ACHIEVING SOFTWARE QUALITY USING THE NFR FRAMEWORK: MAINTAINABILITY AND PERFORMANCE

Bill Andreopoulos
Department of Computer Science
York University
Toronto, Ontario, M3J 1P3, Canada
billa@cs.yorku.ca

Abstract: We propose a general methodology for evaluating and improving the quality of a software system. To illustrate how the methodology works, our work focuses on the software qualities of maintainability and performance. The Non-Functional Requirements (NFR) framework is adopted to represent and analyze the software qualities of maintainability and performance. Specifically, we analyze the software attributes that affect either quality, the heuristics that can be implemented in source code to achieve either quality and how the two qualities conflict with each other. Experimental results are discussed to determine the effect of various heuristics on maintainability and performance. A methodology is described for selecting the heuristics that will improve a system's software quality the most. The results of our research were encoded in XML files, and made available on the World Wide Web (WWW) for use by software developers. The URL is: http://www.cs.yorku.ca/~billa/SIG/SIG.xml

Keywords: Maintainability, Performance, Software, Quality, NFR Framewok.

1 INTRODUCTION

As organizations rely increasingly upon software to perform their vital functions, building software of high quality becomes a critical issue. The quality of software will affect the delivery of services, the overall performance of the system and the long-term cost of software maintenance. Thus, it is necessary to be able to represent and analyze software quality effectively, throughout the entire software lifecycle.

1.1 Objectives and Methodology

We propose a general methodology for evaluating and improving the *maintainability* and *performance* qualities of a software system. An important part of our work examined how the two qualities conflict with each other. We adopt the NFR Framework to represent the qualities of maintainability and performance, the software characteristics that affect them and the heuristic transformations (or *heuristics*) that can be implemented in source code to achieve them. The purpose of this website is to provide a tool that can be used by software developers for optimizing a system's source code. This website can be used to select the set of heuristics that will benefit the system's maintainability and/or performance the most, while minimizing the negative side effects. This website also gives precise definitions for the terms used in this work, as well as the full versions of the softgoal interdependency graphs shown in the figures.

1.2 Software Qualities and the NFR framework

In requirements engineering, a requirement can be described as a condition or capability to which a system must conform, and which is either derived directly from user needs, or stated in a contract, standard, specification, or other formally imposed document. Non-functional requirements (or *software qualities*) are constraints on the design and/or implementation of the solution. *Software qualities* describe not *what* the software will do, but *how* the software will do it, by specifying constraints on the system design and/or implementation. Unfortunately, software qualities are usually specified briefly and vaguely for a particular system, since no exact techniques for representing them have been standardized yet by the software engineering community.

The *NFR framework* for representing software qualities was developed by Lawrence Chung, Brian Nixon, Eric Yu and John Mylopoulos at the University of Toronto. [1] The NFR framework represents quality requirements as *softgoals*. A softgoal may contribute positively or negatively towards achieving another softgoal. A softgoal can be *satisficed* or not: satisficing refers to satisfying a goal to some level, but without necessarily producing the optimal solution. [1] A *softgoal interdependency graph* records all softgoals being

considered, as well as the interdependencies between them. [1] An example of a softgoal interdependency graph is given in Figure 1.
A developer can start constructing a *softgoal interdependency graph* by identifying the top-level quality requirement that the system is expected to meet and respresent it as a node. The softgoal interdependency graph provides a hierarchical decomposition of the softgoals; more general *parent softgoals* are shown above more specific *offspring softgoals*. In Figure 1 the general *high maintainability* softgoal gets decomposed into the more specific *high source code quality* and *high documentation quality* softgoals. In some cases the interdependency links in a decomposition are grouped together with an arc; this is referred to as an *AND* contribution of the offspring softgoals towards their parent softgoal, and means that all offspring softgoals must be satisficed to satisfice the parent. Figure 1 shows that both softgoals for *high source code quality* and *high documentation quality* must be satisficed to satisfice the *high maintainability* softgoal. In other cases the interdependency links are grouped together with a double arc; this is referred to as an *OR* contribution of the offspring softgoals towards their parent softgoal and means that only one offspring softgoal needs to be satisficed to satisfice the parent. Figure 1 shows that either *low span of data* or *high data consistency* can satisfice the *high information structure quality* softgoal.

The bottom of the graph consists of the *heuristic transformations* (also called *heuristics* or *operationalizing softgoals*) that can be implemented in the system, to achieve one or more parent softgoals. Figure 1 illustrates the *dead code elimination*, *elimination of GOTO statements* and *elimination of global data types and data structures* heuristics. Like other softgoals, heuristics also make a contribution towards one or more parent softgoals. In this case, a heuristic's contribution towards satisficing a parent softgoal can be positive ("+" or "++") or negative ("-" or "--") [1] and this contribution is indicated next to the interdependency link.

2 MAINTAINABILITY AND PERFORMANCE

We used the NFR framework to analyze the maintainability and performance qualities as described next.

2.1 Decomposing Maintainability into Softgoals

Maintainability is defined as the characteristics of the software that affect the maintenance process and are indicative of the amount of effort necessary to perform maintenance changes. It can be measured as the time necessary to make maintenance changes to the product. [2, 5] To effectively deal with such a broad quality, we treat it as a *softgoal* (see Section 1.2) and then decompose it down into more specific softgoals. Figure 1 shows the *softgoal interdependency graph* for maintainability. For the full graph see our website. In cases where there exist conflicting views of how attributes affect the maintainability of software, these cases are noted throughout our descriptions.

The maintainability quality can be decomposed into the softgoals high source code quality [2], and high documentation quality [3]. This decomposition is shown in Figure 1. Both softgoals of high source code and documentation quality must be satisficed for a system to have high maintainability. This is referred to as an *AND* contribution of the offspring softgoals towards their parent softgoal, and is shown by grouping the interdependency lines with an arc. The rationale behind this *AND* contribution is that a software system with clear source code but bad documentation will be hard to maintain, since maintainers will need to study requirements and design documents in order to understand how the system works. A software system with clear documentation but badly written code will also be hard to maintain, since maintainers will need to understand how the source code works in order to make changes to it. Thus, developers must try to satisfice both softgoals in a system.

The *high source code quality* softgoal can be further decomposed into the sub-softgoals high control structure quality [2], high information structure quality [2], and high code typography, naming and commenting quality [6, 7]. This decomposition is shown in Figure 1. As shown, this is also an *AND* contribution, i.e. all three sub-softgoals must be satisficed to achieve the *high source code quality* softgoal. The rationale behind this *AND* contribution is that the source code will be hard to understand if it is badly commented or if it is laid out in a bad manner (typography qualities). But source code will also be hard to understand if characteristics such as modularity, encapsulation or cohesion have not been achieved (control and information structure qualities).

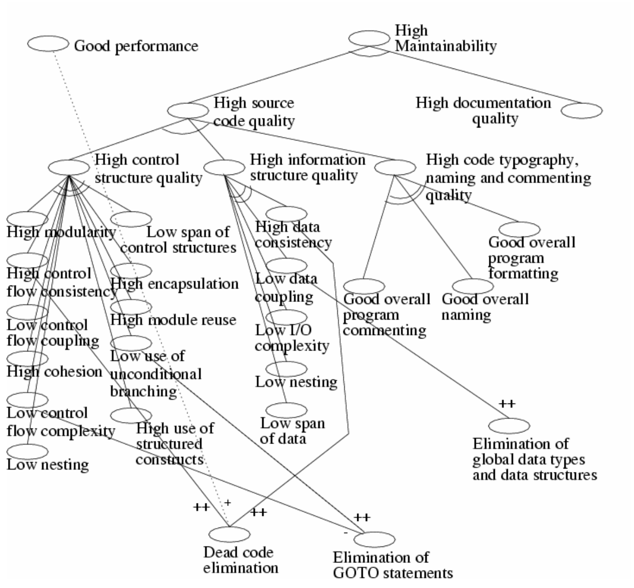
Good performance
High Maintainability
High source code quality
High documentation quality
High control structure quality
High information structure quality
High code typography, naming and commenting quality
High modularity
Low span of control structures
High control flow consistency
High encapsulation
Low control flow coupling
High module reuse
High cohesion
Low use of unconditional branching
Low control flow complexity
High use of structured constructs
Low nesting
High data consistency
Low data coupling
Low I/O complexity
Low nesting
Low span of data
Good overall program formatting
Good overall program commenting
Good overall naming
Elimination of global data types and data structures
Dead code elimination
Elimination of GOTO statements

Figure 1 - Maintainability softgoal interdependency graph, including a subset of the heuristics. The full set of heuristics can be found on our website.

2.2 Decomposing Performance into Softgoals

Like maintainability, we also view performance as a *softgoal* (see Section 1.2) that can be broken down into more specific softgoals. Figure 2 shows the *softgoal interdependency graph* for performance. For the full graph see our website.

The *high performance* quality can be decomposed into the softgoals good time performance [4], and good space performance [4]. This decomposition is shown in Figure 2. As shown, this is an *AND* contribution, i.e. both softgoals must be satisficed to achieve the *performance* softgoal. The rationale behind this *AND* contribution is that it is inconceivable for a system that is fast but makes bad memory-utilization to be characterized by good performance. It is also inconceivable for a system that makes good memory-utilization but is slow, to be characterized by good performance.

The *good space performance* softgoal can be decomposed into the sub-softgoals low main memory utilization, and low secondary storage utilization. This decomposition is shown in Figure 2. As shown, this is also an *AND* contribution, i.e. both sub-softgoals must be satisficed to achieve the *good space performance* softgoal. The rationale behind this *AND* contribution is that the system may be stored either in main memory or in secondary storage and ‘space’ refers to both.

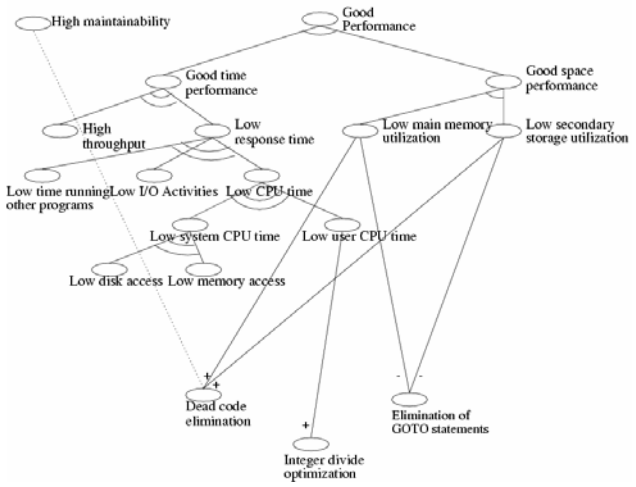

Figure 2 - Performance softgoal interdependency graph, including a subset of the heuristics. The full set of heuristics can be found on our website.

2.3 Identifying Heuristic Transformations to Achieve Software Quality

Up to now we have been providing precise definitions for the qualities of maintainability and performance. However, we have not yet described the *heuristics* (see Section 1.2) by which one could satisfice the quality requirements of high maintainability and performance in a system. The NFR framework treats these heuristics as *softgoals* (see Section 1.2), because this allows developers to decompose heuristics into more specific ones. Like other softgoals, heuristics also make a contribution towards one or more parent softgoals, but in this case the contribution types are positive/negative. This is represented with a *"+", "++", or "-", "--"* symbol. [1]

2.3.1 Identifying Heuristics to Satisfice Maintainability

We briefly describe some of the *heuristics* that can be implemented in a system's source code to contribute towards satisficing the maintainability quality requirement. Our website provides a full description of all the maintainability heuristics as well as their contributions, and should be consulted for further details.

The softgoal interdependency graph, given in Figure 1, illustrates a subset of these heuristics as well as their contributions towards their parent softgoals. As shown in Figure 1, an example of a maintainability heuristic is *elimination of GOTO* *statements*. This means to minimize the number of GOTO statements in the source code. Implementing this heuristic makes a "++" contribution towards meeting the *low use of unconditional branching* softgoal. Implementing this heuristic also makes a "-" contribution towards meeting the *low control flow complexity* softgoal. Elimination of GOTO statements may also affect performance in various ways. We discuss these contributions in the next section.

2.3.2 Identifying Heuristics to Satisfice Performance

We briefly describe some of the *heuristics* that can be implemented in a system's source code to satisfice the performance quality requirement. Our website provides a full description of all the performance heuristics as well as their contributions and should be consulted for further details.

The softgoal interdependency graph, given in Figure 2, illustrates a subset of these heuristics as well as their contributions towards their parent softgoals. As shown in Figure 2, an example of a performance heuristic is *elimination of GOTO statements*. This means to minimize the number of GOTO statements in the source code. Implementing this heuristic makes a "-" contribution towards meeting the *low main memory utilization* and *low secondary storage utilization* softgoals. Elimination of GOTO statements may also affect maintainability in various ways, as discussed in the previous section.

3 MAINTAINABILITY AND PERFORMANCE MEASUREMENTS

We performed maintainability and performance optimization activities, by implementing different heuristics at the source code level. Each optimization activity we performed corresponds directly to a specific heuristic that is described in our website. We evaluated the effect of applying each optimization heuristic on the overall maintainability and performance of the source code. For each optimization activity, a set of maintainability metrics models and performance measures were applied to the source code, before and after the optimization activity took place.

We chose two software systems - WELTAB, an election tabulation system, and the AVL GNU tree and linked list libraries - that were originally written in C, to be reengineered in object-oriented C++ by using an automated tool that resulted in very low C++ code quality. This low quality motivated us to try to improve the source code by implementing optimization heuristics.

3.1 Maintainability Metrics Models

In order for maintenance processes to be improved and for the amount of effort expended in software maintenance activities to be reduced, it is first necessary to be able to measure software maintainability. [8] In this section, we describe the most important maintainability metrics that were extracted from the WELTAB and AVL C++ source code to evaluate the effects of optimizations. In the metrics defined below, the term *avg-E* is the average Halstead Volume V per module; *avg-V(G)* is the average extended cyclomatic complexity per module; *avg-LOC* is the average count of lines of code (LOC) per module; *avg-CMT* is the average percent of lines of comments per module

MI1 - This is a single maintainability index, based on Halstead's metrics. It is computed as follows:
*MI1 = 125 - 10 * LOG(avg-E)*
MI2 - This is a single maintainability index, based on Halstead's metrics, McCabe's Cyclomatic Complexity, lines of code and number of comments. It is computed as follows:
*MI2 = 171 - 5.44 * ln(avg-E) - 0.23 * avg-V(G) - 16.2 * ln(avg-LOC) + 50 * sin(sqrt(2.46 * (avg-CMT / avg-LOC)*
MI3 - This is a single maintainability index, based on Halstead's metrics, McCabe's Cyclomatic Complexity, lines of code and number of comments. It is computed as follows:
*MI3 = 171 -3.42 * ln(avg-E) - 0.23 * avg-V(G) - 16.2 * ln(avg-LOC) + 0.99 * avg-CMT*

3.2 A study of the optimization activities

The pre-post analysis of the maintainability and performance metrics was performed on nine different code optimization heuristics. Four of these heuristics focused on improving performance and the other five focused on improving maintainability. Following is a brief description of the performance and maintainability optimization heuristics:

Hoisting and Unswitching - The FOR loops were optimized, so that each iteration executed faster (performance optimization).

Address Optimization - References to global variables that used a constant address were replaced with references using a pointer and offset (performance optimization).

Integer Divide Optimization - Integer divide instructions with power-of-two denominators were replaced with shift instructions, which are faster (performance optimization).

Function Inlining - When a function was called in the program, the body of the function was expanded inline (performance optimization).

Elimination of GOTO statements - The number of GOTO statements in the source code was minimized (maintainability optimization).

Dead Code Elimination - Code that was unreachable or that did not affect the program was eliminated (maintainability optimization).

Elimination of Global Data Types and Data Structures - Global data types and data structures were made local (maintainability optimization).

Maximization of Cohesion - Classes with low cohesion were split into many smaller classes, when possible (maintainability optimization).

Minimization of Coupling Through ADTs - Variables declared within a class, which have a type of ADT that is another class definition, were eliminated (maintainability optimization).

We extracted maintainability and performance metrics on the WELTAB and/or AVL source code before and after the optimization activities took place. Results are described next for each optimization. Due to space limitations, we only describe results for 3 optimizations.

3.2.1 Function Inlining

This heuristic was implemented in both WELTAB and AVL. The maintainability measurements taken on WELTAB are shown in the Table below. All MIs decreased after this heuristic was applied. These decreases can be attributed to the fact that all Halstead’s metrics and lines of code increased. So this heuristic affected maintainability negatively.

Metric	Pre-value	Post-value
M1	71.9263	71.4982
M2	36.6910	35.5612
M3	61.3768	60.4460

The performance measurements taken on WELTAB are shown in the Table below. As we can see, performance was improved.

WELTAB Function	Pre-performance	Post-performance
Weltab-poll	0.81	0.42
Weltab-spol	0.32	0.23

3.2.2 Maximization of Cohesion

This heuristic was implemented in AVL only. The maintainability measurements taken are shown in the Table below. All measurements show an increase in maintainability after maximizing cohesion.

Metric	Pre-value	Post-value
M1	70.43	76.32
M2	36.22	55.32
M3	61.43	73.67

The performance measurements taken are shown in the Table below. As we can see, performance was affected negatively after applying this heuristic.

AVL Function	Pre-performance	Post-performance
SampleRec	0.67	0.69

3.2.3 Minimization of Coupling Through ADTs

This heuristic was implemented in AVL and measurements were taken on specific functions. The maintainability measurements taken are shown in the Table below. All measurements show an increase in maintainability after this optimization.

Metric	Pre-value	Post-value
M1	76.86	79.31
M2	98.77	102.67
M3	108.44	111.45

The performance measurements taken are shown in the Table below. As we can see, this heuristic affected performance negatively.

AVL Function	Pre-performance	Post-performance
Ubi_cacheRoot	0.67	0.68
Ubi_idbDB	0.56	0.58

4 SELECTING THE HEURISTIC TRANSFORMATIONS TO BE IMPLEMENTED

The set of heuristics selected for implementation must be the ones that will benefit system maintainability and performance the most, by maximizing the ratio of gains to losses. An *evaluation procedure* can be used to determine the degree to which each top-level quality requirement (maintainability or performance) will be achieved. In Figure 3, the heuristics that are chosen to be implemented (or satisficed) in the target system are indicated as check-marks ("√") inside the nodes. On the other hand, rejected candidates are represented as "**X**". Suppose the developer selects the *dead code elimination* heuristic, for satisficing *high control flow consistency* and *high data consistency*. Suppose the developer also selects the *minimization of the number of direct children* and *minimization of the response set* heuristics to satisfice *low control flow complexity*.

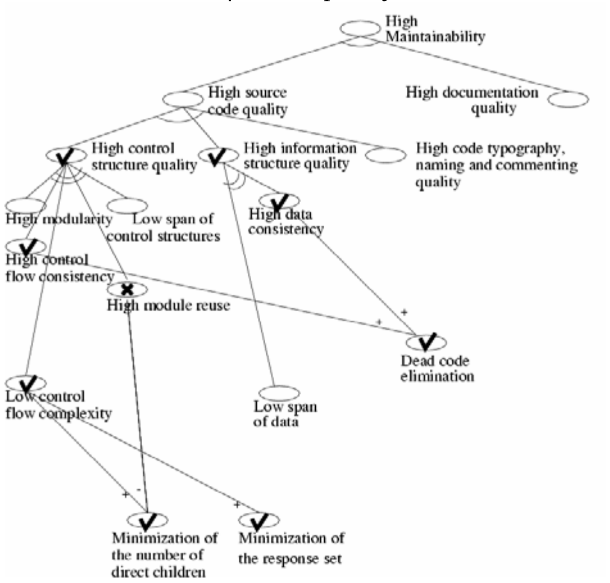
High Maintainability
High source code quality
High documentation quality
High control structure quality
High information structure quality
High code typography, naming and commenting quality
High modularity
Low span of control structures
High data consistency
High control flow consistency
High module reuse
Dead code elimination
Low control flow complexity
Low span of data
Minimization of the number of direct children
Minimization of the response set
+
+
-
+
+

Figure 3 - Selecting among alternative heuristics.

Once the developer has selected the heuristics to be implemented, the impact of these selections on top-level softgoals has to be evaluated. The evaluation process can be viewed as working bottom-up, starting with bottom leaves of the graph representing heuristics. The evaluation process works towards the top of the graph, determining the impact of offspring softgoals on parent softgoals. The impact upon a parent softgoal is computed from the contributions that all the offspring softgoals make towards it. This impact is represented by assigning labels (**"√"** and "**X**") to the higher-level parent softgoals. Figure 3 shows that the heuristic *minimization of the number of direct children* which is satisficed (**"√"**) makes a negative contribution towards its parent softgoal *high module reuse*. The result is that softgoal *high module reuse* is denied ("**X**"). On the other hand, the heuristic *dead code elimination* that is satisficed (**"√"**) makes a positive contribution towards its parent softgoals *high control flow consistency* and *high data consistency*. Thus, both softgoals are satisficed (**"√"**). Suppose a softgoal receives contributions from more than one offspring. Then the contribution of each offspring towards the parent is determined, using the above approach, and the individual results are then combined.

5 CONCLUSIONS

The major contributions of this work include using the *NFR framework* to model two particular software qualities, maintainability and performance. We identified and described many heuristics that affect these software qualities and can be implemented in a system's source code to achieve them. We implemented some of the heuristics in two medium-sized software systems and then collected measurements to determine the effect on software quality. We have presented a methodology for selecting the set of optimization heuristics that will improve the system's software quality the most – maintainability and performance - while minimizing negative effects.

6 REFERENCES

[1] Lawrence Chung, *"Non-Functional Requirements in Software Engineering",* PhD thesis, University of Toronto, Toronto, Ontario, Canada, 1993.

[2] J. R. Hagemeister, "A Metric Approach to Assessing the Maintainability of Software", Master's thesis, University of Idaho, Moscow, Idaho, 1992.

[3] J. Arthur and K. Stevens, "Assessing the Adequacy of Documentation Through Document Quality Indicators", in *Proceedings Conference on Software Maintenance*, pp. 40-49, IEEE CS Press, 1989.

[4] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach* (Morgan Kaufmann, San Mateo, CA, 1990).

[5] IEEE, *Standard Glossary of Software Engineering Terminology*, 1990.

[6] R. M. Baecker and A. Marcus, *Human Factors and Typography for More Readable Programs,* (Addison Wesley, 1989)

[7] P. W. Oman and C. R. Cook, Typographic Style is More than Cosmetic, Communications of the ACM (CACM)", Vol.33, pp. 506—520, Communications of the ACM (CACM), 1990.

[8] T. Pearse and P. Oman, "Maintainability Measurements on Industrial Source Code Maintenance Activities", in *Proceedings 1995 International Conference on Software Maintenance,* pp. 295-303, IEEE CS Press, 1995.