Software Metrics: Structure and Some new Research Results

Wohlin, C., Lennselius, B. and Vrana, C.

Proceedings Milcomp, pp. 221-226,
London, United Kingdom, 1986.

SOFTWARE METRICS - STRUCTURE AND SOME NEW RESEARCH RESULTS

Claes Wohlin*, Bo Lennselius* and Ctirad Vrana**

ABSTRACT

Software properties and the effects of introducing new methods and tools have to be measured. We need to manage all aspects of software (e.g. structure, resource allocation, qualities) in an optimal way. This is not possible without the aid of software metrics. This paper will discuss the need of software metrics, classify qualities, and present some explicit results from research into complexity, reliability and correctness.

1. INTRODUCTION

A condition for all industrial activities is that the achieved results can be measured. The fact is that what you cannot measure, you cannot control either. Management by objectives, Figure 1, requires that the following three conditions be met:

1. There have to be measures on qualities.
2. The relationships between these and other quantities (e.g. resources, planning, economy) have to be known.

These two conditions call for formal methods during specification, design, coding, testing etc.

3. We have to have a feedback (organizational) to be able to use the results, in order to plan and control.

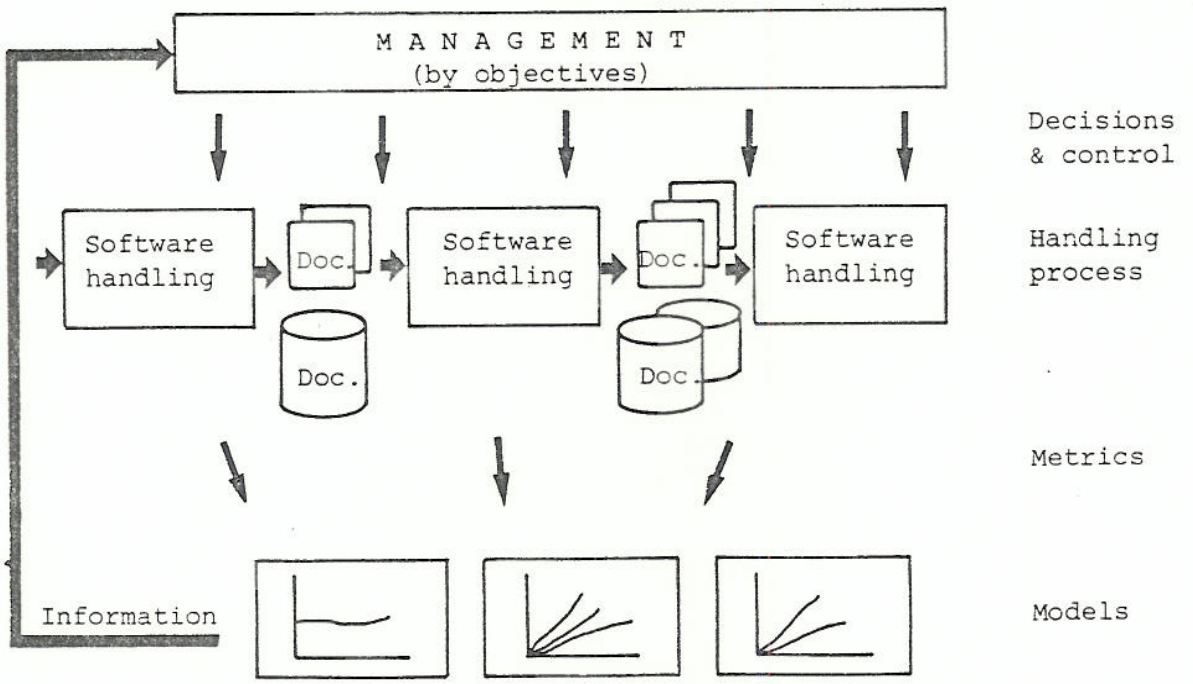

Figure 1. Management by objectives.

* Lund Institute of Technology, Lund, Sweden.
** Telelogic AB, Malmö, Sweden.

Metrics are both of a qualitative and a quantitative nature. They can be related to qualities of the product, the handling of the product and, consequently, they can be related indirectly to methods, tools and actors (that is organizations and humans who participate in the handling process).

Metrics are the theoretical basis of the whole field of Software Engineering. The list of areas and occasions where different metrics are needed is very long. Some typical examples are; contracts/responsibilities, requirement specifications, guarantees, planning and cost estimation of projects, control of projects, the choice of methods, the choice of basic techniques, quality assurance, tests, and the regaining of experiences.

2. THE STRUCTURE

The basic strategy is to find a structure among the different qualities and to see how this structure harmonizes with other models and ways of looking at things connected to the concept "software products".

The structure presented below harmonizes with the product concept applied to software products within the organization of the Swedish Telecommunication Administration (and in other companies as well), and imitates the well-known way of handling mechanical products.

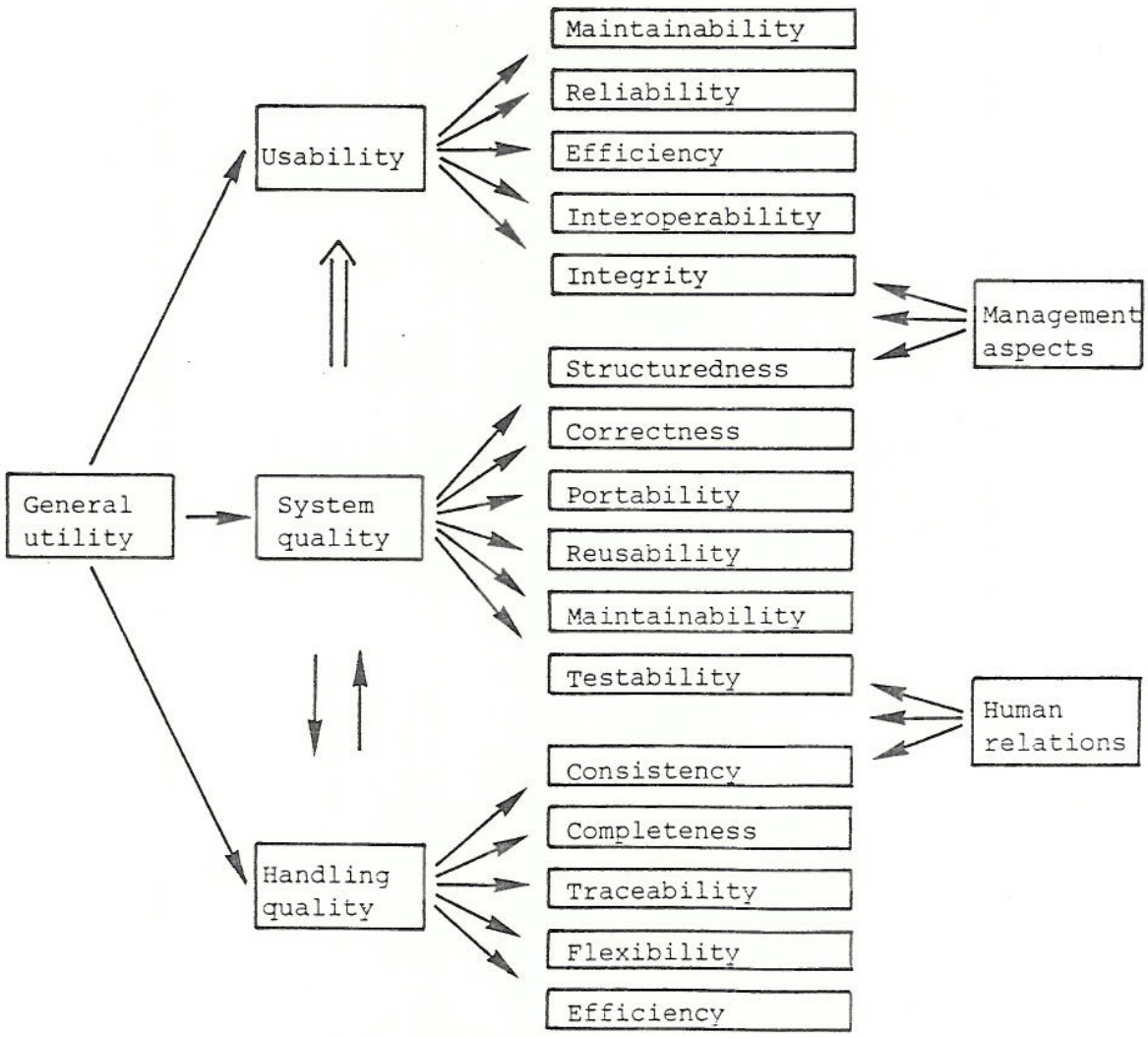

Figure 2. A criteria structure.

In accordance with the system and lifecycle models for a system, [1], a system is defined by its documentation (through definition the system does not exist if there is no adequate documentation). All handling of a system is defined as a transformation of these documents. From the abstract system (the source system) the operative systems are derived (copied, constructed, manufactured).

The different economical and quality aspects can (and should) be structured in a similar manner. Even if the structure is not completely orthogonal it makes the derivation and studies of metrics and their applications considerably easier. A structure of qualities is presented in Figure 2. A study presented in [2] shows how qualities are impacted in a system built in a so-called hierarchical modular structure. In systems built in this way almost all qualities are improved.

3. SOME SPECIFIC METRICS

As pointed out, there are many qualities that can be of interest. Which ones are to be considered important depends on the application and the constraints on that specific application. The area is not completely investigated.

Yet some metrics have, traditionally and due to their importance, been used for a period of time. They have also been studied more theoretically than others. To this category belongs metrics, as denoted above, such as reliability, correctness, complexity (and different qualities derived from it) and the relations between them. This paper will concentrate on these specific metrics.

3.1 Complexity metrics

Introduction

Complexity metrics can be used to measure how complex (or difficult) a software product is. Historically the complexity metrics have been based on the source code. But if complexity metrics are to be used as a tool for management, it is necessary to measure before the coding phase. This can be done if the product is documented with a well-defined and standardized description technique during the different phases of the software lifecycle. An example of a well-defined description language is SDL (Specification and Description Language, standardized by CCITT, [3]).

Description complexity (C_d) is a measure of how complex a description (document) is and the complexity can be divided into different parts, i.e. complexity originating from size (C_s), control structure (C_{cs}), interdependencies of different descriptions (modules of a program) (C_{co}) and so on. This gives us the following relationship

$$C_d = f(C_s, C_{cs}, C_{co}, \ldots) \qquad (1)$$

There is no metric which covers all these aspects of description complexity for a software product today. Because of this it is very important to use different types of complexity metrics, that is, to use different metrics to measure different aspects of the complexity.

Together with other factors, complexity affects the human handling of the descriptions (the product) throughout the software lifecycle. Assuming that the influence from other factors is equal for all modules within the same product, the following relationship is obtained between the number of errors in the product (ERROR) and the complexity

$$\text{ERROR} \approx k_1 C_s^{z_1} + k_2 C_{cs}^{z_2} + k_3 C_{co}^{z_3} + \ldots \qquad (2)$$

where k_j and z_j are constants which depend on the description language, programming language, type of product, methodology and tools.

Another very important application of the complexity analysis is to identify error-prone modules in a product, in order to make a cost-effective allocation of quality assurance resources, test resources, etc.

Some results

A study of three similar, large, real-time systems was conducted [4]. In total, 63 modules (size: 500-7000 lines of code) were examined in order to study the dependencies between the number of errors and the complexity. The modules are described with a SDL-like language. In the investigation, metrics from these descriptions were used to measure, for instance, size, control structure, and dependencies between different modules. The SDL-based metrics were compared with the often-used measure, "Lines of code".

By using regression analysis the dependencies (correlation) between the number of errors and each metric were studied. The result was that the SDL-based metrics had a higher or at least the same correlation with the number of errors as the metric "Lines of code". This indicates, for example, that the SDL-based metrics can be used early in the lifecycle in order to estimate the number of errors and to identify the error-prone modules in the product.

By defining error outliers (error-prone modules) and metric outliers (complexity-prone modules) we were able to study how well the metric outliers pointed out the error outliers within each product. Out of 63 modules investigated, 11 were error-prone. The metrics identified between 6 and 10 of these modules (i.e. a metric outlier is also an error outlier). Between 2 and 4 modules were indicated as error outliers, but were not (i.e. a metric outlier which was not an error outlier). The SDL-based metrics were better than "Lines of code" and the best metric was the measure of dependencies between the module studied and the other modules (10 error outliers were identified and only 2 were pointed out to be error outliers though they were not).

This investigation shows that it is possible to identify error-prone modules early in the lifecycle of the software product and to take counter-measures before the fact is faced (i.e. a coded product). An important property of a well-defined description language, for instance SDL, is also shown, and that is the possibility to measure, estimate and control before the coding phase.

3.2 Reliability

Introduction

One of the most important quality aspects is the reliability of the software. Software reliability can be defined as the probability that the software does what it is supposed to do during a prespecified time. The reliability of the software is strongly dependent on the number of errors that are introduced when developing the program, since the software does not wear out the same way as the hardware. This is not, however, the whole truth, since the software is often changed, improved, and new features are added throughout the lifetime of the software.

In order to get a good picture of the reliability, factors such as the remaining number of errors and time between failures (that is to predict how failures will occur in the future) have to be determined. An early estimate of the number of errors can be obtained through complexity metrics, see above. This estimate can be used as input to software reliability models, in order to get better estimates of the reliability factors mentioned above as the project proceeds. The results from the models are improved as time goes by through collection of failure data. This will make it possible to refine the estimates of the parameters of the models. By predicting the failure occurrences it is possible, for example, to predict a suitable time to release the software product and to determine the allocation of resources, in order to deliver on the target date a product which meets the quality constraints.

The availability of software reliability models

A realistic prediction of the software failure occurrences is, of course, a must to be able to draw any conclusions from it. It is therefore necessary that software reliability models well suited for the situation (i.e. the environment and the application) are used. The models have to be reasonable according to, for example, development environment, tools, application, test environment, etc. A thorough investigation, classification and comparison of existing models as well as a study of our own environment is needed in order to identify the models which are possible to use for our products. A comparison of two models is presented in [5].

The results of the investigation can be either that some models are applicable or that no model is suitable for the environment and the application studied. It is of great importance not to take a model and start using it without a thorough study of the assumptions and the application areas of the model. In the cases where no model is applicable it is necessary to develop a model of our own or accept that it is impossible to say anything about the future failure occurrences. To accept the latter is a big step towards losing control of the software project.

In [6] an example is presented of how an investigation of a specific test environment can lead to the development of a software reliability model which is adapted to the environment. The investigation also leads to the conclusion that it is probably not possible to find a global model, that is, different models are needed during the different phases of the software lifecycle. The models must be adapted to each phase and when a transition is made to a new activity the need for a new model has to be accepted. Most of the existing software reliability models are adapted to the operational phase or similar conditions, which in most cases are not valid during the testing of a system.

The importance of developing models which are applicable during testing can not be stressed enough, since testing is one of our best weapons against software errors. Testing will give us a more reliable product. If software reliability models which are applicable during testing were available it would be possible to determine the reliability (or correctness) at different times, for instance, the time of release. An idea of how a quality constraint can be put on the product in order to release the product at the economically optimal time is presented in [7]. It is possible to apply the idea if a model for the lifecycle costs is developed and if reasonable software reliability models are available. The latter is available at present, but more research is needed into the former, and this calls for cooperation between business economists and technicians.

4. CONCLUSION

Further study will concentrate on the two areas presented here and on studies of other qualities. An area which will be looked into in the near future is problems concerning efficiency during run-time and capacity aspects.

The results from the different studies have been supplied to those responsible for methods, construction of tools, and quality assurance, in order to apply the results in operative activities. The different relations between structuring and complexity will be used within programming support environments which are developed at Telelogic AB for SDL and Ada. A prototype of a tool for following-up the reliability of the products has been developed and the operative application will be further studied. In the long run other qualities will be measured and controlled in the integrated handling environment.

In this way, we believe the possibilities for managing the problems of developing software products will be essentially improved in the future.

5. ACKNOWLEDGEMENT

This project is supported by Telelogic AB and the Swedish Telecommunication Administration, Sweden.

6. REFERENCES

1. Rapp, D., and Vrana, C., "Systems and System Management Environment", NT-symposium, Turku, Finland, 1984.
2. Vrana, C., "S/W Engineering Economics - Models for Systems with a Hierarchical Modular Structure", NT-symposium, Turku, Finland, 1984.
3. CCITT, "Recommendations Z101-104, Yellow Book", Volume VI, Fascicle VI.6, Geneva, Switzerland, 1981.
4. Lennselius, B., "Software Complexity and Its Impact on Different Software Handling Processes", Proc. IEE, 259, pp 148-153, 1986.
5. Wohlin, C., "A Comparison Between Two Software Reliability Models", Technical Report, Lund Institute of Technology, Lund, Sweden, 1986.
6. Wohlin, C., "Software Testing and Reliability for Telecommunication Systems", Proc. Software Engineering '86, Southampton, England, 1986.
7. Wohlin, C., and Vrana, C., "A Quality Constraint Model to be Used During the Test Phase of the Software Lifecycle", Proc. IEE, 259, pp 136-141, 1986.