New Datasets for Bug Prediction and a Method for Measuring Maintainability of Legacy Software Systems

Zoltán Tóth

Department of Software Engineering
University of Szeged

Szeged, 2019

Supervisor:

Dr. Rudolf Ferenc

Summary of the Ph.D. Thesis

University of Szeged
Ph.D. School in Computer Science

Introduction

Software systems have become more complex over the years. We developed additional abstraction levels to leverage the complexity developers have to face. We are now using high level programming languages to ease the writing of such complex tasks. Since humans are involved as the most important factor in the software development process, software will have faults. In the beginning of the 21st century, software bugs cost the United States economy approximately $60 billion a year. In 2016, that number is $1.1 trillion. The industrial sector often gives up on software quality due to time pressure. A great example is the Boeing scandal in 2019. The MCAS system that automatically adjusts the plane's flight trajectory was faulty in its software. It was more than just a software bug, pilots were not educated as they should had been to handle this fault in the system. Still, the most crucial factor was the software bug itself, which caused the death of 356 innocents.

The later the phase a bug is fixed in, the more it costs. Eliminating defects in an early stage is the ideal scenario. Testing could serve as one of the best tools for this purpose. However, testing should involve automated checks, where it is possible to reveal defective candidates early. The research work behind this thesis aims to help in locating future defects with the help of bugs fixed in the past, moreover, to provide methodology for measuring the maintainability (which is one of the most important quality characteristics) of RPG legacy software systems.

In this respect, the thesis encapsulates two main topics: *the construction and evaluation of new bug datasets* and *introducing a methodology for measuring maintainability of RPG software systems*. These topics both emphasize the importance of software quality and try to give state-of-the-art solutions in the fight against software faults.

Public bug datasets have been present for a long time. In spite of the fact that bug prediction related studies have been rapidly growing in recent years, there are only a few datasets available. Bug datasets are usually constructed from open-source projects that managed to store issues with the appropriate issue tracking methods. This way, the source code elements of the systems could be mapped with the corresponding bug(s) [10]. Datasets are usually constructed at file or class level. The entries of the datasets are characterized somehow, in order to describe the bugs. One possible way is to calculate static software product metrics for each entry. Currently, open-source projects are hosted on the popular GitHub platform. We created a new, state-of-the-art public bug dataset, the GitHub bug dataset, which includes a wide range of static source code metrics (more than 50) to characterize the bugs. Moreover, we also gathered all existing bug datasets and built a unified dataset on top of them. During this process, we pointed out the inconsistencies in the datasets, and we also showed how different results could be obtained by using another tool for the static analysis. We demonstrated the power of the built datasets by showing their capabilities in bug prediction.

Measuring maintainability in RPG systems is a narrowed research area, which also applies generally in the domain of legacy systems. A large amount of the systems used in the banking sector still run on IBM mainframes, hence they use the RPG programming language as well. Finding and eliminating bugs in these softwares could be a matter of national interest. Contrarily, research trends not to reflect the importance of providing novel techniques for monitoring the quality of such legacy systems. However, the industry could adapt different methodologies and solutions from other domains, since usually there is a lack of underlying tools to support the analysis of legacy systems. We did not only develop a methodology for measuring the maintainability, but also provided the appropriate tools needed to overcome these barriers.

The thesis consists of two main parts, which are also the two thesis points. In this booklet, we summarize the results of each thesis point in the corresponding parts.

I New Datasets and a Method for Creating Bug Prediction Models for Java

The contributions of this thesis point are related to public bug datasets for Java systems and their extension to provide a powerful and strong foundation for building bug prediction models.

A New Public Bug Dataset and Its Evaluation in Bug Prediction

The focus of this research area was to investigate the possible imperfections of the existing public bug datasets (which use static source code metrics to characterize the entries in the datasets), and to create a new bug dataset that solves these imperfections. In spite of the fact that the trend of hosting open-source projects points in the direction of GitHub, none of the existing datasets used it as the source of information. Available bug datasets are quite old, hence the systems included are aged as well. Based on our findings, we constructed a new dataset for 15 projects selected from GitHub containing more than 3.5 million lines of code, more than 114,000 commits, and more than 8,700 closed bug reports in total. We collected bug data in an automatic way and created 6-month-long time intervals for every project and accumulated bug information for these chosen releases (105 both at file and class level). Contrary to previous bug datasets, we aggregated bugs for the preceding release versions not for the succeeding ones. Besides the bug dataset being our main contribution, we investigated two research questions:

RQ 1: *Is the constructed database usable for bug prediction? Which algorithms or algorithm families perform the best in bug prediction using our newly created bug dataset?*

RQ 2: *Which machine learning algorithms or algorithm families perform the best in bug coverage?*

Table 1: F-measures at class level

Project	**SGD**	**Simple Logistic**	**SMO**	**PART**	**Random Forest**
Android Universal I. L.	0.6258	0.5794	0.5435	0.6188	0.7474
ANTLR v4	0.7586	0.7234	0.7379	0.7104	0.8066
Broadleaf Commerce	0.8019	0.8084	0.8081	0.7813	**0.8210**
Eclipse p. for Ceylon	0.6891	0.7078	0.6876	0.7283	0.7503
Elasticsearch	0.7197	0.7304	0.7070	0.7171	0.7755
Hazelcast	0.7128	0.7189	0.6965	0.7267	0.7659
jUnit	0.7506	0.7649	0.7560	0.7262	0.7939
MapDB	0.7352	0.7667	0.7332	0.7421	0.7773
mcMMO	0.7192	0.6987	0.7203	0.6958	0.7418
Mission Control T.	0.7819	0.7355	0.7863	0.6862	**0.8161**
Neo4j	0.6911	0.7156	0.6835	0.6731	0.6767
Netty	0.7295	0.7437	0.7066	0.7521	0.7937
OrientDB	0.7485	0.7359	0.7310	0.7194	0.7823
Oryx	0.8012	0.7842	**0.8109**	0.7754	0.8059
Titan	0.7540	0.7558	0.7632	0.7301	0.7830
Avg.	**0.7346**	**0.7312**	**0.7248**	**0.7189**	**0.7758**

For answering RQ 1, we built 13 different prediction models using a machine learning framework called Weka. During our training and testing process, we used 10-fold cross validation with random undersampling to equalize the number of buggy and non-buggy source code elements [4]. We applied random under sampling 10 times and calculated the averages. The results of the top five algorithms can be seen in Table 1. Random Forest seemed to perform the best in this task, which is proven by the high F-Measure values. We achieved a 0.77 average at class level. Similar, but little lower F-measure values were obtained at file level with an average of 0.71. File level results are limited due to the narrow set of source code metrics. These results imply a positive answer to this research question, and we can state that the GitHub Bug Dataset is a good candidate for being the training set in bug prediction.

For answering RQ 2, we reused the previously built 10 models (from the random under sampling cases) and evaluated them on the whole dataset (without random under sampling). During the evaluation, we used majority voting for an element (if more than five models predict the element to be faulty then we tagged it as faulty, otherwise we tagged it as non-faulty). It is important to note that we count a bug covered if at least one corresponding source code element is marked as buggy. Perfect or nearly-perfect bug coverage could be reached by tagging around 31% of the source code elements as buggy in case of RandomTree and RandomForest, which are quite promising results. The same is true for file level as well, however, more entries have to be marked as buggy to achieve perfect bug coverage. If precision is preferred over recall then using Naive Bayes could be a good option.

The GitHub Bug Dataset can be downloaded at: `http://www.inf.u-szeged.hu/~ferenc/papers/GitHubBugDataSet/`

A Unified Public Bug Dataset and Its Assessment in Bug Prediction

During the process of collecting existing public bug datasets, we realized that there is a need to validate the built bug prediction methods on a larger and more general dataset, thus we identified a goal to unite all the public bug datasets into a larger one. Starting with a literature review, we performed an exhaustive search for all available public bug datasets. We collected 5 candidates, which includes the PROMISE Dataset, the Eclipse Bug Dataset, the Bug Prediction Dataset, the Bugcatchers Bug Dataset, and our own GitHub Bug Dataset. We merged these datasets into a grandiose one to fulfil the need for a large, more general dataset. We used an open-source static source code analyzer, named OpenStaticAnalyzer (OSA), to extract more than 50 static source code metrics for all the systems included in the Unified Bug Dataset. This way, we obtained a uniform metric suite for the whole dataset, in which we kept the original bug numbers for each entry (pairing the original entries with the results of OSA was based on standard class names and filenames). We investigated the root causes of the inability to match entries from the original datasets with the results of OSA. One major cause was the presence of entries in the original datasets which are not real Java source files (Scala sources or package-info files). In some cases, we could not find the proper source code for the given system, so two different, but close versions of the same system might be conjugated. We were unable to match 624 class level entries and 28 file level entries out of 48,242 and 43,772 entries, respectively. This means that only 0.71% (652 out of 92,014) of the elements were left out from the unified dataset.

We also pointed out that the metric definitions and the metric namings can severely differ between datasets. Even in the case of Logical Lines of Code, the metric values significantly differed, which is due to using byte code and source code based analyzer tools in different datasets.

We evaluated the datasets according to summary meta data and functional criteria. Summary meta data includes the investigation of the used static analyzer, granularity, bug tracking and version control system, the set of used metrics, etc. As functional criteria, we compared the prediction capabilities of the original metrics, the unified ones, and both together. We used the J48 decision tree algorithm (an implementation of C4.5) from Weka to build and evaluate bug prediction models per projects with 10-fold cross validation in the Unified Bug Dataset. We achieved 0.892 and 0.886 F-measure values at class level using the original and the OSA metrics, respectively. The same insignificant difference could be found at file level, however with lower F-measure scores. Applying both metric suites at the same time makes a small but negligible increase in F-measure at class level, but not at file level. Inconsistencies between the original and OSA metrics could cause lower values. It is important to note, that the generally higher F-measure values come from the fact, that we did not use random undersampling in this case, since we were only interested in the differences of the bug prediction capabilities when we use the original and the OSA metrics.

As an additional functional criterion, we used different software systems for training and for testing the models, also known as cross project training. We performed this step on all the systems of the various datasets. Results achieved on the GitHub Bug Dataset are the most consistent, which is shown in Table 2. The darker the color, the higher the F-measure value is.

Altogether, our experiments showed that the Unified Bug Dataset can be used effectively in bug prediction, achieving higher than 0.8 F-measure values in cross project learning. We encourage researchers to use this large and public unified bug dataset in their experiments and we also welcome new public bug datasets. The Unified Bug Dataset is accessible at:
`http://www.inf.u-szeged.hu/~ferenc/papers/UnifiedBugDataSet`.

The Author's Contributions

The author designed the methodology for extracting bug information from GitHub and the idea behind the construction of the GitHub bug dataset. He performed the literature review in the field of public bug datasets, and collected all relevant datasets and their characteristics. He took part in the process of defining criteria for the projects to be included in the GitHub bug dataset. In case of the unified bug dataset, the author constructed all statistics for the gathered datasets and projects, moreover, performing static source code analysis on the subject systems is also the author's work. Collecting and comparing the metric suites, as well as gathering the summary meta data on datasets are the author's own results. The author also participated in the building of bug prediction models for the GitHub bug dataset and for the unified bug dataset as well. The author formed the final machine learning results for both datasets, and the conclusions were drawn by him.

- **Zoltán Tóth**, Péter Gyimesi, and Rudolf Ferenc. A Public Bug Database of Github Projects and Its Application in Bug Prediction. In Proceedings of the 16th International Conference on Computational Science and Its Applications (ICCSA 2016), Beijing, China. Pages 625-638, Published in Lecture Notes in Computer Science (LNCS), Volume 9789, Springer-Verlag. July, 2016.

- Rudolf Ferenc, **Zoltán Tóth**, Gergely Ladányi, István Siket, and Tibor Gyimóthy. A Public Unified Bug Dataset for Java. In Proceedings of the 14th International Conference on Predictive Models and Data Analytics in Software Engineering, PROMISE'18. Oulu, Finland. Pages 12–21, ACM. October, 2018.

Table 2: Cross training (GitHub – Class level)

Train/Test	Android Universal I. L.	ANTLR v4	Broadleaf Commerce	Eclipse p. for Ceylon	Elasticsearch	Hazelcast	jUnit	MapDB	mcMMO	Mission Control T.	Neo4j	Netty	OrientDB	Oryx	Titan
Android Universal I. L.	0.943	0.885	0.772	0.822	0.813	0.821	0.827	0.827	0.825	0.836	0.868	0.859	0.852	0.850	0.853
ANTLR v4	0.611	0.929	0.785	0.852	0.843	0.842	0.847	0.847	0.845	0.862	0.893	0.882	0.876	0.874	0.876
Broadleaf Commerce	0.635	0.877	0.934	0.928	0.870	0.859	0.863	0.862	0.860	0.873	0.894	0.886	0.880	0.879	0.879
Eclipse p. for Ceylon	0.611	0.894	0.781	0.867	0.847	0.847	0.851	0.851	0.848	0.864	0.895	0.884	0.879	0.877	0.879
Elasticsearch	0.642	0.868	0.835	0.875	0.922	0.900	0.902	0.900	0.897	0.904	0.914	0.904	0.897	0.895	0.895
Hazelcast	0.635	0.876	0.792	0.831	0.828	0.861	0.864	0.863	0.861	0.871	0.888	0.879	0.872	0.871	0.872
jUnit	0.644	0.849	0.774	0.837	0.819	0.813	0.821	0.822	0.821	0.835	0.867	0.858	0.854	0.853	0.854
MapDB	0.670	0.852	0.799	0.855	0.852	0.853	0.857	0.859	0.858	0.871	0.894	0.884	0.879	0.877	0.878
mcMMO	0.642	0.879	0.793	0.855	0.845	0.849	0.853	0.853	0.853	0.866	0.888	0.880	0.874	0.873	0.875
Mission Control T.	0.611	0.890	0.773	0.843	0.836	0.836	0.840	0.840	0.838	0.856	0.890	0.879	0.872	0.870	0.872
Neo4j	0.611	0.890	0.774	0.843	0.836	0.836	0.841	0.840	0.838	0.856	0.890	0.878	0.871	0.869	0.871
Netty	0.644	0.873	0.787	0.848	0.834	0.831	0.837	0.836	0.834	0.847	0.874	0.876	0.869	0.867	0.868
OrientDB	0.611	0.845	0.800	0.857	0.835	0.841	0.846	0.846	0.845	0.859	0.888	0.878	0.884	0.882	0.882
Oryx	0.712	0.874	0.787	0.845	0.837	0.840	0.845	0.845	0.843	0.857	0.879	0.870	0.865	0.866	0.868
Titan	0.611	0.895	0.787	0.849	0.840	0.843	0.847	0.847	0.845	0.859	0.890	0.880	0.874	0.872	0.878

II Methodology for Measuring Maintainability of RPG Software Systems

The contributions of this thesis point are related to measuring maintainability in RPG systems. Giving a solution for analyzing RPG software systems was an industrial need. We performed this research in consortium with R&R Software Ltd. who has a long history in developing RPG software systems. They drew attention to the need for an RPG static source code analyzer and a methodology to properly measure maintainability of RPG systems. The research artifacts described in this thesis point were supported by the Hungarian national grant GOP-1.1.1-11-2012-0323.

Comparison of Static Analysis Tools for Quality Measurement of RPG Programs

The goal of this research was to give an exhaustive comparison about state-of-the-art RPG static source code analyzers. The research is focused on the data obtained using static analysis, which is then aggregated to higher level quality attributes. SourceMeter is a command line toolchain capable of measuring various source attributes like software metrics, coding rule violations, and code clones. This toolchain is of our own development. SonarQube is a quality management platform with RPG language support. To facilitate the objective comparison, we used the SourceMeter for RPG plugin for SonarQube, which seamlessly integrates into the framework, extending its capabilities. This way, the interface of the tools under examination was the same, hence the comparison was easier to perform. We collected 179 RPG source code files to be the input of the comparison. The evaluation is built on analysis success and depth, source code metrics, coding rules and code duplications. SourceMeter could analyze all the systems successfully, while SonarQube was unable to handle 3 source files, because there were unsupported language constructs in the files (e.g. free-form blocks). SourceMeter outputs entries at four levels: *System, Program (File), Procedure*, and *Subroutine*. Contrarily, SonarQube only works with *System* and *File* levels. SonarQube gives a limited set of metrics, while SourceMeter also calculates static source code metrics at a finer granularity (procedure and subroutine levels). There is a large common set of coding rule violations, but both tools support rules which are not handled by the other. SourceMeter provides metric-based rules as well, which are triggered when the calculated metric values for a specific source code element exceed or fall behind a given acceptable metric interval. In case of detecting code clones or duplicates, SonarQube can identify copy-pasted code clones or Type-1 clones. SourceMeter uses the Abstract Syntax Tree (AST) as an input for clone detection, hence it can detect Type-2 clones (for instance, using different identifiers will not bypass the detector). After evaluating both tools, we investigated the effect of low level characteristics on higher level attributes, namely the quality indices. We used technical debt and SQALE metrics, which are provided by the SonarQube platform and are using the coding rule violations heavily, but not other low-level characteristics (source code metrics or code duplications), thus these indicators can not reflect or express the overall quality of the system well. The summary of the comparison of low-level characteristics can be seen in Table 3. We found that SourceMeter is more advanced in analysis depth, product metrics and finding duplications, while their performance on coding rules and analysis success is rather balanced. Considering all these factors, we chose to apply SourceMeter in the following research areas, which use low-level quality characteristics to calculate sophisticated high-level quality indices.

Table 3: Overall comparison results

Aspect	**Result**	**Note**
Analysis success	Balanced	SonarQube failed to analyze some input files
Analysis depth	SourceMeter	SourceMeter provides statistics in lower levels
Code metrics	SourceMeter	SourceMeter provides much more metrics
Coding rules	Balanced	Large common set, balanced rule-sets
Code duplications	SourceMeter	SourceMeter found more duplicated code blocks

Integrating Continuous Quality Monitoring Into Existing Workflow – A Case Study

The goal of this research was to create a general and flexible quality model [1] for the RPG programming language, which we then applied in our case study. Having a good quality model means having metrics at a higher level with strong descriptive capabilities such as Quality Index and Maintainability Index. Our constructed quality model is based on the ISO/IEC 25010 standard and focuses on maintainability as the main component of quality (thus quality model and maintainability model expressions used interchangeably). The model includes reusability, analyzability, modifiability, and testability as the subcharacteristics contributing to maintainability. The built model is shown in Figure 1, where dark gray nodes are the subcharacteristics, light gray nodes are helpers defined by us to ease the aggregation and grouping of sensor nodes, which are the low-level quality attributes provided by SourceMeter.

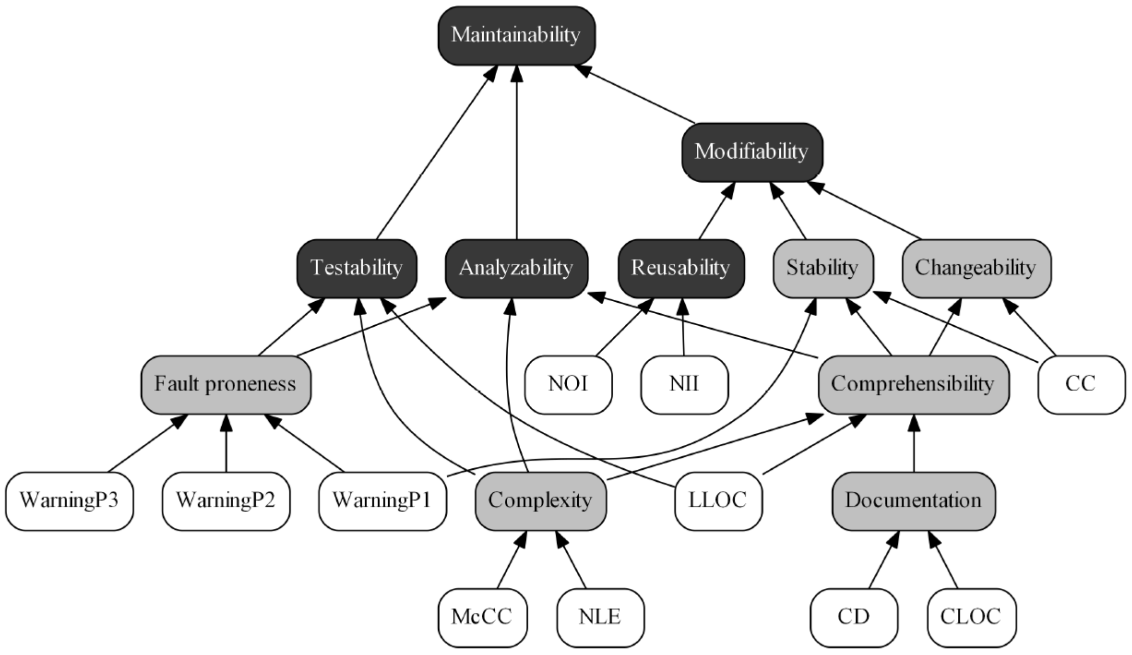

Figure 1: RPG maintainability model

After defining a quality model, we carried out a case study, in which we integrated our quality model into the development cycle of a mid-sized company, named R&R Software Ltd. After fine tuning the SourceMeter for RPG tool (e.g. setting up parameters for metric based rules, determining forbidden operations) and the quality model as well (determining the weights in the model), we constructed a benchmark in the *initial phase* which consists of 4 modules.

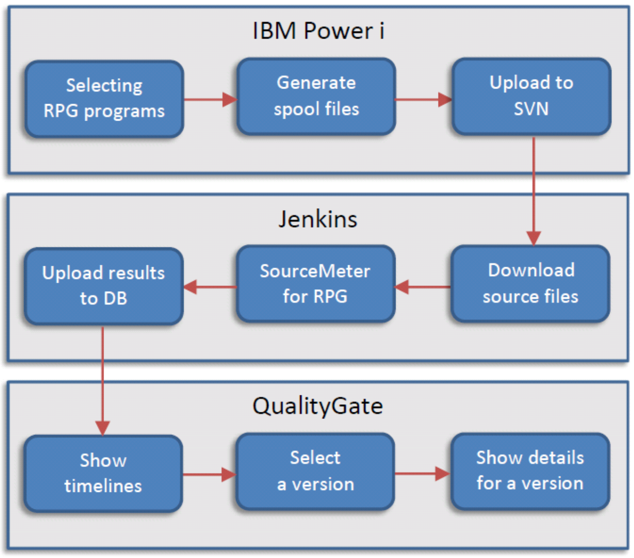

Figure 2: Process chain of the approach.

In the second, *integration phase* (depicted in Figure 2), we adapted a method to seamlessly integrate the continuous quality monitoring into the development cycle of the company. Commits automatically trigger source code analysis, the results of which are stored in a database for long term access. The company intended to increase the quality of a specific module, thus a *refactoring phase* was applied, in which some developers eliminated critical and major rule violations from the system. Doing so, the maintainability of the selected module increased continuously almost from commit to commit (shown in Figure 3).

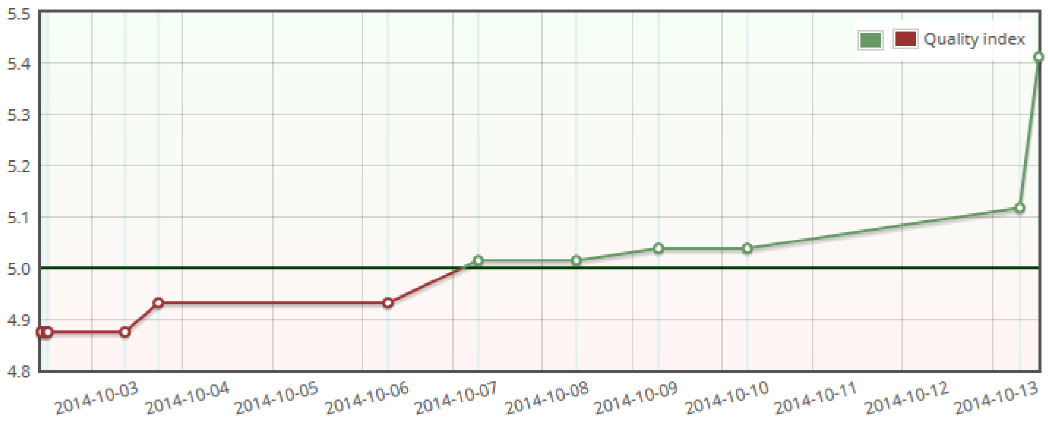

Figure 3: Maintainability quality timeline.

In the *discussion phase*, we concluded that the maintainability characteristic of the chosen module had increased and had acquired the GO state, since it passed the baseline value. Based on the opinions of the developers and the management, the industrial application of our method was a success. They were able to check coding conventions inside the company, which forces developers to avoid undesirable solutions and to learn common practices for solving different problems. Furthermore, they found it easy to customize our approach (creating benchmark, weighting the edges of the quality model). According to the developers, this maintenance work could be done effectively, because of the guidance of the SourceMeter and QualityGate tools.

Redesign of Halstead's Complexity Metrics and Maintainability Index for RPG

This research is intended to extend the quality model capabilities by applying further metrics into the model. We first proposed the definitions of the Halstead's complexity metrics for RPG/400 and RPG IV. There is no standard way to calculate these metrics, since different programming languages contain different language constructs that can be either operands and operators. In case of RPG, we extended former definitions for RPG II and RPG III [3] to be complete for RPG/400 and RPG IV, where many new language features were introduced. Next, we examined the Halstead's complexity metrics and four Maintainability Index metrics in detail to get more insight about how they correlate with other software product metrics and how we could use them to improve the capabilities of the quality model to better describe the system under investigation. To do so, we used Principal Component Analysis (PCA) to show the dimensionality and behavior of these metrics [5]. We found that Halstead's complexity metrics form a strongly correlated metric group that can be used to reveal more details about RPG software systems. Principal Component Analysis, furthermore, outputs the so-called factor loadings, which show the linear combination of the most dominant factors (dimensions) that captures the most variability. Factor loadings can be seen in Table 4, from which we can see that the Halstead's Complexity metrics are the most dominant ones in the first factor (which captures more than the half of the variability both at program and subroutine level).

Table 4: Factor loadings

	Program					**Subroutine**				
	F1	**F2**	**F3**	**F4**	**F5**	**F1**	**F2**	**F3**	**F4**	**F5**
CC	0,153	-0,078	0,616	0,663	0,017	-0,047	-0,081	**0,947**	-0,081	-0,167
HCPL	**0,969**	0,082	-0,090	-0,028	0,072	**0,853**	-0,397	-0,100	-0,193	0,069
HDIF	**0,874**	0,020	0,166	-0,109	-0,249	**0,725**	0,442	0,014	0,291	-0,199
HEFF	**0,891**	0,358	-0,151	-0,009	0,049	**0,774**	-0,198	0,035	0,530	-0,044
HNDB	**0,955**	0,260	-0,070	-0,039	-0,035	**0,928**	-0,052	0,010	0,328	-0,079
HPL	**0,966**	0,230	-0,074	-0,038	-0,007	**0,888**	-0,426	-0,064	-0,026	0,030
HPV	**0,971**	0,022	-0,055	-0,040	0,051	**0,906**	-0,245	-0,102	-0,237	0,046
HTRP	**0,891**	0,358	-0,151	-0,009	0,049	**0,774**	-0,198	0,035	0,530	-0,044
HVOL	**0,956**	0,259	-0,099	-0,030	0,016	**0,827**	-0,508	-0,071	-0,021	0,053
MI	**-0,771**	0,580	-0,136	0,068	-0,127	**-0,892**	-0,285	0,065	0,305	0,032
MIMS	**-0,771**	0,580	-0,136	0,068	-0,127	**-0,891**	-0,286	0,066	0,309	0,032
MISEI	**-0,736**	0,643	0,071	-0,102	0,010	**-0,887**	-0,297	0,076	0,312	0,058
MISM	**-0,745**	0,631	0,049	-0,072	0,034	**-0,877**	-0,313	0,076	0,323	0,061
NLE	0,602	-0,326	0,186	-0,240	-0,288	0,463	0,664	-0,097	0,035	-0,049
McCC	**0,947**	0,260	-0,090	-0,042	-0,020	0,678	0,393	0,092	0,478	0,044
NOI	0,297	-0,430	0,377	-0,422	-0,281	0,258	0,315	0,203	-0,036	**0,782**
CD	0,072	0,215	0,627	-0,516	0,412	**-0,724**	-0,439	0,155	0,348	0,215
CLOC	0,537	0,185	0,455	-0,065	0,150	**0,771**	-0,455	0,008	-0,108	0,247
NII	-	-	-	-	-	-0,059	0,182	0,047	0,208	0,123
LLOC	0,516	-0,502	-0,180	0,185	0,549	**0,899**	-0,393	-0,059	-0,033	0,046
Warning Info	**0,837**	0,206	0,314	0,306	-0,110	0,397	-0,012	**0,897**	-0,088	-0,095
Clone Metric Rules	**0,768**	0,209	0,379	0,362	-0,125	0,188	-0,029	**0,948**	-0,162	-0,129
Complexity Metric Rules	**0,899**	0,202	-0,101	-0,068	-0,033	0,538	0,453	0,089	0,311	0,055
Coupling Metric Rules	**0,813**	0,332	-0,106	-0,008	-0,028	0,156	0,191	0,201	0,028	**0,829**
Doc. Metric Rules	**0,808**	0,126	-0,312	0,053	0,081	0,439	0,125	0,002	0,199	-0,244
Size Metric Rules	**0,947**	0,021	-0,030	-0,105	0,001	**0,705**	-0,330	-0,011	0,056	0,001

As a final statement, we suggest to involve the Halstead's Number of Delivered Bugs (HNDB) metric into the model to contribute to the calculation of fault proneness since it has the largest correlation coefficients with the warning occurrences. Furthermore, we also recommend including the Halstead's Program Vocabulary (HPV) metric to contribute to the Complexity aggregated node, since it has low correlation with the McCabe's cyclomatic complexity in case of subroutines, but it has a large weight in the linear combination in factor loading (dominant metric), thus McCabe's complexity, NLE and HPV forms a unit together to describe the overall complexity of the system.

The Author's Contributions

The author led the effort of implementing the SourceMeter for RPG toolchain that is capable of parsing and analyzing RPG/400 and RPG IV programs. SourceMeter for RPG serves as the basis for the comparison and for the quality model as well. He collected the most relevant studies related to quality assurance in RPG software systems. He gathered the RPG source code to be analyzed in the comparison research and which was also a benchmark in the case study. He also ran the static analyzers and collected their results which he later compared exhaustively. He participated in the organization and the implementation of the case study. Making suggestions about the extension of the quality model and performing the Principal Component Analysis are also the author's work. The publications related to this thesis point are:

- **Zoltán Tóth**, László Vidács, and Rudolf Ferenc. Comparison of Static Analysis Tools for Quality Measurement of RPG Programs. In Proceedings of the 15th International Conference on Computational Science and Its Applications (ICCSA 2015), Banff, Alberta, Canada. Pages 177–192, Published in Lecture Notes in Computer Science (LNCS), Volume 9159, Springer-Verlag. June, 2015.

- Gergely Ladányi, **Zoltán Tóth**, Rudolf Ferenc, and Tibor Keresztesi. A Software Quality Model for RPG. In Proceedings of 2015 IEEE 22nd International Conference on Software Analysis, Evolution and Reengineering (SANER). Pages. 91–100. IEEE (2015).

- **Zoltán Tóth**. Applying and Evaluating Halstead's Complexity Metrics and Maintainability Index for RPG. In Proceedings of the 17th International Conference on Computational Science and Its Applications (ICCSA 2017), Trieste, Italy. Pages 575-590, Published in Lecture Notes in Computer Science (LNCS), Volume 10408, Springer-Verlag. July, 2017.

Summary

In this thesis, we discussed two main topics, these being the construction of new bug datasets with their evaluation in bug prediction, and a methodology for measuring maintainability in legacy systems written in the RPG programming language.

In case of *bug datsets*, we collected existing public bug datasets that use static software product metrics to characterize the bugs. These datasets often operate with the set of classic Chidamber & Kemerer object oriented metrics but nothing more. The available set of projects are quite old, since they were part of older datasets. These datasets encapsulate data gathered from various platforms, such as SourceForge, Jira, Bugzilla, CVS, and SVN. GitHub, being a trend for hosting open-source projects, was a good candidate to gather new projects from. We constructed a new dataset in order to overcome these deficiencies and to propose a new dataset with up-to-date bug data. Moreover, we presented a method for unifying public bug datasets, thus they share common metrics as descriptors. We showed how heterogeneous different datasets can be by comparing their metric suites. We also presented the capabilities of built bug prediction models in the case of the newly created GitHub Bug Dataset and in the case of the Unified Bug Dataset as well. We suggest that researchers should first try using existing bug datasets, and only if none of them conforms to their needs, construct a new customized dataset for their very specific requirements.

In the field of *maintainability in RPG systems*, we first introduced an in-depth comparison of state-of-the-art static source code analyzers for RPG systems. We provided a methodology on how to properly measure the maintainability, which is the most dominant characteristic of software quality, for RPG legacy systems. We also performed a case study, in which we successfully integrated our methodology into a mid-sized company's development lifecycle. Finally, we showed how the measuring of maintainability can be further improved by involving Halstead's Complexity Metrics in the model we built.

№	[8]	[2]	[9]	[6]	[7]
I.	♦	♦			
II.			♦	♦	♦

Table 5: Thesis contributions and supporting publications

Acknowledgements

It comes as no surprise, but I could not have managed to get here without the help of others. First, I would like to thank my supervisor, Dr. Rudolf Ferenc, for his guidance and for reinforcing me when I was full of doubts. He showed me that being enthusiastic and dedicated to a research area always pays off but one must never give up. This way of thinking should be applied to every other aspect of my life as well. I would also like to thank Dr. Tibor Gyimóthy, the former head of the Software Engineering Department, for supporting me on my way to becoming a PhD student, and for providing me with interesting research topics. Many thanks to my co-authors and my colleagues, namely Dr. István Siket, Péter Gyimesi, and Gergely Ladányi, who helped me overcome technical problems and gave me many great ideas. Many thanks to Edit Szűcs for her stylistic and grammatical comments on this thesis.

Last, but not least I wish to express my gratitude to my beloved family including my first lady, Dorka and my one and only little princess, Olivia. I greatly appreciate my mother for pushing me forward and giving me so much support during the years.

This research was supported by the Hungarian Government and the European Regional Development Fund under the grant number GINOP-2.3.2-15-2016-00037 ("Internet of Living Things").

Zoltán Tóth, 2019

References

[1] Tibor Bakota, Péter Hegedűs, Péter Körtvélyesi, Rudolf Ferenc, and Tibor Gyimóthy. A probabilistic software quality model. In *Software Maintenance (ICSM), 2011 27th IEEE International Conference on*, pages 243–252. IEEE, 2011.

[2] Rudolf Ferenc, Zoltán Tóth, Gergely Ladányi, István Siket, and Tibor Gyimóthy. A public unified bug dataset for java. In *Proceedings of the 14th International Conference on Predictive Models and Data Analytics in Software Engineering*, PROMISE'18, pages 12–21, Oulu, Finland, Oct 2018. ACM.

[3] Sandra D Hartman. A counting tool for rpg. In *ACM SIGMETRICS Performance Evaluation Review*, volume 11, pages 86–100. ACM, 1982.

[4] Haibo He, Edwardo Garcia, et al. Learning from imbalanced data. *Knowledge and Data Engineering, IEEE Transactions on*, 21(9):1263–1284, 2009.

[5] I.T. Jolliffe. *Principal Component Analysis*. Springer-Verlag New York, 2 edition, 2002.

[6] Gergely Ladányi, Zoltán Tóth, Rudolf Ferenc, and Tibor Keresztesi. A software quality model for rpg. In *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, pages 91–100, Montreal, QC, Canada, March 2015.

[7] Zoltán Tóth. Applying and evaluating halstead's complexity metrics and maintainability index for rpg. In *17th International Conference on Computational Science and Its Applications (ICCSA 2017)*, volume 10408, pages 575–590, Trieste, Italy, July 2017. Lecture Notes in Computer Science (LNCS).

[8] Zoltán Tóth, Péter Gyimesi, and Rudolf Ferenc. A public bug database of github projects and its application in bug prediction. In *16th International Conference on Computational Science and Its Applications (ICCSA 2016)*, volume 9789, pages 625–638, Beijing, China, July 2016. Lecture Notes in Computer Science (LNCS).

[9] Zoltán Tóth, László Vidács, and Rudolf Ferenc. Comparison of static analysis tools for quality measurement of rpg programs. In *15th International Conference on Computational Science and Its Applications (ICCSA 2015)*, volume 9159, pages 177–192, Banff, AB, Canada, June 2015. Lecture Notes in Computer Science (LNCS).

[10] Chadd Williams and Jaime Spacco. Szz revisited: verifying when changes induce fixes. In *Proceedings of the 2008 workshop on Defects in large software systems*, pages 32–36. ACM, 2008.