Acta Cybernetica **21** (2013) 135–147.

A Probabilistic Quality Model for C# – an Industrial Case Study*

Péter Hegedűs†

Abstract

Both for software developers and managers it is crucial to have clues about different aspects of the quality of their systems. Maintainability is probably the most attractive, observed and evaluated quality characteristic of all. The importance of maintainability lies in its very obvious and direct connection with the costs of altering the behavior of the software.

In this paper we present an existing approach and its adaptation to the C# language for estimating the maintainability of the source code. We used our model to assess the maintainability of the C# components of a large international company. We analyzed almost a million lines of code and evaluated the results with the help of IT professionals of our industrial partner.

The application of our method and model was successful as the opinions of the developers showed a 0.92 correlation with the maintainability values produced by our C# maintainability model.

Keywords: software maintainability, ISO/IEC 9126, C# quality model, industrial case study

1 Introduction

Both for software developers and managers it is crucial to have clues about different aspects of the quality of their systems. The information can mainly be used for making decisions, backing up intuition, estimating future costs and assessing risks. The ISO/IEC 9126 standard [13] defines six high-level product quality characteristics which are widely accepted both by industrial experts and academic researchers. These characteristics are: *functionality*, *reliability*, *usability*, *efficiency*, *maintainability* and *portability*. The characteristics are affected by low-level quality properties that can be *internal* (measured by looking inside the product, e.g. by analyzing the source code) or *external* (measured by execution of the product, e.g. by performing testing).

*The publication is supported by the European Union and co-funded by the European Social Fund, TÁMOP-4.2.2/B-10/1-2010-0012.

†University of Szeged, E-mail: hpeter@inf.u-szeged.hu

DOI: 10.14232/actacyb.21.1.2013.10

Maintainability is probably the most attractive, observed and evaluated quality characteristic of all. The importance of maintainability lies in its very obvious and direct connection with the costs of altering the behavior of the software. Although, the quality of source code unquestionably affects maintainability, the standard does not provide a consensual set of source code measures as internal quality properties. The standard also does not specify the way how the aggregation of quality attributes should be performed. These are not deficiencies of the standard, but it offers a kind of freedom to adapt the model to specific needs.

We have introduced a practical quality model in one of our previous works [3] that differs from the other models (e.g. [4, 5, 6, 12, 17]) in many ways:

- It uses a large number of other systems as benchmark for the qualification.
- The approach takes into account different opinions of many experts, and the algorithm integrates the ambiguity originating from different points of view in a natural way.
- The method uses probabilistic distributions instead of average metric values, therefore providing a more meaningful result, not just a single number.

Although the introduced model proved to be useful and accepted by the scientific community, real industrial settings and evaluations are required to show that our solution is useful and applicable in real environments too. Additionally, the first published model is only a prototype for the Java language and weighted by a small number of researchers and practitioners. The goal of the current work is to develop a method and model for estimating the maintainability of the C# systems of a large international company. To achieve this goal, the following tasks were completed:

- Together with the industrial partner, we have introduced a new maintainability model for systems written in the C# language.
- A benchmark from the C# systems of the company has been created (almost a million C# code lines has been analyzed).
- A method and tool has been developed for qualifying the smaller components of the company's software using the benchmark - producing a relative measure for maintainability of the components (we were able to rank the components of the company).
- A new weighting has been created involving the developers and managers.
- According to the method and model, a large number of components have been evaluated.

The results were discussed after the evaluation and compared to the developers' opinions. The industrial application of our method and model was successful, as the opinions of the developers highly correlated with the maintainability values

produced by our C# maintainability model. This result shows that our probabilistic quality model is applicable in industry, as the industrial partner accepted the provided results and found our approach and tool very useful.

The rest of the paper is organized as follows. First, we give a short overview about the related work in Section 2. Then, in Section 3 we introduce our approach for estimating maintainability and describe the details of the case study setup. In Section 4 we present and evaluate the results of the case study. Section 5 collects the possible threats to the validity of our work. Finally, we conclude the paper in Section 6.

2 Related Work

A large number of works deal with software maintainability analysis using the source code of the software. Kanellopoulos et al. [14] apply data mining and clustering techniques to comprehend an object-oriented system and evaluate its maintainability. They successfully explore some quality attributes of the *JBoss* open source system. To understand whether some technologies consistently outperform others, Sartori et al. [19] use some crude indicators, such as the density of bugs and the time required to fix bugs. They find that there is a connection between these indicators and the programming language of the software. We also use source code metrics and static analysis but unlike these works we are particularly interested in the ISO/IEC 9126 based qualification of an industrial software. Moreover, we focus on the C# language and its specific quality attributes to be able to assess the maintainability of programs.

There are many other existing practical quality models. Some of them [2, 4, 7, 12, 16] adapt the ISO/IEC 9126 standard. These models are based on low level source code metrics and a method for aggregating these values to higher levels. There are other models also, e.g. Bansiya and Davis [4] present a hierarchical model for assessing the quality of object-oriented design. Their model focuses on design level metrics. Plösch et al. introduce a quality model [18] based on a technical topic classification. Their approach is more technical than the ISO/IEC 9126 and makes it easier to assign metrics - provided by static code analysis tools - to quality attributes. Letouzey presents a quality model and analysis model [15] which is used to estimate the quality and the technical debt of an application source code. He uses the concept of technical debt for expressing the quality of the software (larger debt indicates worse maintainability) based on different indicators calculated from the source code. Although we also introduce a new quality model, our primary purpose is to empirically validate its results with the help of an industrial case study rather than showing only a formal approach or results on small examples. We focus on the quality of C# systems while the mentioned quality models are either specific to Java or very general (applicable to any OO system).

There are many papers presenting industrial case studies of maintainability analysis of C# systems. But while we are interested in the ISO/IEC 9126 based qualification, they focus on different quality properties. Gatrell et al. [10] perform a study analyzing the refactorings in a commercial C# software comprising 270

versions. In another work Gatrell and Counsell focus on the fault-proneness [9] of the code, they document a study of change in commercial, proprietary C# software and attempt to determine whether a relationship exists between class changes and faults. The work of Goldschmidt et al. [11] concentrates particularly on the maintainability of the applied persistency techniques in C# systems.

3 Approach

3.1 Probabilistic Software Quality Model

Our probabilistic software quality model [3] is based on the quality characteristics defined by the ISO/IEC 9126 standard.

The computation of the high level quality characteristics is based on a directed acyclic graph whose nodes correspond to quality properties that can either be internal (low-level) or external (high-level). Internal quality properties characterize the software product from an internal (developer) view and are usually estimated by using source code metrics. External quality properties characterize the software product from an external (end user) view and are usually aggregated somehow by using internal and other external quality properties. The nodes representing internal quality properties are called *sensor nodes* as they measure internal quality directly. The other nodes are called *aggregate nodes* as they acquire their measures through aggregation. The edges of the graph represent dependencies between an internal and an external or two external properties. The aim is to evaluate all the external quality properties by performing an aggregation along the edges of the graph, called Attribute Dependency Graph (ADG).

Goodness is the term that we use to express how good or bad an attribute is. We assume that the goodness of each sensor node is not known precisely; hence it is represented by a random variable with a probability density function, the *goodness function*. For constructing goodness functions we make use of the metric histogram over the source code elements, as it characterizes the whole system from the aspect of one metric. As the notion of goodness is relative, we expect it to be measured by means of comparison with other histograms. Applying a distance function between two histograms, we get one goodness value for the subject histogram that is relative to the other histogram. In order to obtain a proper goodness function, we repeat this comparison with histograms of many different systems independently. In each case we get a goodness value which can basically be regarded as sample of a random variable from the range $[-\infty, \infty]$. Interpolation of the empirical density function leads us to the goodness function of the sensor node.

Besides constructing goodness functions for sensor nodes, we need a way to aggregate them along the edges of the ADG. We created an online survey where we asked many experts (both industrial and academic people) for their opinion about the weights between quality attributes. The number assigned to an edge is considered to be the amount of contribution of source goodness to target goodness. Consequently, the votes form a multi-dimensional random variable for each

aggregate node. Taking into account every possible combination of goodness values and weights, and also the probabilities of their outcome, we defined a formula for computing goodness functions for every aggregate node. It is the multi-dimensional extension of the classical linear combination. The aggregation method ensures that the goodness functions of aggregate nodes are really expressing the probabilities of their goodness from the aspect represented by the node.

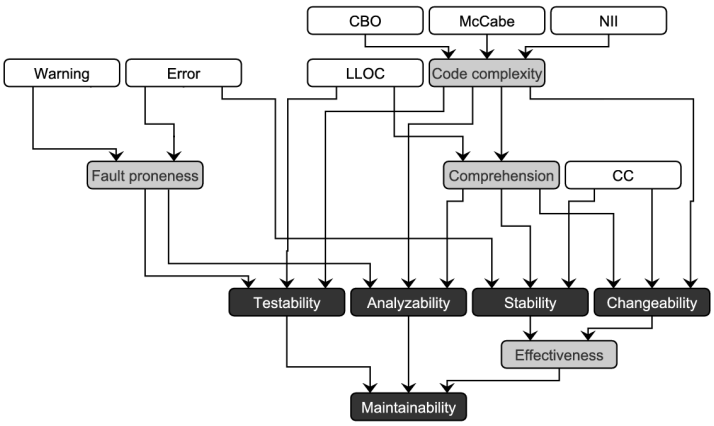

Figure 1: Java maintainability model

In our previous work [3] we have introduced a prototype ADG (see Figure 1) for Java language. To be able to perform the construction of goodness functions in practice, we have built a source code metric repository database, where we uploaded source code metrics of more than 100 open source and industrial software systems. To get an absolute measure for software maintainability, we calculated all the maintainability values of the systems in the repository. The resulting distribution of the values can be seen on Figure 2. As the values given by the quality model are unbounded we can assign to each system a rank value according to this distribution. The rank is a real value between 0 and 1 that objectively reflects the absolute maintainability value of a system based on the used repository database. Note that the rank is exactly the proportion of the systems in the repository that have a worse maintainability value than the subject system.

3.2 Adaptation of the Approach to C# Language

To adapt the previously described approach for qualifying the C# components of our industrial partner we have introduced a new ADG. As a result of a collective work the ADG shown in Figure 3 has been developed. It is much larger than the Java prototype ADG and contains some C# specific rule violations also. We chose FxCop [1] as a rule checker and built the number of different rule violations into the model as sensor nodes. The reason behind choosing FxCop is that it is a widely accepted rule checker in the C# world and our industrial partners already used this

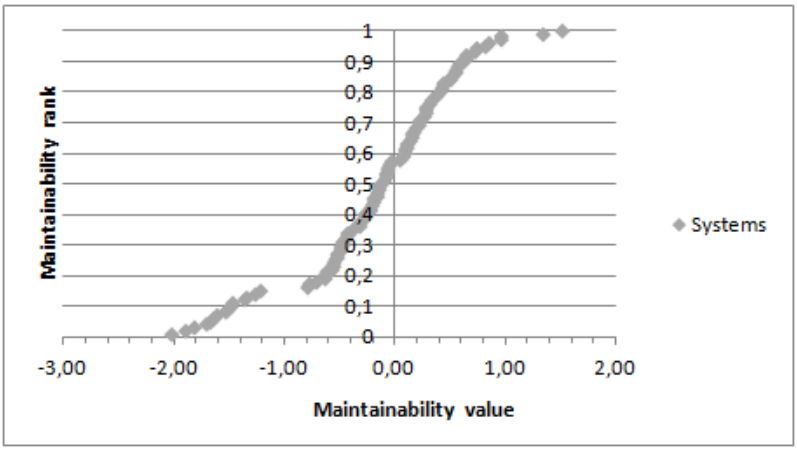

Figure 2: Distribution of the maintainability of benchmark systems

checker at the time of model construction. For calculating the source code metrics we used the Columbus toolset [8]. The sensor nodes included in the model can be seen in Table 1. We note that the differences in the sensor nodes compared to the Java model are rather general improvements on the model than specialties of C#. Only FxCop rule violations were added due to the adaption of the C# language.

Table 1: Sensor nodes in the model

DIT	Depth of inheritance tree	*NLE*	Nesting level
NOI	Number of outgoing invocations	*IR*	Interoperbility Rules
CBO	Coupling between object classes	*NR*	Naming Rules
McCabe	McCabe's cyclomatic complexity	*CC*	Clone coverage
LCOM5	Lack of cohesion on methods	*PR*	Performance Rules
DR, UR	Design Rules and Usage Rules[1]	*SR*	Security Rules
NII	Number of incoming invocations	*LLOC*	Logical code lines of
LLOC	Logical code lines of	*(class)*	classes
(method)	methods		

The model contains the following intermediate aggregated nodes:

- *Unit Complexity* – the class level complexity of the system.
- *System Complexity* – the complexity of the system as a whole.
- *Code Complexity* – the general complexity of the source code.

[1] All the rule sensor nodes refer to the number of FxCop rule violations of that group found in the system.

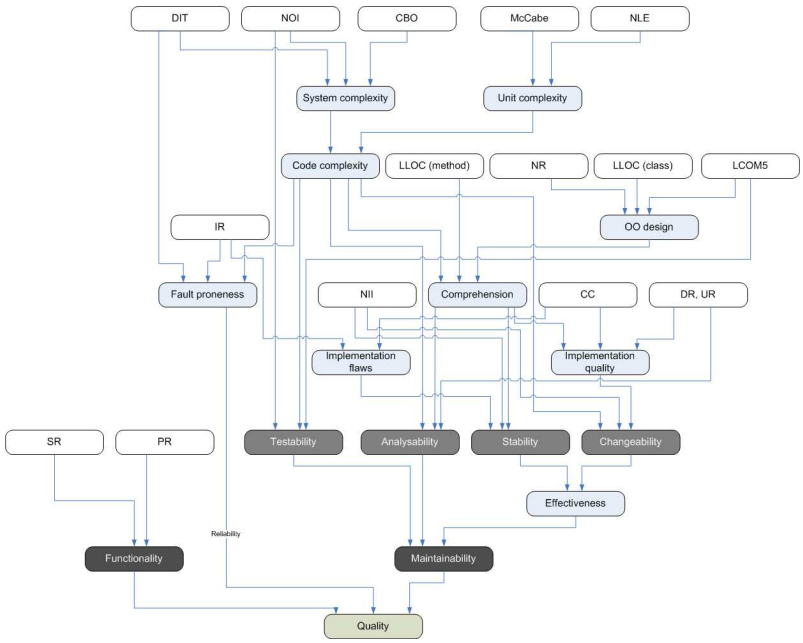

Figure 3: The created C# maintainability model

- *OO design* – the fulfillment of OO design principles.
- *Fault-proneness* – the fault proneness of the code due to dangerous program constructs.
- *Comprehension* – how easy it is to comprehend the code of the system.
- *Implementation flaws* – the implementation problems in the system.
- *Implementation quality* – the low level quality of the code implementation.
- *Effectiveness* – the effectiveness of the code change.

The ISO/IEC 9126 quality characteristics[2] in the model are the following:

- *Testability* – the capability of the software product to enable modified software to be validated.
- *Analyzability* – the capability of the software product to be diagnosed for deficiencies or causes of failures in the software, or for the parts to be modified.
- *Changeability* – the capability of the software product to enable a specified modification to be implemented, where implementation includes coding, designing and documenting changes.

[2]The model does not include all the characteristics defined by the standard yet.

- *Stability* – the capability of the software product to avoid unexpected effects from modifications of the software.
- *Functionality* – the capability of the software product to provide functions which meet stated and implied needs when the software is used under specified conditions. The functions satisfy the formulated or supposed conditions.
- *Maintainability* – the capability of the software product to be modified. Modifications may include corrections, improvements or adaptation of the software to changes in environment, and in requirements and functional specifications.
- *Quality* – the overall quality of the software system.

4 Results and Evaluation

To make the maintainability model work we had to build a benchmark database from different C# systems. Since our approach of creating benchmark database from large number of open source systems was not applicable, in this case we needed a new idea. As our industrial partner owns a huge amount of C# code itself and they were only interested in the code maintainability of their components compared to each other we decided to build a benchmark from their over 300 components[3]. This way we were able to give a relative maintainability value for each component (estimate the component's maintainability in comparison to other components). Moreover, we could define a ranking based on the relative maintainability between the components. Some basic properties of our partner's source code can be seen in Table 2.

Table 2: Basic properties of the industrial partner's software

Property	**Value**
Total number of logical lines of code	711 944
Total number of classes in the system	4 942
Total number of methods in the system	48 787
Number of components in the system	315

As a final step before the qualification of the components a weighting was introduced on the created C# ADG. 7 IT professionals from our industrial partner and 5 academical co-workers voted on the importance of each dependency between quality attributes. We assigned the distribution of the votes to each edge in the ADG as weights to be able to perform the aggregation algorithm (for details see Section 3.1).

With the help of the adapted model, benchmark and votes we calculated the maintainability values of each component. The results of 10 selected components

[3]Here we refer to the source code of a self-compilable *dll* or *exe* as a component.

can be seen in Table 4. The detailed results of the best out of these 10 components is shown in Figure 4. On the left hand side we present the goodness values of low level quality properties (i.e. sensor nodes) of the system. Although our model works with goodness functions we can simply derive a single value by taking the average of the samples. 0 means the worst, while 1 means the best achievable result. On the right side the goodness values of high level quality attributes (i.e. aggregate nodes) are shown. The method level code lines (*LLOC*) got the worst qualification from the sensor nodes. From ISO characteristics *Functionality* got the best score according to our model.

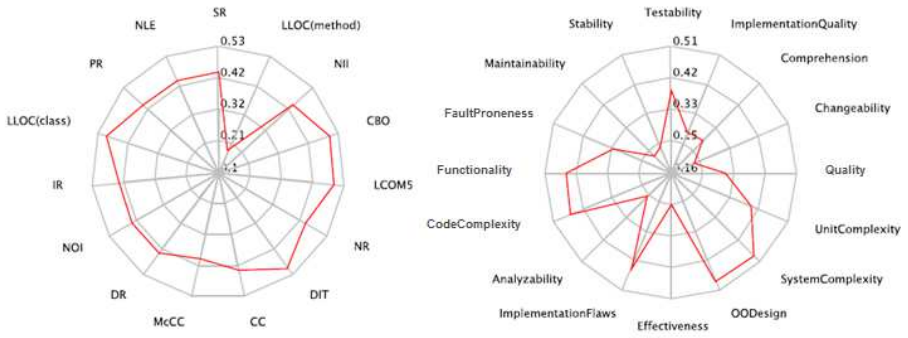

Figure 4: Detailed results of a C# component

The maintainability values of the 10 presented components were manually evaluated by 7 IT professionals of the company. We chose components that each of the IT professionals knew well. So they were able to subjectively assess the maintainability of these components. Every IT professional scored the maintainability of the 10 components on a scale from 0 to 10. 0 means the worst possible maintainability, 10 the best. After collecting all the votes we calculated the average of these votes and divided the value by 10 to convert the value to the [0,1] interval. In this way we could compare the maintainability values provided by our model with the average votes of the IT professionals. Table 4 shows the maintainability values together with the normalized average subjective votes. Although the average human votes are higher than the estimated values the Pearson correlation analysis gave **0.923**, a very high correlation between the two data sets. Since our model does not calculate an absolute maintainability measure, the values cannot be compared directly. However, the correlation analysis showed that our model was able to assess the maintainability of the components relatively to each other which was our goal.

Due to the small sample size and low variance in the values we performed a test where we randomly excluded two out of the ten cases and recalculated the Pearson's and Spearman's rank correlation values. We wanted to rule out that the

high correlation is caused by a few dominant values. Table 3 shows the recalculated correlation values for ten cases.

Table 3: The recalculated correlation values

Pearson's	0.879	0.925	0.879	0.924	0.948
Spearman's	0.896	0.896	0.896	0.854	0.970
Pearson's	0.925	0.924	0.923	0.901	0.777
Spearman's	0.812	0.854	0.916	0.896	0.766

The last column presents the case when the largest and smallest value is removed. Even in this case the correlation values remain fairly high. But in general, we can say that there is not much variance in the original and recalculated values meaning that the high correlation is not caused by some dominant values. Moreover, Spearman's rank correlation is even more stable than Pearson's correlation. It is good because it relies on the ordering of the values and the ordering of the qualifications is the most important in our case study.

Additionally to the subjective voting, we have discussed the results of all these components with the IT professionals in detail. Every extreme sensor values have been justified and for every component we reached a consensual acceptance of the goodness values of high level quality characteristics. Moreover, all the IT professionals agreed with the ranking of the components suggested by the maintainability values.

Table 4: The maintainability values and the average IT professional votes

Maintainability	0.311	0.261	0.261	0.261	0.26
Avg. expert vote	0.56	0.48	0.473	0.53	0.47
Maintainability	0.26	0.221	0.221	0.216	0.178
Avg. expert vote	0.49	0.4	0.44	0.45	0.3

5 Threats to Validity

An obvious threat to the validity of the presented results is the fact that the calculated maintainability values are relative. This means that the presented values reflect the maintainability of the components in comparison to the code base of our industrial partner. What follows from this is that the best component has a maintainability value of 1 and the worst has 0. We cannot place the maintainability of the components on an absolute scale based on these values. However, our primary goal was not to give an absolute measure for maintainability but to be able to rank

and compare the components of our partner according to their maintainability. For this it is enough to have a relative maintainability value because the ranking of the components would be the same using absolute measures. Moreover, it is very easy to adapt our approach to get an absolute measure for maintainability. Only the benchmark database should be extended with many other third party C# systems.

Another major threat is a relatively small number of experts taking part in model declaration and dependency weighting. As we try to estimate a subjective concept a very large number of human opinions would be required to get an objective estimation. However, our experiences show that with the votes of more than 10 persons a very good estimation can be reached. Additionally, it is also very easy to add more expert votes to our model and get more reliable data.

Since it would have been a tiresome task to manually validate the results of all the 315 components we selected 10 of them. So our conclusions about the validity of our maintainability estimations are limited to the evaluated components. Nevertheless, we think that due to the nature of the selection we examined a representative sample of the results and our model indeed approximates the opinions of the IT professionals well.

6 Conclusions and Future Work

In this paper we presented an approach and a concrete model for estimating the maintainability of C# systems. Our primary purpose was to prove the applicability of our previous approach in an industrial environment. Therefore, we adapted our Java specific quality model to assessing the maintainability of C# systems.

This work has been performed in collaboration with the co-workers of our industrial partner, an international software development company. The adaptation of our previous approach included the following steps: 1) introduction of a new maintainability model for C# systems, 2) creation of a benchmark from the C# components of the company, 3) development of a method and tool for qualifying the smaller components of the company's software using the benchmark, 4) introduction of a new weighting involving the developers and managers of the company, and 5) evaluation of a large number of components.

The results were discussed after the evaluation and compared to the developers' opinions. The industrial application of our method and model was successful, as the opinions of the developers highly correlated with the maintainability values produced by our C# maintainability model. This result shows that our probabilistic quality model is applicable in industry, as the industrial partner accepted the provided results and found our approach and tool very useful.

An obvious direction of the future research is to collect more votes to model weights and evaluate more C# systems. The manual inspection of the model based estimations could help improving the qualification process and model.

We also plan to extend the presented approach to different languages. We already have preliminary results on a simple PL/SQL and C maintainability model. Besides these languages there is also a plan to adapt our qualification approach to

C++ and Python languages. After completing the prototype versions of these models we plan to perform a similar industrial case study to that presented here.

References

[1] FxCop Home Page.
`http://msdn.microsoft.com/en-us/library/bb429476(v=vs.80).aspx`.

[2] Baggen, R., Schill, K., and Visser, J. Standardized Code Quality Benchmarking for Improving Software Maintainability. In *Proceedings of the Fourth International Workshop on Software Quality and Maintainability (SQM2010)*, 2010.

[3] Bakota, T., Hegedűs, P., Körtvélyesi, P., Ferenc, R., and Gyimóthy, T. A Probabilistic Software Quality Model. In *Proceedings of the 27th IEEE International Conference on Software Maintenance*, ICSM 2011, pages 368–377, Williamsburg, VA, USA, 2011. IEEE Computer Society.

[4] Bansiya, J. and Davis, C.G. A Hierarchical Model for Object-Oriented Design Quality Assessment. *IEEE Transactions on Software Engineering*, 28:4–17, 2002.

[5] Carvallo, J. P. and Franch, X. Extending the ISO/IEC 9126-1 Quality Model with Non-technical Factors for COTS Components Selection. In *Proceedings of the 2006 international workshop on Software quality*, WoSQ '06, pages 9–14, New York, NY, USA, 2006. ACM.

[6] Chua, B.B. and Dyson, L.E. Applying the ISO9126 Model to the Evaluation of an E-learning System. In *Beyond the comfort zone: Proceedings of the 21st ASCILITE Conference*, pages 184–190, Perth, Australia, 2004. Citeseer.

[7] Correia, J. P., Kanellopoulos, Y., and Visser, J. A Survey-based Study of the Mapping of System Properties to ISO/IEC 9126 Maintainability Characteristics. *IEEE International Conference on Software Maintenance*, pages 61–70, 2009.

[8] Ferenc, R., Beszédes, Á., Tarkiainen, M., and Gyimóthy, T. Columbus – Reverse Engineering Tool and Schema for C++. In *Proceedings of the 18th International Conference on Software Maintenance (ICSM 2002)*, pages 172–181. IEEE Computer Society, October 2002.

[9] Gatrell, M. and Counsell, S. Size, Inheritance, Change and Fault-proneness in C# Software. *Journal of Object Technology*, 9(5):29–54, 2010.

[10] Gatrell, M., Counsell, S., and Hall, T. Empirical Support for Two Refactoring Studies Using Commercial C# Software. In *Proceedings of the 13th international conference on Evaluation and Assessment in Software Engineering*, EASE'09, pages 1–10, Swinton, UK, 2009. British Computer Society.

[11] Goldschmidt, T., Reussner, R., and Winzen, J. A Case Study Evaluation of Maintainability and Performance of Persistency Techniques. In *Proceedings of the 30th international conference on Software engineering*, ICSE '08, pages 401–410, New York, NY, USA, 2008. ACM.

[12] Heitlager, I., Kuipers, T., and Visser, J. A Practical Model for Measuring Maintainability. *Proceedings of the 6th International Conference on Quality of Information and Communications Technology*, pages 30–39, 2007.

[13] ISO/IEC. *ISO/IEC 9126. Software Engineering – Product quality.* ISO/IEC, 2001.

[14] Kanellopoulos, Y., Dimopulos, T., Tjortjis, C., and Makris, C. Mining Source Code Elements for Comprehending Object-oriented Systems and Evaluating Their Maintainability. *SIGKDD Exploration Newsletter*, 8(1):33–40, June 2006.

[15] Letouzey, J. L. The SQALE Method for Evaluating Technical Debt. In *Third International Workshop on Managing Technical Debt (MTD)*, pages 31–36. IEEE, June 2012.

[16] Muthanna, S., Ponnambalam, K., Kontogiannis, K., and Stacey, B. A Maintainability Model for Industrial Software Systems Using Design Level Metrics. In *Proceedings of the Seventh Working Conference on Reverse Engineering (WCRE'00)*, WCRE '00, pages 248–256, Washington, DC, USA, 2000. IEEE Computer Society.

[17] Oman, P. and Hagemeister, J. Metrics for Assessing a Software System's Maintainability. In *Proceerdings of the Conference on Software Maintenance*, volume 19, pages 337–344. IEEE Computer Society Press, 1992.

[18] Plösch, R., Gruber, H., Körner, C., Pomberger, G., and Schiffer, S. A Proposal for a Quality Model Based on a Technical Topic Classification. In *Proceedings of SQMB 2009 Workshop*, 2009.

[19] Sartori, V., Eshete, B., and Villafiorita, A. Measuring the Impact of Different Metrics on Software Quality: a Case Study in the Open Source Domain. In *Proceedings of the Fifth International Conference on Digital Society*, 2011.