Advances in Software Product Quality Measurement and its Applications in Software Evolution

Péter Hegedűs

Department of Software Engineering
University of Szeged

Szeged, 2014

Supervisor:

Dr. Rudolf Ferenc

Summary of the Ph.D. thesis submitted for the degree of Doctor of Philosophy

of the University of Szeged

Veritas
Virtus
Libertas

University of Szeged

PhD School in Computer Science

Introduction

The growing dependence on software systems (e.g. flight control systems and software systems in nuclear facilities) has helped make the areas of software quality and reliability vital for research. Unfortunately, software quality is such a complex and subjective concept that systematically exploring and modeling it is quite hard.

In this thesis, we focus on the maintainability aspect of software quality. According to the definition of the ISO/IEC 9126 standard [15] (superseded by ISO/IEC 25010 [16]) for software quality, maintainability is "the capability of the software product to be modified". Based on this definition it is clear that maintainability has a close connection with the cost of altering the behavior of a software system and it is closely related to the source code of the system. As such, it is a good indicator of "software health" (software integrity) and it is also related to the probability of introducing errors into the source code; so we can think of it as the *technical quality* of a software system. Hence maintainability has become a central issue in modern software industry, and lots of recommendations and counter proposals exist on how to write or modify programs to achieve better maintainability (e.g. design patterns [18], anti-patterns [1] and refactoring techniques [7]).

Nevertheless, in the software industry maintainability is often overshadowed by feature developments (adding functionality), whose business value is more evident – at least in the short term. Because applying techniques that improve the maintainability of the code or avoid structures that degrade systems has an additional cost without having a short-term financial benefit, they are often neglected by the business stakeholders. By better understanding the relation between different coding practices and maintainability (and its effect on the long-term development cost), it should be possible to show a return on investments by applying these techniques and making them more appealing to the business stakeholders as well. In addition, we should (i) ensure that software developers who will perform the maintenance tasks get sufficiently technical, low-level guidelines on how to effectively improve the overall maintainability of a system; (ii) demonstrate that the extra effort they put into increasing maintainability has a noticeable, beneficial effect (e.g. they have fewer bugs after the software release or they can perform developments in the future quicker).

This dissertation focuses on solving the problems outlined above. The three main results of the thesis are the following:

I. **We provide a high-level measure for maintainability that improves the state-of-the-art methods and gives valuable information even to those who have no technical knowledge (e.g. managers). We validated the approach with a prototype model for the Java language and devised a C# model that we applied in a real industrial setting.**

II. **We elaborate methods to acquire useful low-level information about maintainability at the level of the source code elements that can be used to improve the overall system maintainability or help technical persons perform different software evolution tasks like focusing on testing efforts, guiding code reviews and estimating costs.**

III. **We performed empirical case studies to demonstrate the potential of the novel maintainability modeling methods in bug localization, future development cost estimation and in revealing the connection between coding practices (like design patterns) and software maintainability, aided by a general benchmark for reverse engineering tools.**

I System Level Software Quality Models

The contributions of this thesis point are related to software product quality measurement at the system level.

A Probabilistic Maintainability Model and its Validation

To eliminate the common shortcomings of the existing maintainability models indicated in a survey [5], we provide a probabilistic approach [4] for computing high-level quality characteristics defined by the ISO/IEC 9126 [15] and ISO/IEC 25010 [16] standards, which integrates expert knowledge, and handles ambiguity issues at the same time. This method applies so-called "goodness" functions, which are continuous generalizations of threshold-based approaches. 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). The probabilistic statistical aggregation algorithm uses a benchmark as the basis of the qualification, which is a source code metric repository database with 100 open source and industrial software systems. Examining two Java systems with the novel probabilistic quality model, we learned that the changes in the results of the model reflect the development activities, i.e. during development the quality usually decreases, while during maintenance (e.g. performing refactoring activities) the quality usually increases. We also found that the goodness values computed by the model display relatively high correlation values with the expert votes. Table 1 presents the averages of the developers' ranks for each version of both software systems along with the model-based values in brackets. As can be seen, they display a relatively high correlation with each other, meaning that they vary in a similar way.

Version	Changeability	Stability	Analyzability	Testability	Maintainability
REM v0.1	0.625 (0.7494)	0.4 (0.7249)	0.675 (0.7323)	0.825 (0.7409)	0.625 (0.7520)
REM v1.0	0.6 (0.7542)	0.65 (0.7427)	0.75 (0.7517)	0.8 (0.7063)	0.75 (0.7539)
REM v1.1	0.6 (0.7533)	0.66 (0.7445)	0.7 (0.7419)	0.66 (0.6954)	0.633 (0.7402)
REM v1.2	0.65 (0.7677)	0.65 (0.7543)	0.8 (0.7480)	0.775 (0.7059)	0.7 (0.7482)
Correlation	**0.71**	**0.9**	**0.81**	**0.74**	**0.53**
System-1 v1.3	0.48 (0.4458)	0.33 (0.4535)	0.35 (0.4382)	0.43 (0.4627)	0.55 (0.4526)
System-1 v1.4	0.6 (0.4556)	0.55 (0.4602)	0.52 (0.4482)	0.4 (0.4235)	0.533 (0.4484)
System-1 v1.5	0.64 (0.4792)	0.64 (0.4966)	0.56 (0.4578)	0.46 (0.4511)	0.716 (0.4542)
Correlation	**0.87**	**0.81**	**0.94**	**0.61**	**0.77**

Table 1: Averaged grades for maintainability and its attributes based on the developers' opinions

A Maintainability Model for C#

Besides Java, we also devised a maintainability model for C# [10] in collaboration with one of our industrial partners, whose staff were very pleased with the results achieved. The model was used to assess the overall maintainability of over 300 components of the company, and to provide an ordering among them. The newly created model is depicted in Figure 1.
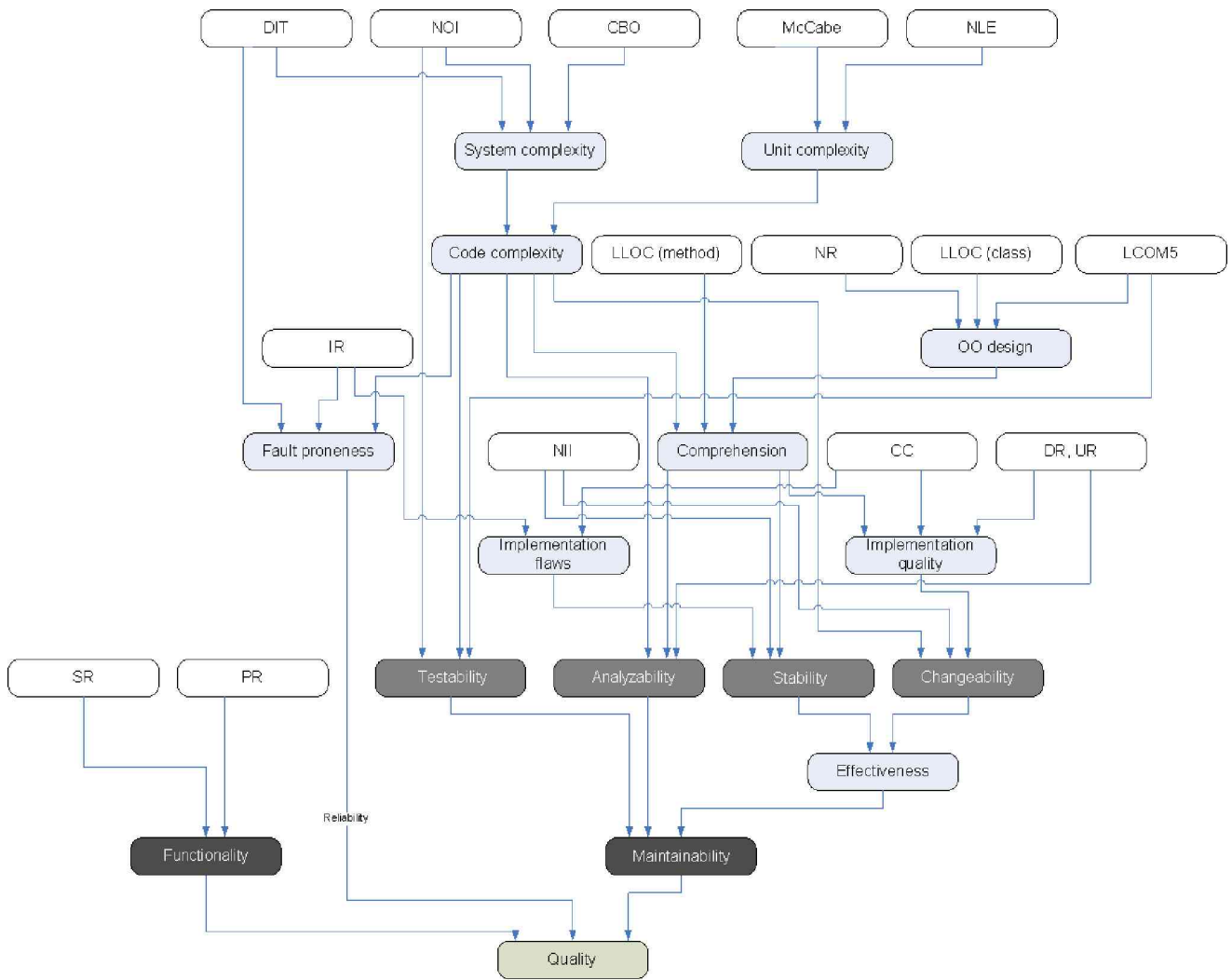
Figure 1: The C# maintainability model (ADG)

We compared the results of our model with the opinions of developers on 10 components and although the average human votes were higher than the estimated values (see Table 2), a Pearson correlation analysis gave a result of 0.92 at a significance level of 0.01, strongly suggesting a high correlation between the two data sets.

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

Table 2: The maintainability values and the average developer votes for 10 components

Implementation and Evaluation of the Approach

The novel probabilistic approach was implemented in a tool named SourceAudit [3] as part of a continuous quality monitoring framework called QualityGate (see a screenshot of its main view in Figure 2). This tool was used in several Hungarian and international R&D projects and it is an official commercial product of FrontEndART Ltd. In addition, we compared and evaluated the tool against other similar tools for software quality assessment purposes [5].
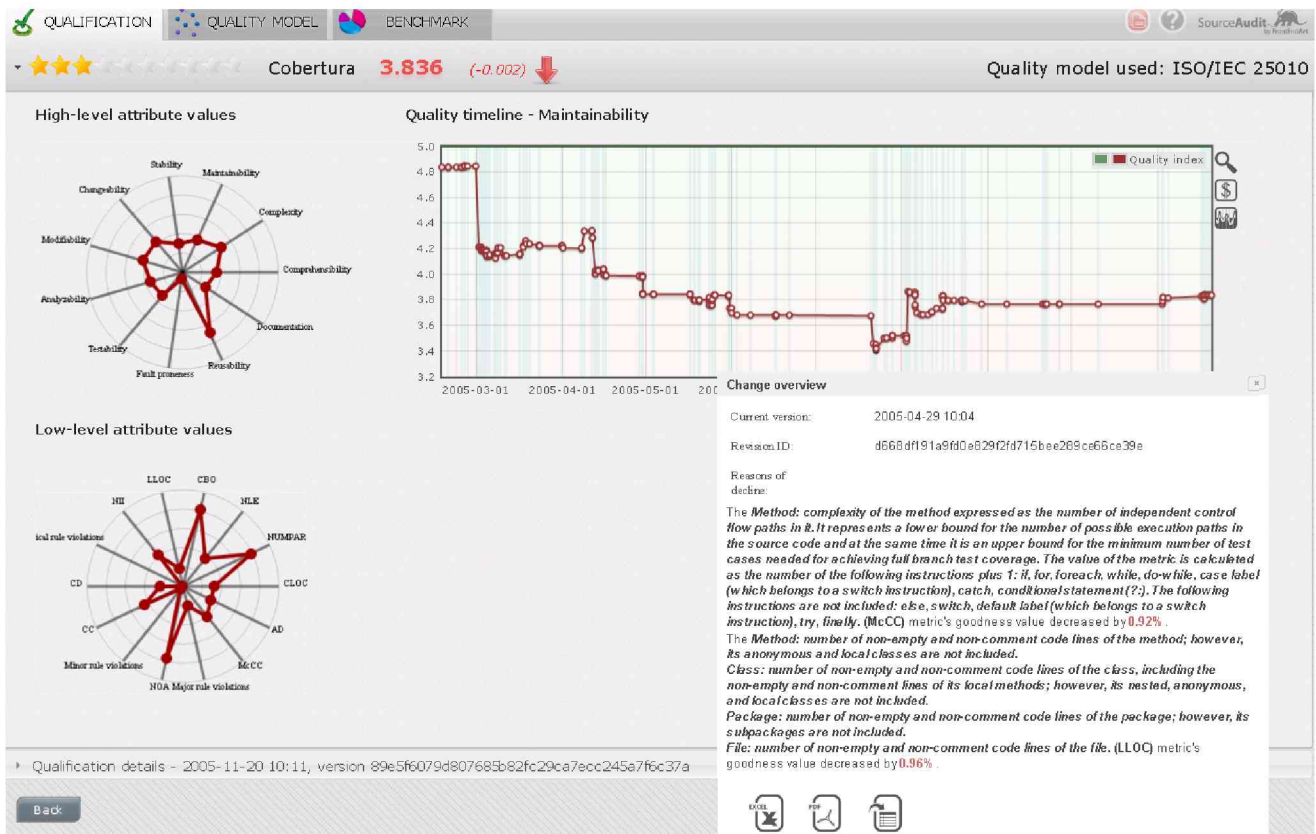
3

Figure 2: The certification details of a system.

Own contributions

The author performed a survey on existing practical models and explored their theoretical background. He performed an empirical validation of a novel probabilistic quality assessment method, evaluated the results and implemented the prototype tools supporting the empirical validation. The entire C# quality model is the author's work; that is, the creation of the C# specific model, the collection of expert weights and benchmark systems required for the quality assessment, the implementation of the necessary tools, carrying out and evaluating the empirical validation of the model. The author took part in designing the SourceAudit tool that implements the above approach. He performed and evaluated the case study of software quality tools.

II   Source Code Element Level Software Quality Modeling

The contributions of this thesis point are related to software product quality measurement at the source code element level.

Preparatory Case Studies

We performed three large case studies [12, 14] to examine the feasibility of predicting software maintainability at the source code element level, based on software product metrics. For this, we collected a large number of subjective opinions on the quality characteristics of different source code elements from IT experts and students with various degrees of expertise. The quality characteristics were those defined in the ISO/IEC 9126 standard and the evaluators rated the characteristics of many source code elements on a scale from 0 to 10 (0 being the worst, 10 being the best). Using the average votes

of the evaluators, we were able to build prediction models based on machine learning techniques using source code metrics as predictors to predict the subjective opinions of humans on the various quality attributes of a software system. We found that metrics had the potential to predict high-level quality indicators assessed by humans (the votes of the evaluators displayed a deviation of between 0.5 and 2 on a scale of 10). As can be seen in Table 3, with the *Changeability* property, the decision tree-based classifier had a precision of nearly 77% (here we grouped the evaluations into three categories).

	ZeroR	J48 Decision Tree	Log. Regression	Neural Network
Analyzability	67.93%	73.68%	70.97%	70.25%
Changeability	66.79%	76.65%	73.00%	74.26%
Stability	70.20%	73.12%	70.55%	70.92%
Testability	66.55%	64.72%	69.45%	70.54%

Table 3: Rate of the correctly classified instances

After reviewing the different regression techniques available, we can say that they are even more appropriate for building prediction models using a continuous scale instead of classes than the standard classifier methods. The best regression model trained on our evaluation data predicted *Maintainability* with a correlation of 0.72 and mean average error (MAE) of 0.83. Table 4 shows the performance of the various regression techniques.

	ZeroR		Neural Network		Linear Reg.		Decision Tree	
	MAE	**Corr.**	**MAE**	**Corr.**	**MAE**	**Corr.**	**MAE**	**Corr.**
Analyzability	1.201	-0.162	1.076	0.408	1.076	0.466	0.884	0.660
Changeability	1.026	-0.116	1.088	0.362	0.965	0.437	0.861	0.571
Comprehens.	1.574	-0.153	1.387	0.275	1.188	0.491	1.048	0.621
Stability	0.822	-0.239	0.824	0.297	0.833	0.360	0.670	0.572
Testability	1.189	-0.118	1.168	0.427	1.145	0.363	0.926	0.639
Maintainability	1.187	-0.122	1.193	0.587	0.909	0.615	0.831	0.723
Average	**1.166**	**-0.152**	**1.123**	**0.393**	**1.019**	**0.455**	**0.870**	**0.631**

Table 4: The MAE and correlation values of the regression techniques examined

A Drill-down Approach to Derive a Source Code Element Level Maintainability Measure

Based on the lessons learned from our empirical studies, we propose a novel method for drilling down to the root causes of a quality rating [11] and giving a relative maintainability index (RMI) of individual source code elements (e.g. classes, methods) that in contrast to current approaches [5], takes the combinations of different metrics into account. This allows us to rank source code elements in such a way that the most critical elements are at the top of the list; and this should allow the system maintainers to utilize their resources better and achieve a maximal improvement in the source code with minimal investment. We validated the approach by comparing the model-based maintainability ranking with the manual ranking of 191 Java methods of the jEdit open source text editor tool. The manual maintainability evaluation of the methods performed by some 200 students displayed a Spearman correlation of 0.68 ($p < 0.001$) with the model-based evaluation. The results of the correlation analysis are presented in Table 5. The drill-down algorithm was later included in the SourceAudit [3] commercial quality monitoring tool mentioned above.

Quality attribute	Correlation with students' opinions (R value)	p-value
Analyzability	0.64	< 0.001
Comprehensibility	0.62	< 0.001
Changeability	0.49	< 0.001
Stability	0.49	< 0.001
Testability	0.61	< 0.001
Maintainability	**0.68**	$<\textbf{0.001}$

Table 5: Spearman's correlation between the RMI-based and the manual evaluations

Own contributions

The author devised the preliminary case study concepts, the survey questions and the basic principles of a Web-based metric evaluation framework. He evaluated and compared the survey data and the results of the machine learning algorithms and drew some key conclusions. He established the theoretical basis for the drill-down approach, elaborated the validation method of the approach and performed a validation on open-source systems and then interpreted and evaluated the results.

III   Applications of the Proposed Quality Models

The contributions of this thesis point are related to the utilization of the proposed techniques, models and tools described above.
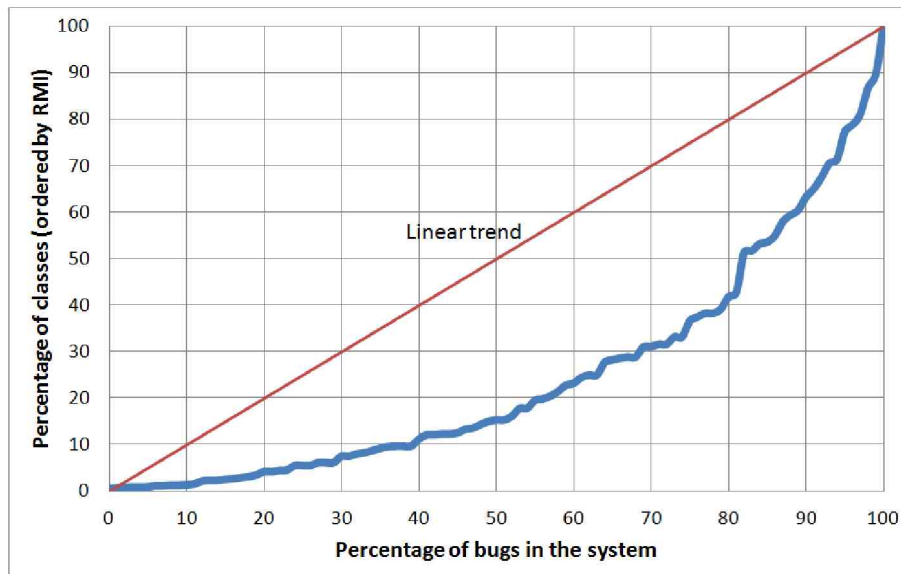

Figure 3: Various bug coverage rates with RMI-based ordering

Bug Localization Capability of the Drill-down Method

From a statistical analysis we demonstrated that the relative maintainability measure is effective in separating fault-prone classes (i.e. classes containing lots of bugs) from classes which are unlikely to have faults (i.e. bugs) in them [17]. Our case study on 30 releases of different open-source systems revealed that on average 30% of classes with the worst maintainability index contained over 70% of the total bugs. The various bug coverage rates have been plotted as a function of the percentage of

the RMI-based class ordering in Figure 3. These results suggest that ranking the classes based on their maintainability is a very good strategy for focusing testing efforts or guiding code review.

A Maintainability-based Cost Model and its Validation

We also proposed a cost model [2] that is able to predict future development effort based on the maintainability change of a system. Using some simple assumptions and adopting the concept of entropy from thermodynamics, we were able to show that the maintainability of a system decreases exponentially with the invested development effort if intentional code improvement actions are not performed. We made use of the revealed connection between maintainability and cost to assess the future development cost of two open-source and three proprietary systems. The results of the assessment are shown in Figure 4. The left-hand side diagrams show both the real cost (dark lines) and the cost function predicted by the model (light lines). On the right hand side, the dark lines represent the changes in maintainability, while the light lines show the predicted maintainability. The diagrams also show Pearson correlations between the real and the predicted curves. The high correlations indicate that both cost and maintainability are nicely described by the same model.

Investigating the Effect of Design Pattern Usage on Maintainability

Our proposed quality model can also be utilized to learn the concrete connection between maintainability and coding practices (e.g. design patterns, anti-patterns, code clones and refactoring techniques) that are considered to have a positive or negative impact on maintainability. In particular, the belief that utilizing design patterns will create better quality software system is fairly widespread; however, there is relatively little evidence to objectively indicate that their usage is indeed beneficial. In fact; some studies found that the use of design patterns can be quite risky [18]. As a first step towards empirically investigating the effect of design patterns, we analyzed [13] some 300 revisions of JHotDraw, a Java GUI framework, whose design relies heavily on some well-known design patterns. We found that every pattern instance introduced caused an improvement in the different quality attributes for JHotDraw, as shown in Table 6. Moreover, the average design pattern line density displayed a high, Pearson correlation of 0.89 with the estimated maintainability at a significance level of 0.05.

Revision (r)	Pattern	Pattern Line Density	Maintain-ability	Test-ability	Analyz-ability	Stability	Change-ability
531	+3	↗	↗	↗	↗	↗	↗
574	+1	↗	↗	↗	↗	↗	↗
609	−1	↘	—	—	—	—	—
716	+1	↘	↗	↗	↗	↗	↗
758	+1	↗	↗	↗	↗	↗	↗

Table 6: Software quality attribute tendencies in the case of design pattern changes

To verify our initial findings, we repeated the study on 9 different open source systems using the design pattern results of 5 different tools available in the DPB [6] online benchmark. The pattern densities displayed a similarly high Pearson correlation (between 0.59 and 0.78) and Spearman correlation (between 0.68 and 0.82) with software maintainability at a significance level of 0.05. Filtering out false positive pattern instances based on the evaluations in the benchmark, we were able to improve the correlation values by about 10%.
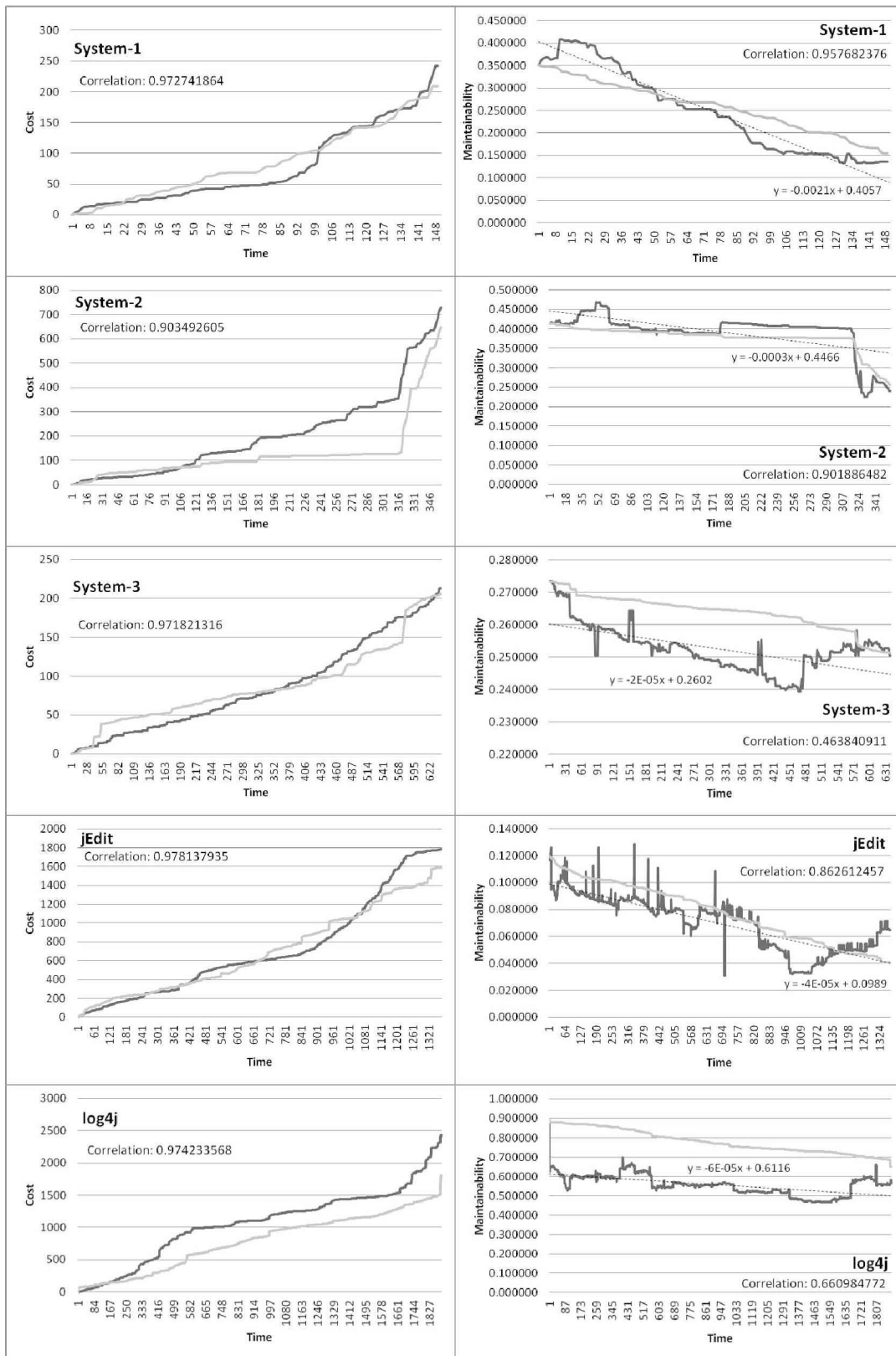
7

Figure 4: Estimated and real costs and maintainability as a function of time

A Benchmark for Reverse Engineering Tools

To support our long-term goal of empirically investigating the effect of other coding practices and patterns on software maintainability, we propose to make use of our benchmark called BEFRIEND (BEnchmark For Reverse engInEering tools workiNg on source coDe) [9], which processes, evaluates and compares the outputs of reverse engineering tools. It may be viewed as a generalization of our DEEBEE design pattern benchmark [8]. With the help of BEFRIEND, we can collect a large amount of precise input data to examine the effect of different coding practices and patterns on maintainability.

Aspect	Mean	Deviation	Min	Max	Median
#32	66.0%	0.0%	66.0%	66.0%	66.0%
#33	83.0%	24.04%	66.0%	100.0%	83.0%
#34	33.0%	46.67%	0.0%	66.0%	33.0%
#40	83.0%	24.04%	66.0%	100.0%	83.0%
#43	16.5%	23.33%	0.0%	33.0%	16.5%
#44	33.0%	46.67%	0.0%	66.0%	33.0%
Mean	**52.1%**	**13.39%**	**42.63%**	**67.88%**	**52.1%**
Deviation	**26.92%**	**16.69%**	**31.85%**	**18.67%**	**26.92%**
Min	**0.0%**	**0.0%**	**0.0%**	**33.0%**	**0.0%**
Max	**100.0%**	**46.67%**	**100.0%**	**100.0%**	**100.0%**
Median	**66.0%**	**0.0%**	**66.0%**	**66.0%**	**66.0%**

Summary

Number of instances:	43
Number of evaluated instances:	43
Number of instances above the threshold:	27
Precision:	62.79%
Total number of instances:	56
Total number of evaluated instances:	56
Total number of instances above the threshold:	32
Recall:	84.38%

Figure 5: Bauhaus clone detector tool correctness statistics

BEFRIEND provides several views for collecting the user evaluations and it can also analyze the evaluation results. For example, in the statistics view the user is provided with statistics based on the evaluation criteria and the user votes obtained earlier (see Figure 5).

Own contributions

The author elaborated the statistical methods used to analyze the bug localization capability of the drill-down approach. He applied these procedures, evaluated and presented the results. He performed the empirical validation of the cost model, implemented the prototype tools supporting the validation, analyzed and evaluated the results. He developed the approach to reveal the connection between design pattern utilization and the maintainability of a software system. An analysis of the subsequent revisions of JHotDraw and the systems in the different benchmarks, and also an evaluation of the empirical results were also his work. Except for the generalized sibling algorithm, he implemented and presented BEFRIEND, which is now a general benchmark for evaluating reverse engineering tools.

Summary

The contributions are grouped into three major thesis points. First, we examined the current practical approaches of software maintainability modeling and proposed a novel probabilistic approach that eliminates most of the shortcomings of the existing approaches by taking the subjective opinions of experts into account and by applying a benchmark of other systems as the baseline of maintainability assessment. These results could be of great help for managers and other non-technical personnel to get an overall picture of the maintainability of their system. Based on the prototype model for Java, we established a C# maintainability model that was successfully applied in an industrial environment. The model results reflected the expert opinions to a great extent. To help apply the new research results in practice, we also provided a full implementation of our approach that can be downloaded.

To provide lower-level technical information to developers who wish to improve the maintainability of a system based on the model results, we introduced a so-called drill-down approach with which maintainability measures can be derived for each individual source code element, and not just for the system as a whole. With this measure one can pinpoint those problematic source code elements that should be corrected quickly in order to achieve a substantial increase in system-level maintainability. Moreover, this low-level information can help in identifying fault-prone hot-spots, focusing testing efforts and guiding code reviews. Besides the theoretical results, we presented practical applications of the proposed maintainability measures in software evolution. With the help of empirical case studies, we demonstrated that the maintainability measure of classes is a good indicator of fault-proneness. We also introduced a cost model that is based on the maintainability changes of a system and it is able to predict the effort required for future developments.

Next, we presented case studies where we analyzed the relationship between maintainability and design pattern utilization in the source code because applying design patterns is thought to be a good coding practice for attaining high maintainability. We found that design pattern density and technical quality of the source code are indeed closely related. These findings are only the first step towards empirically investigating common beliefs concerning the effect of different coding practices (e.g. design patterns, ant-patterns and refactoring) on software maintainability that is needed to convince the industrial stakeholders on a return on investment in maintainability.

Here, we also summarize the main publications related to the various thesis points in Table 7.

$\mathcal{N}o.$	[4]	[10]	[3]	[5]	[11]	[12]	[14]	[13]	[2]	[17]	[8]	[9]
I	●	●	●	●								
II			●	●	●	●	●					
III								●	●	●	●	●

Table 7: Thesis contributions and supporting publications

Acknowledgements

References

[1] *Marwen Abbes, Foutse Khomh, Yann-Gaël Guéhéneuc, and Giuliano Antoniol. An Empirical Study of the Impact of Two Antipatterns, Blob and Spaghetti Code, on Program Comprehension.* In Proceedings of the 15th European Conference on Software Maintenance and Reengineering (CSMR), *pages 181–190. IEEE, 2011.*

[2] *Tibor Bakota, Péter Hegedűs, Gergely Ladányi, Péter Körtvélyesi, Rudolf Ferenc, and Tibor Gyimóthy. A Cost Model Based on Software Maintainability.* In Proceedings of the 28th IEEE International Conference on Software Maintenance (ICSM 2012), *pages 316–325, 2012.*

[3] *Tibor Bakota, Péter Hegedűs, István Siket, Gergely Ladányi, and Rudolf Ferenc. QualityGate SourceAudit: a Tool for Assessing the Technical Quality of Software.* In 2014 Software Evolution Week-IEEE Conference on Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE), *pages 440–445. IEEE, 2014.*

[4] *Tibor Bakota, Péter Hegedűs, Péter Körtvélyesi, Rudolf Ferenc, and Tibor Gyimóthy. 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.*

[5] *Rudolf Ferenc, Péter Hegedűs, and Tibor Gyimóthy. Software Product Quality Models.* In *Tom Mens, Alexander Serebrenik, and Anthony Cleve, editors,* Evolving Software Systems, *pages 65–100. Springer Berlin Heidelberg, 2014.*

[6] *Francesca Arcelli Fontana, Andrea Caracciolo, and Marco Zanoni. DPB: A Benchmark for Design Pattern Detection Tools.* In Proceedings of the 16th European Conference on Software Maintenance and Reengineering (CSMR), *pages 235–244, 2012.*

[7] Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. *Refactoring: Improving the Design of Existing Code*. *Addison-Wesley, 1999.*

[8] Lajos Jenő Fülöp, Árpád Ilia, Ádám Zoltán Végh, Péter Hegedűs, and Rudolf Ferenc. *Comparing and Evaluating Design Pattern Miner Tools.* ANNALES UNIVERSITATIS SCIENTIARUM DE ROLANDO EÖTVÖS NOMINATAE Sectio Computatorica, *XXXI:167–184, 2009.*

[9] Lajos Jenő Fülöp, Péter Hegedűs, and Rudolf Ferenc. *BEFRIEND – a Benchmark for Evaluating Reverse Engineering Tools.* Periodica Polytechnica Electrical Engineering, *52(3-4):153–162, 2008.*

[10] Péter Hegedűs. *A Probabilistic Quality Model for C# – an Industrial Case Study.* Acta Cybernetica, *21(1):135–147, 2013.*

[11] Péter Hegedűs, Tibor Bakota, Gergely Ladányi, Csaba Faragó, and Rudolf Ferenc. *A Drill-Down Approach for Measuring Maintainability at Source Code Element Level.* Electronic Communications of the EASST, *60:1–21, 2013.*

[12] Péter Hegedűs, Tibor Bakota, László Illés, Gergely Ladányi, Rudolf Ferenc, and Tibor Gyimóthy. *Source Code Metrics and Maintainability: a Case Study. In* Proceedings of the 2011 International Conference on Advanced Software Engineering & Its Applications (ASEA 2011), *pages 272–284. Springer-Verlag CCIS, 2011.*

[13] Péter Hegedűs, Dénes Bán, Rudolf Ferenc, and Tibor Gyimóthy. *Myth or Reality? Analyzing the Effect of Design Patterns on Software Maintainability. In* Proceedings of the 2012 International Conference on Advanced Software Engineering & Its Applications (ASEA 2012), *pages 138–145. Springer-Verlag CCIS, 2012.*

[14] Péter Hegedűs, Gergely Ladányi, István Siket, and Rudolf Ferenc. *Towards Building Method Level Maintainability Models Based on Expert Evaluations. In* Computer Applications for Software Engineering, Disaster Recovery, and Business Continuity, *pages 146–154. Springer, 2012.*

[15] ISO/IEC. *ISO/IEC 9126. Software Engineering – Product quality 6.5. ISO/IEC, 2001.*

[16] ISO/IEC. *ISO/IEC 25000:2005. Software Engineering – Software product Quality Requirements and Evaluation (SQuaRE) – Guide to SQuaRE. ISO/IEC, 2005.*

[17] Gergely Ladányi, Péter Hegedűs, Rudolf Ferenc, István Siket, and Tibor Gyimóthy. *The Connection of the Bug Density and Maintainability of Classes. In* 8th International Workshop on Software Quality and Maintainability, *SQM, 2014 (presentation only). http://sqm2014.sig.eu/?page=program.*

[18] William B. McNatt and James M. Bieman. *Coupling of Design Patterns: Common Practices and Their Benefits. In* Proceedings of the 25th International Computer Software and Applications Conference on Invigorating Software Development, *COMPSAC '01, pages 574–579, Washington, DC, USA, 2001. IEEE Computer Society.*