CORRELATION BETWEEN CODING STANDARDS COMPLIANCE AND SOFTWARE QUALITY

Author: Wojciech Basalaj

Co-Author: Frank van den Beuken

Programming Research, 9-11 Queens Road, Hersham, Surrey KT12 5LU, UK
Frank_van_den_Beuken@programmingresearch.com

ABSTRACT

Software Quality has different meaning to different people. The ISO 9126 standard was developed to introduce clarity and establish a framework for quality to be measured. This paper aims to explore how *Internal Quality* characteristics of a software system (source code) can be measured effectively. Instead of relying on traditional software metrics, which are shown to be a poor predictor of underlying software quality, we advocate measuring compliance to a coding standard. We show qualitative and quantitative evidence of how adoption of a coding standard helps organizations in improving the quality of their C/C++ software.

Keywords: Software Quality Modelling, Coding Standards, Software Metrics, Statistical Analysis

1. ISO 9126 QUALITY MODEL

The ISO 9126-1 standard [4] has been introduced to formalise the notion of Quality of a Software System. 3 distinct aspects are considered:

- Internal Quality measured for a non-executable form of the Software System, e.g. its source code.
- External Quality, which pertains to the run-time behaviour of the system, as experienced during dynamic test.
- Quality in use, which addresses the degree to which user goals and requirements are fulfilled.

Internal and External Quality can be further categorised into 6 separate characteristics:

- Functionality
- Reliability
- Usability
- Efficiency
- Maintainability
- Portability

Each of these 6 characteristics can be further subdivided, and there are 27 sub-characteristics in total.

Quality in Use has been divided into 4 characteristics:

- Effectiveness
- Productivity
- Safety
- Satisfaction

ISO 9126-1 advocates measuring each of these characteristics, but does not specify how. Examples of suitable metrics are given in Technical Reports: 9126-2 [5], 9126-3 [6], 9126-4 [7]. The standard stipulates that with suitable choices of metrics Internal Quality should predict, or in other words correlate with External Quality, which in turn should predict Quality in Use.

In this study we will be focusing on the Satisfaction Quality in Use characteristic. We will attempt to demonstrate that this characteristic can indeed be predicted by measuring Internal Quality of a software system, see Section 4.1. We will also be examining empirical evidence of a correlation between Internal and External Quality measures, see Section 3.

Prior to conducting such a study we needed to settle on suitable metrics for Internal Quality. ISO 9126-3 [6] is a Technical Report that proposes such metrics. The vast majority of them are of the following form: percentage of items (functions, variables, etc.) meeting a specific requirement. There are a number of problems with such a definition of metrics. Their calculation cannot be easily automated, and their value needs to be determined by comparing implementation and design documents with specification. These metrics indicate how much work on the project has been completed, rather than the underlying quality of the implementation. Such metrics represent good project management practice for green-field projects, and cannot be applied easily when part of the system is re-engineered. Lastly, quality or lack thereof is not seen as an attribute of source code, as none of the proposed metrics are based on direct measurements on source code. This is against the guidance of ISO 9126-1 [4] page 15.

Prior to ISO 9126 there has been a vast amount of research devoted to software metrics [2]. These traditional metrics, such as Cyclomatic Complexity or Estimated Static Path Count, are concerned with the structure of a function, vocabulary of a source file, etc. Therefore, they may yield the same values for drastically different versions or stages of a Software Product, e.g. Cyclomatic Complexity for pseudo code stage may be the same as for the final implementation. Moreover, there is no well-defined and substantiated mapping for these metrics to ISO 9126 characteristics. We examine possible correlations of such software metrics with Quality in Use metrics in Section 4.2.

2. CODING STANDARDS

Nowadays, increasingly more emphasis is given to following best practice, and defining and enforcing coding standards, especially for high cost of failure software projects. Compliance to a coding standard is often treated as a pass/fail test. However, a different approach is possible, where the level of compliance is measured, either as the absolute number of violations for a particular source file, module or component, or normalised by the size of the entity, e.g. number of lines of code. This would allow correlating compliance with measurements of other aspects of the product, e.g. run time behaviour or user experience.

The most popular coding standard in the public domain for the C language is MISRA-C [12][13]. It constitutes a subset of the C language that restricts usage of poorly defined or unsafe constructs. Less emphasis is given to presentational aspects: naming conventions and layout. Until recently, no such definitive coding standard was available for the C++ language. The first and probably most complete is High Integrity C++ [14]. More recently, the Joint Strike Fighter Air Vehicle C++ Coding Standards [10] were released, demonstrating the growing industrial acceptance of using coding standards. Other C++ guidelines tend to focus on specific programming aspects [3][11][16][17].

The rules of these coding standards represent common pitfalls with developing in the corresponding programming language, and have been derived either from experience or on theoretical grounds, by examining the language specification [8][9]. Therefore, counting the number of violations of such rules in a Software Product appears well founded, and intuitively corresponds to a measure of its Internal Quality. This proposition is rigorously evaluated in Section 4.1.

3. QUALITATIVE RESULTS

We wanted to verify the proposal for measuring Internal Quality of a software product with real-world examples. We have engaged with some software companies, to find out what tangible benefits enforcement of a coding standard has given them. Two of them were able to offer broad qualitative statements, and these are documented in Section 3.1 and 3.2. However, they could not provide, in time for publication of this paper, any numerical data that would allow us to compare, for example, faults found in the field and compliance to a coding standard of specific software modules. However, another company had such data available, and we worked together to establish whether there were any correlations, see Section 4 for details.

3.1 Company A

They have been using MISRA-C:1998 [12] ever since historical process data have been collected. Some extra rules are enforced to do with naming conventions and limiting undefined behaviour. Typically, approximately 90% of the rules of this combined coding standard are adhered to for a project.

For a number of specific projects porting from one platform to another was required, and this was achieved with hardly any re-coding. This result was attributed to restricting undefined and implementation defined behaviour in their coding standard.

In their development process, unit testing occurs on a parallel track to coding, review and bench testing. By examining process data it was found that all the faults found in unit testing were also identified in the development track during code review (of which coding standard compliance is a part) or bench testing stages. Therefore, unit testing, despite being part of industry best practice, did not yield any new issues, apart from fulfilling its secondary role of verifying the specification. Subsequently, for some projects unit testing has been limited or dropped altogether in preference to proceeding straight to integration/system test.

3.2 Company B

The AUTOSAR[1] subset of MISRA-C:1998 [12] is used, as well as other proprietary coding standards, depending on the project, and this is mandated contractually.

The software projects are large, typically around 500KLOC. By defining a software platform, and making it conform to stricter rules on limiting implementation defined behaviour, they were able to migrate from one compiler and micro controller combination to another in a matter of weeks. This result is similar to that of Company A, see section 3.1.

Reuse is very common across projects, and coding standard rules on layout and naming conventions were found to be helpful in this regard.

4. QUANTITATIVE RESULTS

Company C has an ongoing programme for improving customer satisfaction. To this end they are collecting software fault reports from the field, and tracking them on a regular basis. The incidence of critical software faults tends to vary across their products, and the intention is to identify measurements on source code, i.e. Internal Quality metrics, that would correlate with these fault data, i.e. Quality in Use: Satisfaction metric. Once such source code factors are identified, it will be possible to re-engineer the software to minimise their value; and thus, likely to minimise the incidence of critical faults in released software.

Together with Company C we have collected code metrics for a number of their software products, and correlated them with the corresponding critical fault data. These code metrics fall into two categories:

- incidence of coding standard violations,
- traditional software metrics [2].

The results are documented in Sections 4.1 and 4.2 respectively.

4.1 Message Correlation

As a pilot study we focused on 18 software products written in C++, and owned by a single business unit. Critical fault data for each of the products was available, covering a period of 12 months. In order not to disadvantage large projects, we normalised these measurements of Quality in Use: Satisfaction by the size of the corresponding code base, i.e. amount of KLOC.

Rather than narrowing the study to some specific coding standard or guidelines (see Section 2), we decided to include as many coding rules as possible, in our search for the ones that will correlate with the fault data. QA C++, static analyser for C++ from Programming Research, includes nearly 900 rules ranging from ISO Compliance and Undefined Behaviour [9], Best Practice [3][11][14][16][17], to code layout conventions. This includes rules pertaining to individual source files as well as issues occurring across files, see Table 1 for examples.

confid ence	msg#	QA C++ message text
99.5%	1512	'%1s' has external linkage and is declared in more than one file.
99%	1508	The typedef '%1s' is declared in more than one file.
99%	2085	For loop declaration of '%1s' is hiding existing declaration.
99%	4239	Class type control loop variable '%1s' modified in loop block.
97.5%	4217	Variable '%1s' is not accessed after this initialisation before it is next modified.
97.5%	4237	Class type control variable '%1s' not declared here.
97.5%	3600	This 'int' literal is an octal number.
95%	1505	The function '%1s' is only referenced in one translation unit.
95%	4243	Multiple class type loop control variables found: '%1s'.
95%	4325	Variable '%1s' is not accessed further.
95%	4004	Continue statement found.
95%	4208	Variable '%1s' is never used.
0%	2015	This function may be called with default arguments.

Table 1. Message correlation with critical fault data for a sample of QA C++ messages

For every software product we calculated the occurrence of each QA C++ message, and normalised the measurements by the size of the product in KLOC. While we could look for correlations between these raw measurements for fault data and message frequencies, this would make an unnecessary assumption that both of these populations of measurements were distributed similarly.

Instead, we decided to use ranks of the measurements only. If we were to order the software products according to fault data frequency, and for a given QA C++ message according to its frequency of occurrence, similarity between these two orderings would imply a positive correlation between the message and fault data. Considering that we are dealing with a large number of products, from statistical standpoint, it is not necessary that these orderings are identical, for there to be a significant correlation. Given that the number of permutations of 18 entities: 18! = 18*17*…*2 = 6,402,373,705,728,000 is a staggeringly large number, if a pair of orderings is within the 5% group that are the most similar, we can say with 95% confidence that they are correlated. 95% confidence interval is usually considered the minimum level to achieve statistical significance.

This leaves the question of how we are going to judge similarity between two given orderings of 18 products. Spearman's Rank Correlation Coefficient R_s [15] is a non-parametric statistical test, meaning that it works on the ranks of measurements. It evaluates to 1.0 if the orderings are exactly the same and -1.0 if they are exactly opposite, i.e. one is an inversion of the other sequence. The closer the value of R_s to 0 the less similar both orderings are. In this study we are only interested in positive correlations between Quality in Use and Internal Quality metrics: $R_s>0$. Given that we are dealing with 18 products, in order to have 95% confidence of a positive correlation between QA C++ message and fault data, the value of R_s needs to be no smaller than 0.401. Table 1 documents critical values of R_s for higher confidence intervals.

Confidence	95%	97.5%	99%	99.5%	99.9%
Critical Value of R_s	0.401	0.472	0.550	0.600	0.692

Table 2. Critical Values of Spearman's Rank Correlation Coefficient R_s for 18 entities

The first 12 rows of Table 1 list QA C++ messages that are positively correlated with critical fault data for the 18 software products under consideration, with at least 95% confidence. As an illustration the last row contains the message that has the value of R_s closest to 0. Figures 1-5 on page - 9 - display the correlation between the ranks of fault and message frequencies for each software product as a scatter plot, for a representative selection of messages from Table 1. Dots (software products) that lie on the y=x (diagonal) line represent complete agreement between the ranks. In Figure 1 dots are much closer to the diagonal line than in Figure 5, which visually confirms the accuracy of the Spearman's Rank Correlation Coefficient. Figure 6 corresponds to the message with the smallest value of R_s; for convenience both positive y=x and negative y=19-x correlation lines are drawn. As can be seen dots are equally distant from both diagonal lines.

This result can be interpreted as follows: there is at least 95% likelihood that 12 QA C++ messages detailed in Table 1 are positively correlated with critical faults in 18 software products under consideration. This allows us to assume that by re-engineering these products to reduce the incidence of these messages, future occurrence of critical faults may also be reduced. As the organisation is interested in improving customer satisfaction, targeting these messages and monitoring their frequency can supplement the existing quality procedures.

It is worth pointing out that these 12 recommended messages are Best Practice rules, rather than rules targeting Undefined Behaviour, e.g. array access out of bounds, or division by 0. Such rules targeting potential 'bugs' are unlikely to occur frequently in the code. If for 18 products most frequencies are 0 apart from a few, the Spearman's Rank Correlation Coefficient will not exceed the critical value, and so the corresponding QA C++ message will not be flagged up as correlated with critical fault data. Therefore, it is necessary to supplement rules/messages identified by this statistical procedure with rules targeting bugs, portability issues, and other priorities identified for the software products in question.

4.2 Metrics Correlation

Apart from looking for correlations between critical faults and QA C++ messages, we were interested in examining whether traditional software metrics [2] could be of use. QA C++ calculates several function, file and class based metrics. We have recorded the average, maximum and standard deviation value of every metric for each of the 18 software products. We then calculated the values of Spearman's Rank Correlation Coefficient R_s between the critical fault and these metric data across the 18 products, which are collected in Table 3. Critical value of R_s at 95% confidence level is 0.401, and none of the metrics meet that for either average measurement, maximum or standard deviation. Therefore, we could not recommend any of these software metrics to be included in the quality initiative.

5. SUMMARY

In this paper we have proposed using coding standards compliance as a measure of Internal Quality of a Software System. The validity of this metric has been confirmed on a group of real-world software products, as for a number of coding rules it was found to correlate with a metric for Quality in Use: Satisfaction characteristic. Also, compliance to a coding standard has been found by two separate organisations to positively impact External Quality: Portability characteristic of their software.

User satisfaction is a concrete concept, and can be measured, e.g. by recording faults in released software. Coding standards compliance can also be easily measured, and subsequently improved, but does not directly map to improved user experience. However, this could be inferred, if a correlation between user satisfaction and compliance to coding rules is found, as is the case in this paper. An interesting topic for a future study would be to empirically demonstrate validity of this cause and effect hypothesis, by examining whether incidence of faults will be reduced in proportion to improvement in coding standards compliance.

Metric	avg	max	Std dev
Class metrics			
Coupling to other classes	0.041	0.005	0.043
Deepest inheritance	0.083	0.166	0.100
Lack of cohesion within class	-0.012	-0.061	-0.046
Number of methods declared in class	-0.098	-0.020	-0.023
Number of immediate children	0.055	0.025	0.061
Number of immediate parents	0.055	0.034	0.055
Response for class	-0.031	-0.057	-0.031
Weighted methods in class	-0.017	-0.034	-0.069
Function metrics			
Cyclomatic complexity	0.087	-0.141	-0.234
Number of GOTO's	-0.153	-0.238	-0.154
Number of code lines	-0.061	-0.068	-0.256
Deepest level of nesting	0.103	0.234	0.087
Number of parameters	0.129	0.192	0.122
Estimated static program paths	-0.362	n/a[§]	0.084
Number of function calls	-0.102	-0.019	-0.239
Number of executable lines	0.017	0.018	0.009
File metrics			
Comment to code ratio	0.283	0.153	0.287
Number of distinct operands	-0.220	-0.239	-0.304
Number of distinct operators	-0.035	0.260	0.124
Total preprocessed code lines	-0.074	0.142	-0.087
Total number of tokens used	-0.144	0.040	-0.138
Total unpreprocessed code lines	-0.073	0.077	-0.117
Total number of variables	-0.187	-0.044	-0.261

[§] For technical reasons we were not able to accurately calculate this value.

Table 3. Metrics correlation with fault data

Critical value of *Rs* at 95% confidence level is 0.401

REFERENCES

[1] Automotive Open System Architecture, www.autosar.org

[2] N.E. Fenton, S.L. Pfleeger. *Software Metrics: A Rigorous Approach*. 2nd edition. PWS, Boston, 1998

[3] M. Henricson, E. Nyquist, N. Erik. *Industrial Strength C++: Rules and Regulations*. Prentice Hall, 1997

[4] ISO/IEC 9126-1:2001. *Software engineering – Product quality – Part 1: Quality model.*

[5] ISO/IEC TR 9126-2:2003. *Software engineering – Product quality – Part 2: External metrics.*

[6] ISO/IEC TR 9126-3:2003. *Software engineering – Product quality – Part 3: Internal Metrics.*

[7] ISO/IEC TR 9126-4:2004. *Software engineering – Product quality – Part 4: Quality in use metrics.*

[8] ISO/IEC 9899:1990. *Programming languages – C.*

[9] ISO/IEC 14882:2003. *Programming languages – C++.*

[10] Lockheed Martin Corporation. *JSF AV C++ Coding Standards.* http://www.research.att.com/~bs/JSF-AV-rules.pdf 2005

[11] S. Meyers. *Effective C++: 55 Specific Ways to Improve Your Programs and Designs*. 2nd edition. Addison Wesley, Boston, 2005

[12] MIRA, *MISRA-C:1998 - Guidelines for the Use of the C Language in Vehicle Based Software*. www.misra-c.com, 1998

[13] MIRA, *MISRA-C:2004 - Guidelines for the use of the C language in critical systems.* www.misra-c.com, 2004

[14] Programming Research. *High Integrity C++ Coding Standard Manual.* www.codingstandard.com, 2004

[15] S. Siegel. *Nonparametric Statistics for the Behavioral Sciences*. McGraw-Hill Book Company, Berkshire, 1956.

[16] H. Sutter, A. Alexandrescu. *C++ Coding Standards: 101 Rules, Guidelines, and Best Practices.* Addison Wesley, Boston, 2004

[17] H. Sutter. *Exceptional C++*. Addison Wesley, 1999

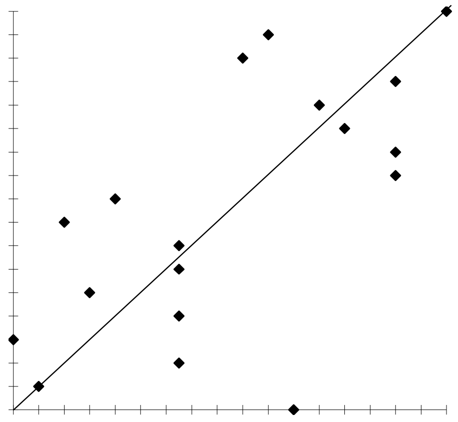
Figure 1. correlation for message 1512
R_s=0.649, confidence interval 99.5%

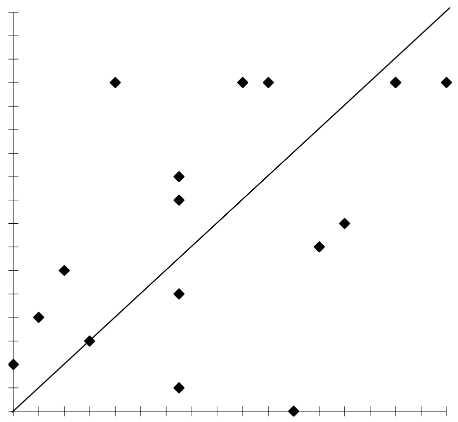
Figure 2. correlation for message 1508
R_s=0.568, confidence interval 99%

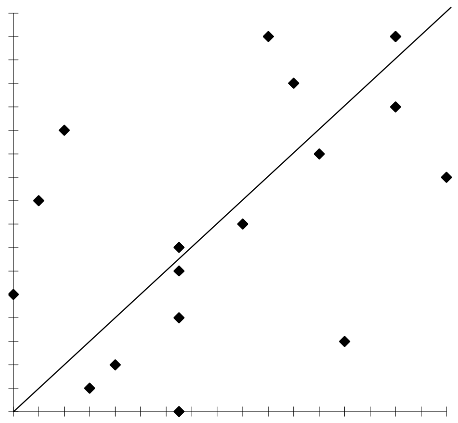
Figure 3. correlation for message 4217
R_s=0.533, confidence interval 97.5%

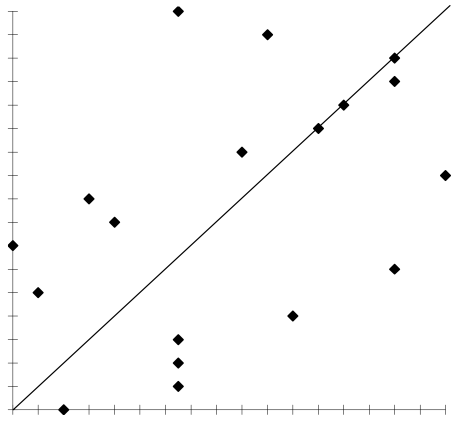
Figure 4. correlation for message 1505
R_s=0.466, confidence interval 95%

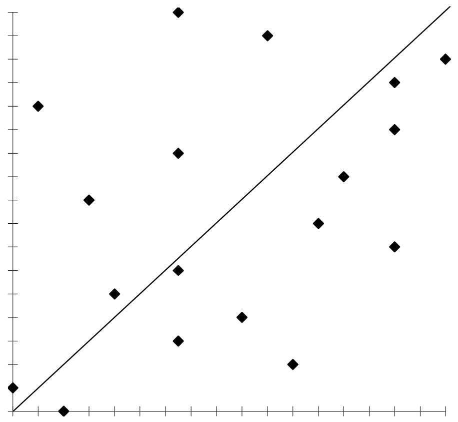
Figure 5. correlation for message 4208
R_s=0.403, confidence interval 95%

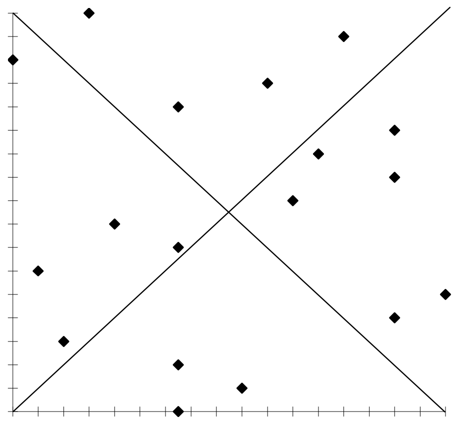
Figure 6. correlation for message 2015
R_s=0.001, i.e. no correlation