PROCEEDINGS OF SCIENCE

# Code Quality from the Programmer's Perspective

**Paolo Tonella***
*Fondazione Bruno Kessler*
*38050 Povo, Trento (Italy)*
*E-mail:* tonella@fbk.eu

**Surafel Lemma Abebe**
*Fondazione Bruno Kessler*
*38050 Povo, Trento (Italy)*
*E-mail:* surafel@fbk.eu

Code quality has traditionally been decomposed into internal and external quality. In this paper, we discuss the differences between these two views and consider the contexts in which either of the two becomes the main quality goal. We argue that for physics software the programmer's perspective, focused on the internal quality, is the most important one. Then, we revise the available tools and techniques for the verification and improvement of the internal code quality, having in mind the programmer's perspective. We conclude with a list of challenges for research in software engineering about aspects of the internal code quality that are largely neglected, but affect deeply the programmer's ability to carry out code modification and bug fixing tasks. Such aspects revolve around the way in which the natural language is embedded into the code as a form of domain modeling.

*XII Advanced Computing and Analysis Techniques in Physics Research*
*November 3-7 2008*
*Erice, Italy*

*Speaker.

 

PoS(ACAT08)001

## 1. Introduction

Quality in general and software quality in particular can be defined in several different ways, but any definition is relative to some person, who represents the target of the delivered quality. Quality is value for some person, rather than absolute value. This is especially true for software quality, where we can identify at least two categories of stakeholders for which quality is a central issue: users and developers. For users, software quality means ease of use, no crash at run-time, compliance with the requirements, correctness, etc. For developers, it means good design, encapsulation of functionalities, proper modularization, meaningful identifier names, documented interfaces, etc.

These two views of the software quality are usually called *external quality* and *internal quality* of the software. Both are important and usually the assumption is made that the two are strongly related, so that it makes sense to invest in the improvement of both to eventually deliver a high quality product to the final user. For example, a good design may facilitate smooth adaptation of the software to a changed requirement or addition of new functionalities. A bad design may not prevent it, but may result in a system that contains more problems, due to the difficulty of evolving it, eventually delivering a lower quality (e.g., more bugs) to the user. The external, user's, view on the quality is focused on "what" the software does, regardless of how it is implemented, but clearly the "how" is strictly connected with the (quality of) the "what", in terms of correct and compliant implementation, ease of evolution and misbehaviors (e.g., crashes) exhibited over time.

While internal and external quality are both a key issue in a software project, different contexts demand for a different emphasis. Let us consider a commercial, closed source (industrial) context, compared to a free, open source one. In an industrial context, the customer decides the fortune of the software. However, the customer's needs and domain are only partially known by the developers. Hence, the risk of not matching the requirements accurately and correctly is probably the most important one. This justifies the large investment in system and acceptance testing, addressing the external quality view, that is typical of these organizations. Improvement of the internal code structure is considered of less importance and often there is no remaining time or resources for it. The open source context represents the other extreme of a continuum. Often, open source projects deal with the development of tools that are used by programmers or computer scientists, so domain and requirements are perfectly known and the risk of a requirement mismatch is relatively low. On the other hand, many developers, distributed world-wide, are often involved. This means that the design of the software is the key factor that decides its fortune, together with internal quality attributes, such as the understandability of the code and its capability of self-documentation.

We have been involved in the assessment and improvement of the quality of the code developed for the ALICE experiment at CERN since 1999. In our experience, this context has several similarities with the open source context. Developers of the software are also the users, and the domain is perfectly known. Many contributors are spread across many countries. The code is subject to a large amount of changes and evolution over time, which demands for a good internal structure and for an easy way to understand the internal functionalities. In this paper, we first give an overview of the traditional way to address the internal software quality problem, which is the main quality concern for ALICE. Then, we report the results obtained so far, thanks to a tool for the enforcement of coding conventions and for the detection of bad programming practices. Finally,

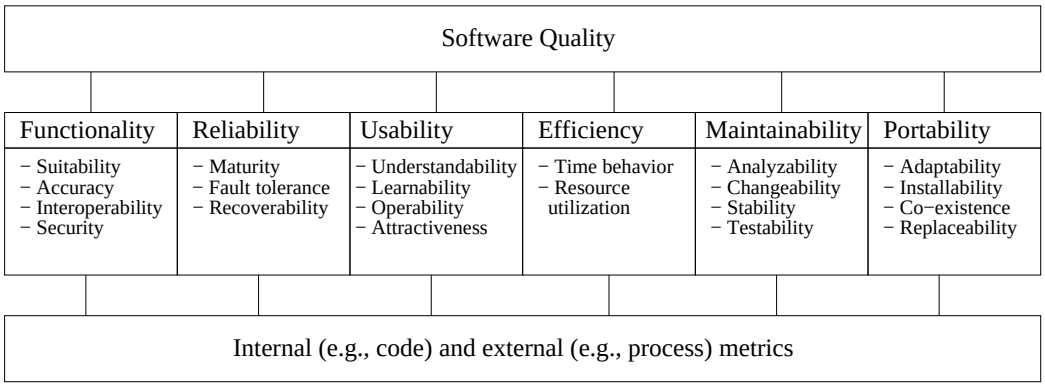


**Figure 1:** ISO 9126 software quality model

we consider one dimension not covered by any existing approach to the internal code quality: the quality of the lexicon used by programmers, which deeply affects code understandability and maintainability. We report some preliminary results obtained from a few empirical studies conducted on the ALICE code. We conclude the paper with the research directions that we consider most promising in trying to address this quality dimension – the programmer's lexicon.

## 2. Internal quality

The internal quality of a software system is usually defined with reference to a set of properties of interest, which give rise to subproperties and eventually measurable properties of the code. Such a decomposition of the various quality aspects, together with the related definitions and metrics, constitute what is called a *quality model*. Quality models are considered an essential tool for the assessment and improvement of the internal (as well as the external) software quality. In the following, we describe in detail one such quality model (ISO 9126) and we consider the benefits and limitations of using it in a software project such as the ALICE offline software at CERN.

### 2.1 Quality models

Several different quality models have been proposed in the literature over time (e.g., Boehm's model [3], McCall's model [11]), differing by quality attributes considered and metrics used to quantify them. One notable attempt to unify and give a common structure to such models was conducted by the International Organization for Standardization (ISO), with its standard n. 9126 [7].

Figure 1 shows an overview of the quality attributes considered in this model. At the top level, software quality is split into:

**Functionality** A set of attributes that bear on the existence of a set of functions and their specified properties. The functions are those that satisfy stated or implied needs.

**Reliability** A set of attributes that bear on the capability of the software to maintain its level of performance under stated conditions for a stated period of time.

**Usability** A set of attributes that bear on the effort needed for use, and on the individual assessment of such use, by a stated or implied set of users.

**Efficiency** A set of attributes that bear on the relationship between the level of performance of the software and the amount of resources used, under stated conditions.

**Maintainability** A set of attributes that bear on the effort needed to make specified modifications.

**Portability** A set of attributes that bear on the ability of the software to be transferred from one environment to another.

Quality characteristics are then decomposed into sub-characteristics. For example, Maintainability can be further decomposed into Analyzability (easy of code analysis), Stability (capability to absorb changes with minimal impact), Changeability (amount of effort needed to perform a change), etc. Parts 2 and 3 of the standard [7] define possible metrics that can be used to quantify such quality attributes, divided into external and internal metrics.

### 2.2 Quality assessment and improvement

Assessment of the internal quality of a software project can be based on a quality model, such as the ISO 9126, briefly summarized in the previous section. The model is first instantiated in the specific software development context, by defining the relevant quality characteristics and sub-characteristics, as well as a way to quantify them. Computing metrics for the selected quality (sub)characteristics may involve usage of automated code analysis tools, as well as execution of manual code inspections or interviews with the developers or the users. Data obtained from the testing phase (e.g., defect density) may be also used in the assessment. More generally, process metrics (e.g., average time needed to implement a change) can be also integrated into the assessment of the internal quality.

The most important tools that can contribute to the internal quality assessment are: (1) code metrics tools; (2) process and workflow management tools; (3) test automation tools. In addition, statistical packages for data analysis may be useful to get a picture of the ongoing phenomena. Data analysis allows for comparative assessments, such as trend of metrics (associated with quality characteristics) over time or comparison of different modules/subsystems based on selected internal metrics. Quality improvement actions are triggered by such analysis.

In open source contexts, which are quite similar to the ALICE project, some internal quality attributes are more important than others. Specifically, Maintainability of the code (including ease of understanding, changing, adapting, extending the code) is a key concern, together with Efficiency and Reliability. Since developers are geographically distributed and often lack a formal computer science background, Maintainability is heavily dependent on how the code is written, and the specific programming language used in the ALICE project, the C++ programming language, does not enforce a single programming paradigm or style. Hence the need for coding conventions that are periodically checked and provide guidance to code writing, so as to ensure some level of uniformity and understandability among partners.

Coding conventions are one way to ensure that specific aspects of the internal quality are properly taken into account. However, there are situations where no single coding convention is enough to ensure that "good" Object-Oriented design principles are followed. In such cases, one useful approach consists of what is called *bad smell detection*. A bad smell [8] is any symptom in the source code that indicates a possible design problem. Common examples of code smells are: duplicate

code, large method, large class, feature envy (a class that uses another class excessively), inappropriate intimacy (a class that depends on implementation details), lazy class, etc. The solution to a code smell is typically performing some refactoring [8] of the code, so as to improve the design of the software though small and controlled steps.

So far we focused on a *static* view of the software, considering its internal quality from the point of view of a programmer who has to read, understand and modify the code. However, quality assurance involves also *dynamic* assessment. For this, all the practices supported by the various testing techniques are available and offer an invaluable contribution, which is complementary but not a replacement of static code quality assessment. In this paper, the focus is on the static assessment, but this does not mean that testing (e.g., unit testing) is considered of less importance for the internal quality.

## 3. Code quality in the ALICE experiment

One of the ways to achieve code quality is to have a standard on how the code is written. In light of this, ALICE has defined and adopted a set of rules and guidelines classified into four major categories: Naming, Coding and Style rules, and Guidelines. These major categories address different internal aspects of code.

Naming rules focus on how the names of source code files, and classes, attributes, methods, local variables, etc. are written in the code. They allow the programmer to easily locate a file that contains a specific class or to understand the kind of an identifier (for example, data member or local variable). In general, these rules increase readability of the code and hence make the code easier to understand.

The second set of rules, Coding Rules, targets the code structure/organization to make reading, understanding and modification of the code easy. This set of rules protects the code from bad design, crash (e.g. due to shallow copies of objects), unwanted behavior, etc.

The last set of rules is Style rules. They ensure that the code is well documented. They specify the places where to put mandatory comments in the code, which makes finding a description for an element of the source code easy.

Guidelines give directions on how to write code that minimizes compilation time and avoids some problems that might arise due to use of some programming features (e.g. templates may cause problems when creating shared libraries).

### 3.1 Tools

When a system is developed in a complex, distributed environment like ALICE, verifying that the code is developed according to the adopted standards is a difficult task. Therefore tool support is particularly important. In this regard, a tool, *RuleChecker*, was developed for ALICE in 1999. The current version of the tool, which was delivered in 2008, is *NewRuleChecker*.

RuleChecker works on the pre-processed file of the source code, while NewRuleChecker works on the XML version of the source code. The conversion of the source code to the equivalent XML file is done using src2srcml [1]. The tools identify and report components of a system that

[1] http://www.sdml.info/projects/srcml/

violate the adopted conventions and need improvement. This allows to easily control the quality of the code in a similar fashion in all collaborating organizations and facilitates integration of the code. It also makes programmers more cautious of the quality of the code they are writing.

Since some rules can be checked locally while others require global information, NewRuleChecker takes advantage of a global fact base, which is precomputed by running a global analysis on all source files converted to XML. For example, a method which does not modify any data member can be made `const`. However, determining whether a method is or is not modifying any data member requires global analysis, since such modifications may happen within any directly or indirectly invoked methods. Another case where global analysis is necessary is when inheritance information is needed. In fact, an overriding method cannot have different `const` status than the overridden one. The fact base storing global information include facts such as inheritance relationships and invocation relationships.

RuleChecker and NewRuleChecker have been developed by the Software Engineering group at Fondazione Bruno Kessler, Trento, Italy[2]. Although these tools are not public domain, being exclusive property of the ALICE collaboration, usually they are made available upon request by the ALICE offline software project leader to other experiments as open source code.

### 3.2 Violations over time/ Density of violations over time

The violations of ALICE rules and guidelines, as reported by NewRuleChecker, are shown in Figure 2 and 3. The general trend which can be seen in Figure 2 shows that there is an increase in number of violations of the adopted conventions. However, this is due to the increase of the code size. If we consider density of violations (violations per Kilo lines of code text), we can observe that there is a stable or decreasing trend (see Figure 3). Guidelines (GC) and Style Rules (RS) remain almost constant (approximately 3 and 4 violations per kLOT respectively). Coding Rules (RC) violations have a slow decrease (from 8 to 7 violations per kLOT), while Naming Rules (RN) violations had a dramatic decrease (from approximately 8 to 2 violations per kLOT).

Style Rules and Guidelines violations remained almost constant possibly because they are perceived as less important (they refer to the style) and less mandatory (they are guidelines). Moreover, the initial number of violations is already very low. Naming Rules, on the other hand, are fundamental for program understanding and programmers worked a lot on their improvement over time. Coding Rules, which affect important quality aspects, such as performance, compilation time and misbehaviors, were also improved over time.

## 4. Quality of programmer's lexicon

The approaches described above, for the assessment and improvement of the internal code quality are based on metrics, usually computed on the code, violations of coding conventions and deviations from "good" Object-Oriented design principles (bad smells). However, programs are written also to communicate with other programmers, rather than to be just interpretable by a computer. Understandability and maintainability of the code is deeply affected by the quality of the lexicon used by programmers to define the various code entities. However, none of the quality

[2]http://se.fbk.eu

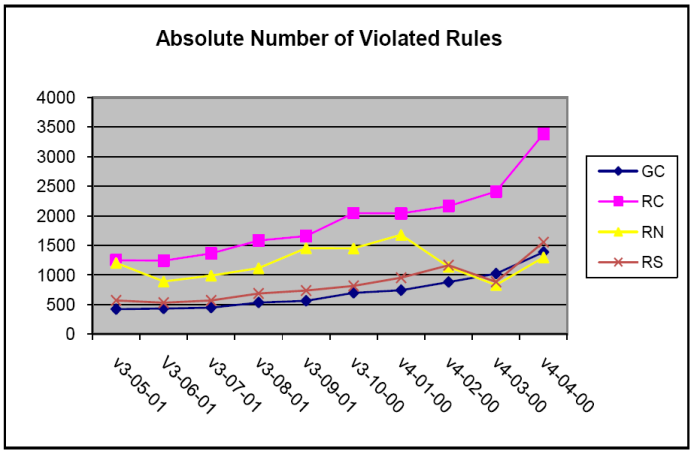


**Figure 2:** Total number of violations in ALICE for each category across versions.

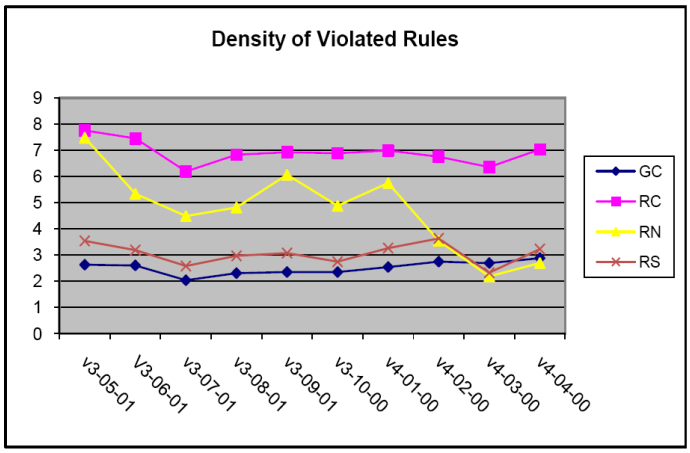


**Figure 3:** Density of violated rules in ALICE per kilo lines of text (kLOT) across versions.

assessment methods described above addresses this dimension of the quality directly. This observation triggered a new research line that deals with the quality of the programmers' lexicon.

The number of words used to build identifiers in the source code takes the lion share of the total number of words used in a system [5], and these identifiers play a major role during program comprehension for various tasks like maintenance and evolution of the system [2, 4, 6, 9, 10]. Good identifiers capture concepts of the real world and label the corresponding entities concisely and consistently, according to a defined structure [5]. In doing so they later help the programmer to easily comprehend the system and re-construct the conceptual model represented in the entities.

Despite the fact that many papers describe the importance of the identifier lexicon, there is very little tool support to manage it. Most of the tools/plug-ins developed so far are related to refactoring (in particular renaming) of identifiers. Deibenbock, F. and Pizka, M. [5] have developed a tool that constructs and maintains an identifier dictionary during software development. The tool provides a bijective mapping between concepts and names that helps to consistently and concisely use identifiers. This tool, however, requires an expert to construct the mapping, making its usage difficult for existing large systems. To improve this, Lawrie, D. et. al. [10] have developed a tool that is

based on a syntactically verifiable definition of consistent and concise identifiers. The tool was used to automatically spot those identifiers that violate the syntactic consistency and conciseness rules described in the paper.

### 4.1 Empirical studies

Two empirical studies have been conducted on the lexicon used by programmers in the ALICE experiment [1, 2]. These studies focus on three aspects of the lexicon of the code: (1) stability of the lexicon, (2) relationship among the lexicon groups used in different elements of the source code and (3) concepts represented by frequent terms.

#### 4.1.1 Stability of the lexicon

The study on the stability of the lexicon in ALICE was conducted to see if the lexicon used in the system undergoes any evolution, in a similar way as the system structure does. It was studied using the stability metrics defined in [2]. In short, such metrics are defined as the cosine similarity between feature vectors for the code entities in the current and in the previous software release. Such vectors capture respectively the lexicon used in a software entity (e.g., term occurrence in identifiers) or the code structure (e.g., called methods, control flow complexity). The study results show that the lexicon is more stable than the structure (see Figure 4, where the lexical stability is represented by the solid line and structural stability is represented by the dashed line).

As the source code evolves over time, the stability of both structure and lexicon is generally increasing, except when there are major changes in the system (e.g., version 4.01). The similar increasing pattern observed for both lexicon and structure, however, doesn't come from the same statistical distribution (as shown by a Wilcoxon non-parametric test, with confidence level 0.05 [2]), which indicates that the two are independent. This independent stability of the lexicon points out that the process of lexicon evolution is separated from the process of code evolution, which indicates good capability of vocabulary reuse when building new code entities. A possible interpretation is that ALICE programmers take a lot of care while changing the lexicon, since this is part of the common knowledge shared within the collaboration. They seem to try to reuse the existing lexicon whenever possible. This is also well reflected while creating new identifiers. On average, 56% of the new identifiers in ALICE are constructed using only terms already existing in the lexicon [1].

The stability of the lexicon was also studied by investigating the frequency of changes to program entities (particularly renaming) due to identifier refactoring. This investigation shows that there is almost no change of lexicon due to identifier refactoring in ALICE. This result can be explained in two ways. One explanation augments the above finding by confirming that programmers do a careful domain analysis while using/re-using vocabulary terms. Whereas the second possible explanation is a programmers resistance to lexicon changes or lack of tool support for refactoring in most C++ development environment. Distinguishing between these two possible hypotheses requires further investigation of the lexicon's quality, which is part of our planned future work.

#### 4.1.2 Relationship among the lexicon groups used in different elements of the source code

The relationship among different lexicon groups was studied to investigate how much concepts

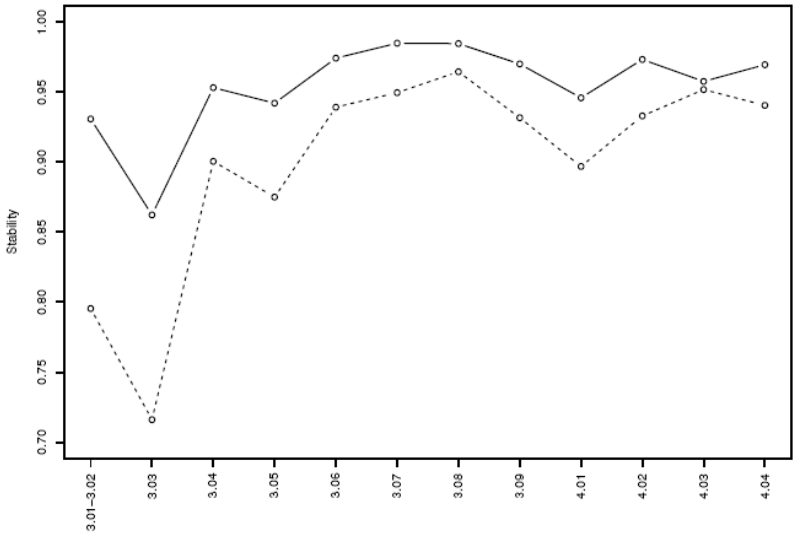


**Figure 4:** Lexical (solid line) and structural (dashed line) stability for ALICE.

are shared among different lexicon groups and to see if terms that are changing (added/deleted) are shared among lexicon groups.

The relationship among the lexicons was studied for class, attribute, function, parameter and comment lexicons. This empirical study shows that the class lexicon has the highest number of shared terms when compared pair wise with the other lexicon categories. This, as described in [1], shows that the core concepts represented in the class lexicon are reflected in the other lexicon groups. This may come from the adoption of "good" Object-Oriented design principles, prescribing that the system is modeled around its core domain entities, represented as classes. The comment lexicon also contains a high percentage of shared terms with the union of the other lexicon groups. However, there are only a small number of shared terms that are changing (added/deleted) together between the comment lexicon and the union of the other lexicon groups. This could be interpreted as a possible indication of obsolete comments, which are not updated when the code changes.

### 4.1.3 Concepts represented by frequent terms

The other focus of the empirical studies is the set of terms which are most frequently used by programmers and specifically whether they reflect domain terms or not. The analysis was conducted by collecting the top 15 frequent terms of the different lexicon groups (class, attribute, function, parameter and comment lexicons) from eight versions of the ALICE software and classifying them into domain or non-domain terms manually.

The domain terms found in the top 15 most frequent terms in the various lexicons across the eight versions are: *hit, track, pad, digit, cluster, chamber, pho, muon, segment, rec, event, tr, rich, tpc, trigger, tof, mp, tp, phosrp, cpv, emc, ppsd, energy, ev, sector*, and *rh*. There is a high number of shared terms among the top 15 terms of class, attribute and function lexicons, compared to comment and parameter lexicon. This result indicates that most frequent terms are good source of information for extracting domain knowledge. The result was confirmed by feedback from the ALICE developers.

### 4.2 Discussion

The main results of the empirical studies conducted so far on the ALICE code lexicon can be summarized as follows:

- The lexicon tends to remain stable over time, but we don't know if this is due to the lack of tool support for its evolution, or to an intrinsic stability.
- The class lexicon and the most frequently used terms represent core domain concepts, that can be regarded as a good starting point for the extraction of semantic knowledge from the code.

These studies are mainly observational and led to no directly usable tool or guideline for programmers. However, they represent an important first step in our research agenda, that ultimately aims at supporting programmers in the management and evolution of the immense knowledge base embedded in identifiers and in the terms used to build them. Such a knowledge is a cornerstone for the quality of a software system, hence we think it deserves adequate tool support, in a similar way as tools support code writing and code evolution. We plan to define lexicon bad smells and develop lexicon refactoring tools, that help evolving and improving the lexicon of a program. We also plan to investigate how to remove the obstacles that make the lexicon inflexible (too stable), if any. We also want to extract automatically domain knowledge from the code lexicon, so as to make it available to other programmers, to simplify lexicon evolution and identifier construction in the future.

## 5. Future research

Our future work will focus on domain modeling through ontology and on the relationship between domain terms and identifiers used in programs. This will be used to develop tools supporting identifier construction and improvement. We will also develop tools that help developers to easily understand, search and browse the code using concepts through automatically built dictionary/glossary of terms used in the program.

## References

[1] S. L. Abebe, S. Haiduc, A. Marcus, P. Tonella, and G. Antoniol. Analyzing the evolution of the source code vocabulary. In *Proc. of CSMR 2009*, March 2009.

[2] G. Antoniol, Y.-G.Gueheneuc, E. Merlo, and P. Tonella. Mining the lexicon used by programmers during software evolution. In *Proc. of ICSM 2007*. IEEE Computer Society, October 2007.

[3] B. Boehm. *Characteristics of software quality*. North-Holland, 1978.

[4] B. Caprile and P. Tonella. Nomen est omen: Analyzing the language of function identifiers. In *Proc. of WCRE 1999*, October 1999.

[5] F. Deibenbock and M. Pizka. Concise and consistent naming. In *Proc. of IWPC 2005*. IEEE Computer Society, May 2005.

[6] J. L. Elshoff and M. Marcotty. Improving computer program readability to aid modification. *Communications of the ACM*, August 1982.

[7] International Organization for Standardization. *Information Technology–Software Product Evaluation: Quality Characteristics and Guidelines for their Use, ISO/IEC IS 9126*. ISO, Geneva, 1991.

[8] M. Fowler and K. Beck. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 2000.

[9] S. Haiduc and A. Marcus. On the use of domain terms in source code. In *Proc. of ICPC 2008*, June 2008.

[10] D. Lawrie, H. Feild, and D. Binkley. Syntactic identifier conciseness and consistency. In *Proc. of SCAM 2006*. IEEE Computer Society, 2006.

[11] J. A. McCall, P. K. Richards, and G. F. Walters. *Factors in software quality*. Rome Air Development Centre, 1977.