial Comprehension of Complex Programs (enough to perform maintenance), Proceeding of the 6th International Workshop on Program Comprehension (IWPC'98), pages 98 -105, June 1998.

[GiS89] Virginia R. Gibson, James A. Senn: System Structure and Software Maintenance Performance, Communications of ACM, Volume 32, Number 3, pages 347 - 358, March 1989.

[Gre84] Lee L. Gremillion: Determinants of program repair maintenance requirements, Communications of ACM, Volume27, Number 8, pages, 826 - 832, August 1984.

[Hal77] Halstead, Maurice H.: Elements of Software Science, Operating, and Programming Systems Series, 127 pages, 1977.

[HTO94] Toyohiko Hirota, Masaharu Tohki, Michael Overstreet, Masaaki Hashimoto, Robert Cherinka: An approach to predict software maintenance cost based on ripple complexity, IEEE Proceedings on Software Engineering Conference, pages 439 - 444, December 1994.

[IEE93] IEEE Standard for Software Productivity Metrics, Software Engineering Standards Subcommittee of the Technical Committee on Software Engineering of the IEEE Computer Society, USA, 1993.

[LIK00] Shuqin Li-Kokko: Code and Design Metrics for Object-Oriented Systems, Helsinki University of Technology, 9 pages, 2000.

[LoK94] Lorenz Mark, Kidd Jeff: Object-Oriented Software Metrics: A Practical Guide. P T R Prentice Hall, Prentice-Hall, Inc. A Pearson Education Company, 146 pages, 1994.

[McC76] McCabe Thomas J.: A Complexity Measure, IEEE Transactions on Software Engineering, Volume SE-2, Number 2, pages 308 - 320, September 1976.

[MMC99] Mancoridis S., Mitchell B. S., Chen Y., Gansner E.R.: Bunch: A Clustering Tool for the Recovery and Maintenance of Software System Structures, IEEE Proceedings of the International Conference on Software Maintenance (ICSM'99), pages 50 - 59, August - September 1999.

[RBB03] Reifer Donald J., Basili Victor R., Boehm Barry W., Clark Betsy: Eight Lessons Learned during COTS-Based Systems Maintenance, IEEE Software, Volume 20, Issue 5, pages 94 - 96, September - October 2003.

[SnB03] Sneed Harry M., Brössler Peter: Critical Success Factors in Software Maintenance A Case Study, IEEE Proceedings of the International Conference on Software Maintenance (ICSM'03), pages 190 - 198, September 2003.

[Sne04] Sneed Harry M.: A Cost Model for Software Maintenance & Evolution, IEEE Proceedings of the 20th IEEE International Conference on Software Maintenance (ICSM'04), pages 264 - 274, September 2004.

[SyY99] Tarja Systä, Ping Yu: Using OO Metrics and Rigi to Evaluate Java software, University of Tampere, Department of Computer Science, Series of Publications A A-1999-9, 24 pages, July 1999.

[TjL01] Tjortjis Christos, Layzell Paul: Expert Maintainers' Strategies and Needs When Understanding Software: A Case Study Approach, IEEE 8th Asia-Pacific Conference on Software Engineering, pages 281 - 287, December 2001.

[VeW83] Iris Vessey, Ron Weber: Some factors affecting program repair maintenance: An empirical study, Communication of ACM, Volume 26, Number 2, pages 128 - 134, February 1983.

[WEY88] Weyuker Elaine J.: Evaluating Software Complexity Measures, IEEE Transactions on Software Engineering, Volume 14, Number 9, pages 1357 - 1365, September 1988.

[YSM02] Ping Yu, Tarja Systä, Hausi Müller: Predicting Fault-Proneness using OO Metrics An Industrial Case Study, Proceedings of the 6th European conference on Software Maintenance and Reengineering (CSMR´02), pages 99 - 107, March 2002.