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

A THESIS SUBMITTED FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY
OF THE UNIVERSITY OF SZEGED

University of Szeged
Ph.D. School in Computer Science

"Well, writing novels is incredibly simple: an author sits down... and writes."

— Christopher Hopper

Preface

As a child I always loved having challenging problems. In the beginning, I mainly loved puzzles in math. Later, in elementary school, we had informatics and we were taught Comenius Logo, a graphical programming language. Our teacher gave us the following problem: "Draw a house with a door and a window, the height and width of which is defined by parameters". It was easy to draw the door and the window with a constant factor. When I finished this task, the teacher asked whether I could have managed to have a roof on that house. That was a much harder problem to figure out since I had no clue how long the diagonal of a square was in 7th class (square root is taught in 8th class). Well okay, no problem, let's try what happens if I go with the turtle to some point and then turn 90 degrees right and draw a line with the exact same length. The roof sometimes ended too early, sometimes it went way over the top left corner, but finally I figured out a fairly good approximation for $\sqrt{2}$ by hand.

Maybe this success led me toward programming and motivated me to learn how different mathematical formulas could be implemented in a program. During the process of deepening my knowledge in computer science I realized that it is not enough to use already discovered laws, but I want to participate in investigations and contribute to the human knowledge. And now here I am, working on my PhD dissertation, which I hope will help better understanding the connections of laws in computer science.

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.

Zoltán Tóth, 2019

Contents

List of Tables

List of Figures

Listings

To my beloved family...

"Research is to see what everybody has seen, and to think what nobody else has thought."

— Albert Szent-Györgyi

1 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 [111]. In 2016, that number is $1.1 trillion [3]. 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 [2].

The later the phase in which the bug is fixed, 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 and keep costs low. 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 a methodology for measuring the maintainability (which is one of the most important quality characteristics) of 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) [102]. 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 software systems could be a matter of national interest. Contrarily, research tends 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, based on the research areas we described.

We introduce new datasets and their assessments in bug prediction in the **first part**, which includes 3 chapters. Chapter 2 briefly introduces the most important public bug datasets and positions them in the field. In Chapter 3, we construct and evaluate a new bug dataset while keeping the weak points of existing ones in mind. Refreshing and validating the most important public bug datasets, we put them into one unified dataset to ease their joint use in the future. The unification and the assessment of this dataset is described in Chapter 4.

The **second part**, which consists of 4 chapters, deals with the question of maintainability in RPG legacy systems. Chapter 5 introduces the main concepts and an overall impression of the RPG programming language. In Chapter 6, we compare state-of-the-art RPG static source code analysis tools, which includes our own toolchain as well. Chapter 7 presents an applied methodology for measuring maintainability in RPG systems, and also describes the experiences collected during a case study in which we successfully integrated this methodology into the development cycle of a mid-ranged software company named R&R Software Ltd. We end this part by showing a possible extension of the previous methodology in Chapter 8.

Chapter 9 sums up the thesis, while in appendices A and B, brief summaries of the thesis are shown in English and in Hungarian, respectively. The appendices, furthermore, contain the thesis points, as well as the author's contributions, and the underlying publications.

Part I

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

"Let he who has a bug free software cast the first stone."

— Assaad Chalhoub

2 Public Bug Datasets

Publishing datasets as public resources for the scientific community is not a new idea [113, 68]. Many papers have dealt with bug datasets and used some kind of bug prediction approaches as demonstrations [42]. Notwithstanding the numerous studies dealing with bug prediction, the number of publicly available bug datasets are incredibly low and neglected. We mainly focus on the datasets which include source code elements (file, class, method) with bug occurrences and characterize the code elements with their static source code metrics (such as LOC - Logical Lines of Code, complexity metrics, etc.).

Researchers often use a dataset created for their own purposes, but these datasets are not published for the community. Public datasets, however, could increase the number of replicable studies which would increase the quality of bug prediction models. With this goal in mind, we collect the public bug datasets and we briefly introduce them in this chapter. Later on, some of them will be investigated in detail in the following chapters.

2.1 PROMISE

PROMISE (a.k.a. tera-PROMISE) [68] is one of the largest research data repositories in software engineering. It is a collection of many different datasets, including the NASA MDP (Metric Data Program) dataset, which was used by numerous studies in the past. However, one should always mistrust the data that comes from an external source [84, 40, 92, 41]. The repository is created to encourage repeatable, verifiable, refutable, and improvable predictive models of software engineering. This is essential for maturation of any research discipline. One main goal is to extend the repository to other research areas as well. The repository is community based, thus anybody can donate a new dataset or public tools, which can help other researchers in building state-of-the-art predictive models. PROMISE provides the datasets under categories like code analysis, testing, software maintenance, and it also has a category for defects.

2.2 Bug Prediction Dataset

The *Bug prediction dataset* [36] contains data extracted from 5 Java projects by using inFusion and Moose to calculate the classic C&K metrics for class level. The source of information was mainly CVS, SVN, Bugzilla and Jira from which the number of pre- and post-release defects were calculated. D'Ambros et al. also extended the source code metrics with change metrics, which, according to their findings, could improve the performance of the fault prediction methods.

2.3 Eclipse Bug Dataset

Zimmerman et al. [113] mapped defects from the bug database of Eclipse 2.0, 2.1, and 3.0. The resulting data set lists the number of pre- and post-release defects on the granularity of files and packages that were collected from the BUGZILLA bug tracking system. They collected static code features using the built-in Java parser of Eclipse. They calculated some features at a finer granularity; these were aggregated by taking the average, total, and maximum values of the metrics. Data is publicly available and was used in many studies since then. Last modification on the dataset was submitted on March 25, 2010.

2.4 Bugcatchers Bug Dataset

Hall et al. presented the *Bugcatchers* [45] Bug Dataset, which operates with bad smells (solely), and found that coding rule violations have a small but significant effect on the occurrence of faults at file level. The Bugcatchers Bug Dataset contains bad smell information about Eclipse, ArgoUML, and some Apache software systems for which the authors used Bugzilla and Jira as the sources of the data.

2.5 Additional datasets

There are additional datasets that are good candidates to experiment with. However, we focused on investigating the datasets that characterize source code elements with static source code metrics. There are recent datasets that would have matched with our plans, however, they have been published after we had done our investigations. We will show some of these datasets and also briefly introduce them.

Software-artifact Infrastructure Repository, or simply **SIR** is a repository of software related artifacts meant to support rigorous controlled experimentation with program analysis and software testing techniques, and education in controlled experimentation. SIR focuses on the bugs from the perspective of software testing, thus the included information is reflecting this approach.

iBugs [35] provides a bug repository with data extracted from real projects with the main purpose of bug localization. As the authors stated, iBugs is a collection of datasets to complement the SIR repository. While SIR mainly contains datasets with seeded (artificially injected) faults, iBugs tries to do the same with real software defects. However, this dataset is still approaching the bugs from the testing perspective (test coverage matrices are used as characterization).

The **ELFF** dataset [93] is a recent one that fulfills the need for more method level bug datasets. The ELFF dataset includes 23 open-source project which were analyzed with JHawk. The projects were mined from SourceForge and are the final contestants from a set of 50,000 candidates.

The **Had-oops!** dataset [47] is constructed with a new approach presented by Harman et al. They analyzed 8 consecutive Hadoop versions and investigated the impact of chronology on fault prediction performance. They used Support Vector Machines (SVMs) with a genetic algorithm (for configuration) to build prediction models at class level. For a given version, they constructed a prediction model from all the previous versions and a model from the current version only and compared which one performed better. Results are not straightforward since they found early versions preferable in many cases as opposed to models built on recent versions. Moreover, using all versions is not always better than using only the current version to build a model from.

The **Mutation-aware fault prediction dataset** is a result of an experiment carried out by Bowes et al. on using mutation metrics as independent variables for fault prediction [22]. They used 3 software systems from which 2 projects (Eclipse and Apache) were open-source and one was closed. They used the popular PITest (or simply, PIT [31]) to obtain the set of mutation metrics that were included in the final dataset. Besides the mutation metrics, some static source code metrics (calculated by JHawk [1]) were also included in the dataset for comparison purposes. This dataset is also built at class level.

Defects4J is a well-known dataset, also with the goal of providing a dataset for ensuring comparable, reproducible research studies [57]. The dataset includes real bugs with the corresponding fixing commits and with information about the passing and failing tests. Currently, 6 open-source projects are included with a total of 438 real software bugs. Fixing commits are manually pruned or cleaned to exclude unimportant refactorings or feature additions. Bugs that should be fixed by modifying configuration files, documentations, or test files are also ignored.

Awesome-MSR[1] is a curated repository for sharing datasets and tools to conduct evidence-based, data-driven research on software systems. The naming 'MSR' comes from the Mining Software Repositories (MSR) conference series. Awesome-MSR collects different repositories related to the aforementioned topics as well, such as PROMISE, SIR, FLOSSmole. Besides repositories, it lists standalone datasets like Defects4J. Different tools that can be used to characterize datasets are also listed. For instance, different static analyzer tools are enumerated.

[1] `https://github.com/dspinellis/awesome-msr`

"If debugging is the process of removing bugs, then programming must be the process of putting them in."

— Edsger Dijkstra

3

A New Public Bug Dataset and Its Evaluation in Bug Prediction

3.1 Overview

Software systems are likely to fail occasionally, that is obviously unwanted both for the end users and for the software developers. Keeping the software quality at a high-level is more important than ever, since customers define the reputation of the used subject system. Open-source software development paved its way, and has become a cornerstone in the domain of evaluating research ideas and techniques dealing with computer science [98]. These publicly available systems gather a huge amount of historical data stored, for example, in version control systems or bug tracking systems. Researchers have been using the opportunity, given by these public information sets, to prove the power of their approaches for a long time [6, 76, 113]. In spite of this fact, only a few publicly available bug datasets are presented to take role as a basis for further investigations (we listed these datasets in Chapter 2). Many authors do not make the corpus used in their studies public, thus the experiments are not repeatable [59].

Why on earth would anybody want to construct yet another bug dataset if we have already some out there? In our research work, we made an exhaustive investigation to gather all the existing public bug datasets. Unfortunately, these datasets mainly operate with classic C&K [29] metrics and contain accumulated information about bugs at a pre-release or post-release time. Our goal is to include a wider range of metrics, including the number of different coding rule violations and additional code clone metrics as well.

None of these datasets consist of data obtained from GitHub, they mostly gathered them from Bugzilla, Jira, SVN or SourceForge. We conducted an experiment using GitHub as the source of information (both for version control and for bug tracking). GitHub is the absolute trend for hosting open-source projects, thus we can gather information about actively developed, modern, community based software systems.

Not only programming languages, but the way of thinking as a programmer are all actors in the software evolution process. New constructions in programming languages

force developers to think in a different way. This leads to committing different mistakes nowadays than they have in the past. The only way to avoid this pitfall is to keep public datasets curated, up to date, and to include new systems from different domains.

Our study tries to endorse the use of public datasets for addressing different research questions, such as the ones relating to bug prediction, by showing the power of our automatically generated bug dataset in the bug prediction domain. We have developed a toolchain that automatically gathers different information about publicly available projects to build a bug dataset. We selected 15 Java projects from different domains to ensure the generality of the constructed dataset. The characteristics of these open-source projects were extracted from GitHub[1] that hosts millions of projects, using a static source code analyzer tool called SourceMeter.[2] We analyzed these projects that include more than 3.5 million lines of code, and more than 114 thousand of commits in total. From the analyzed commit set, we detected almost 6 thousand commits that referenced at least one bug (inducing a bug fix intention) according to the SZZ algorithm [102]. We used release versions of the systems and created bug datasets for approximately six-months-long intervals.

To show the usefulness of the gathered information, we experimented with 13 machine learning algorithms and achieved quiet promising results. For class level, the best algorithms resulted in higher than 0.7 F-measure values. For file level, we achieved similar, if a little lower, values. Almost full bug coverage can be reached by using these models by tagging only 31% of the source code elements as buggy. We defined two research questions, which are the following:

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

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

The remainder of this chapter is organized as follows. Section 3.2 enumerates the most important research papers dealing with bug prediction techniques, especially the ones focusing on the mining of software repositories (for related bug datasets please refer to Chapter 2). In Section 3.3, we propose our approach and show how our dataset is constructed, and what kind of data entries are stored in it. Next, we introduce the set of selected projects in Section 3.4. Section 3.5 presents the power of the constructed dataset by evaluating different results of the applied machine learning algorithms. Finally, we summarize and conclude the results we obtained in this research.

3.2 Related Work

ReLink[107] is developed to explore missing links between changes committed in version control systems and fixed bugs. This tool could be helpful for any software engineering research that is based on the linkage data, such as software defect prediction. ReLink mines and analyzes information like bug reporter, description, comments, date from bug dataset and then tries to pair the bug with the appropriate source code files based on the set of source code information extracted from a version control system.

[1] `https://github.com`

[2] `https://www.sourcemeter.com`

The history of version control systems shows us the concerned files and their changed lines only, but software engineers are also interested in which source code elements (e.g. classes or methods) are affected by a change or a bug [103]. In our study, we presented a method for tracking low level source code elements' (class, method) positions in files by processing version control system log information [116]. This method helps to keep source code positions up-to-date during the processing of commits.

Kalliamvakou et al. mined GitHub repositories to investigate their characteristics and their qualities [58]. They presented a detailed study discussing different project characteristics, such as (in)activity. Further research questions were involved – whether a project is standalone or a part of a more massive system. Results have shown that the extracted data set can serve as good input for various investigations, however, one must use them with mistrust and always verify the usefulness and reliability of the mined data. It is a good practice to choose projects with many developers and commits, moreover, one should keep in mind that the most important point is to choose projects that fit your own purpose well. In our case, we have tried to create a dataset that is reliable (some manual validation is performed) and general enough for testing different bug prediction techniques.

Bird et al. presented a study on distributed version control systems, thus the paper focuses mainly on Git [20]. They examined the usage of version control systems and the available set of data (such as whether the commits are removable, modifiable, movable) gathered by the type of usage (with respect of differentiating central and distributed systems). The main purpose of this paper was to draw attention to pitfalls and help researchers to avoid such pitfalls during the processing and analysis of mined Git information set.

Many research papers have shown that using a bug tracking system improves the quality of the developed software system. Bangcharoensap et al. introduced a method to locate the buggy files in a software system very quickly using the bug reports managed by the bug tracking system [12]. The presented method contains three different approaches to rank the fault-prone files, namely:

- Text mining: ranks files based on the textual similarity between a bug report and the source code itself.
- Code mining: ranks files based on prediction of the potential buggy module using source code product metrics.
- Change history: ranks files based on prediction of the fault-prone module using change process metrics.

They used the gathered project data collected on the Eclipse platform to investigate the efficiency of the proposed approaches. Finally, they showed that these three ways are suitable for locating buggy files. Furthermore, bug reports with short description and many specific words greatly increase the effectiveness of finding the weak points (the files) of the system.

Not only the above presented method can be used to predict the occurrence of a new bug, but a significant change in source code metrics can also be a clue that the relevant source code files contain a potential bug or bugs [42]. Couto et al. presented a paper that shows the possible relationship between changed source metrics (used as predictors) and bugs [33]. They described an experiment to discover more robust evidence towards causality between software metrics and the occurrences of bugs.

We constructed our dataset with the above mentioned techniques in mind.

3.3 Approach

In this section, we describe how the dataset is constructed. The process is a kind of toolchain, so we will introduce our method phase-by-phase.

We first downloaded the data from GitHub (cloning of the selected repositories), then we processed the raw data to obtain statistical measurements on the projects (these statistics are presented later, in Section 3.4). At this point we selected the relevant software versions to be analyzed by the static source code analyzer. We tried to mark six-month-long intervals for all projects and find the nearest release versions on which an analysis should be performed. After the source code analysis, we performed the dataset building step. We detected the references between the commits and the bugs by using the SZZ algorithm [102]. GitHub also provides the linkage between issues and commits. These links are determined from the messages of the commits. With the use of these links, we accumulated the bug related source code elements (faulty classes). A source code element is bug related, if it was modified in a commit that references the issue (it has to be modified in order to fix the bug).

Our dataset creation process is similar to traditional ones described in various studies [68, 36, 113]. Let us consider a few bugs that were closed (see Figure 3.1).

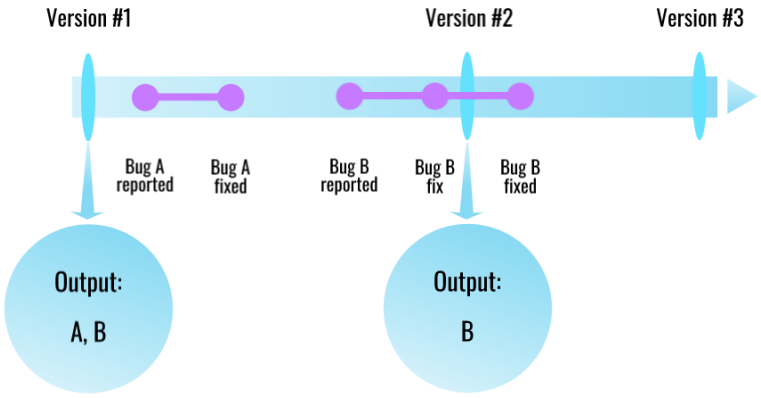

Figure 3.1. The relationship between the bugs and release versions

There are 3 versions of the system and we have 2 bugs in the software. Reporting a bug means that it was revealed in the system, but it may have been present in the system for a long time. We considered the most recent release before the date of the report, and supposed that the bug is present in the system at that time and not in the next release if it was fixed until. Other traditional datasets, contrarily, gather buggy source code elements not for the preceding release version, but for the succeeding one, noting that the source code element was faulty in pre-release state. This is the main difference in the construction process between our and other traditional datasets. In this manner, let us see how this construction works in case of the example. We fixed bug A before version 2, which means bug A is present in the system in version 1, but not in version 2. The same is true for bug B, however, bug B was finally fixed after version 2, thus bug B also appears in the output of version 2, not only in version A. At this point, bug A is already fixed, which causes it not to appear in version B. These examples show how our dataset creation algorithm behaves when the lifecycle of a bug overlaps a selected release version.

For the construction of the dataset we used the so-called traditional approach that means we collected release versions with approximately six-month-long time intervals

for every project. We used six-month-long intervals, since enough bugs and versions are present for such long time intervals. Based on the age of a project, the number of selected release versions could differ for each project. We selected the release versions manually from the list of releases located on the projects' GitHub pages. It is a common practice that projects use the release tag on a newly branched (from master) version of the source code. Since we only use the master branch as the main source of information, we had to perform a mapping when the hash id of the selected release was not representing a commit located in the master. Developers usually branch from master and then tag the branched version as release version, so our mapping algorithm detects when (time stamp) the release tag was applied on a version and searches for the last commit in the master branch that was made right before this timestamp.

We created a dataset for each of the selected release versions. Since bug tracking was not always used from the beginning of the projects, we could not assign any bug information to some of these earlier release versions. Also, the changing developer activity could result in a lack of bug reports and, consequently, bug fixing commits are rare. All of these factors play a role in the created datasets varying in the number of bugs.

3.4 The Selected Projects and the Dataset

To select projects for the dataset construction, we examined many projects on GitHub. We considered a number of criteria during our searching process. First of all, we searched for Java language projects, especially larger ones, since these are more suitable for this kind of analysis. It was also important to have an adequate number of commits and to have many issues which are labeled as bugs. Moreover, it is desired to have enough references to the appropriate bug report from the description of the commits. In addition, we preferred the currently active projects. We found many projects during our search, which would have fulfilled most aspects, but, in many cases, developers used an external bug tracker system, so it would have been more difficult to process them and it would have made the automatic extraction process more complex.

Table 3.1. The selected projects

Project	Domain	kLOC	NC	NBR	Class	File	DB Files
Android Universal I. L.	Android library	13	996	89	639	478	12
ANTLR v4	Language processing	85	3,276	111	2,353	2,029	10
Broadleaf Commerce	E-commerce framework	283	9,292	652	17,433	14,703	22
Eclipse p. for Ceylon	IDE	165	6,847	666	4,512	2,129	1
Elasticsearch	Search engine	677	13,778	2,108	54,562	23,252	24
Hazelcast	Computing platform	515	16,854	2,354	25,130	14,791	18
jUnit	Test framework	36	2,053	74	5,432	2,266	16
MapDB	Database engine	83	1,345	175	2,740	962	12
mcMMO	Game	42	4,552	657	1,393	1,348	12
Mission Control T.	Monitoring platform	204	975	37	6,091	1,904	6
Neo4j	Database engine	648	32,883	439	32,156	18,306	18
Netty	Networking framework	282	6,780	1,039	11,528	8,349	18
OrientDB	Database engine	380	10,197	174	11,643	9,475	12
Oryx	Machine learning	47	363	36	2,157	1,400	8
Titan	Database engine	119	3,830	121	5,312	3,713	12
Total		**3,579**	**114,021**	**8,732**	**183,078**	**105,105**	**210**

The selected 15 software systems are listed in Table 3.1 with several additional statistics. The first column shows the name of the projects (the corresponding GitHub links are enumerated in the public downloadable package). The next column is the main domain of these systems. We can see that there is a large variance between the projects regarding the domain that strengthens the generality of the constructed dataset. The next three columns are the thousand Lines of Code, the Number of Commits and the Number of Bug Reports, respectively, on the master branch measured in May of 2015. Class and File columns show the number of source code elements in the system at that time. DB Files gives the number dataset files for the system. For instance, the Android Universal Image Loader project has 12 DB Files in total, which means that there are 6 selected release versions and we constructed both a class and a file level dataset for these release versions.

These datasets are in CSV format (comma separated values) for the sake of ease of use. The first row in the CSV files contains header information, such as unique identifier, source code position, source name, metric names, rule violation groups, and number of bugs. The actual data in the rest of the lines follows this order. Each line represents a source code element (class, file). In total, we selected 105 release versions for the 15 projects, and created 210 datasets files for six-month-long intervals. The last three columns in Table 3.1 present the number of entries constructed for each project.

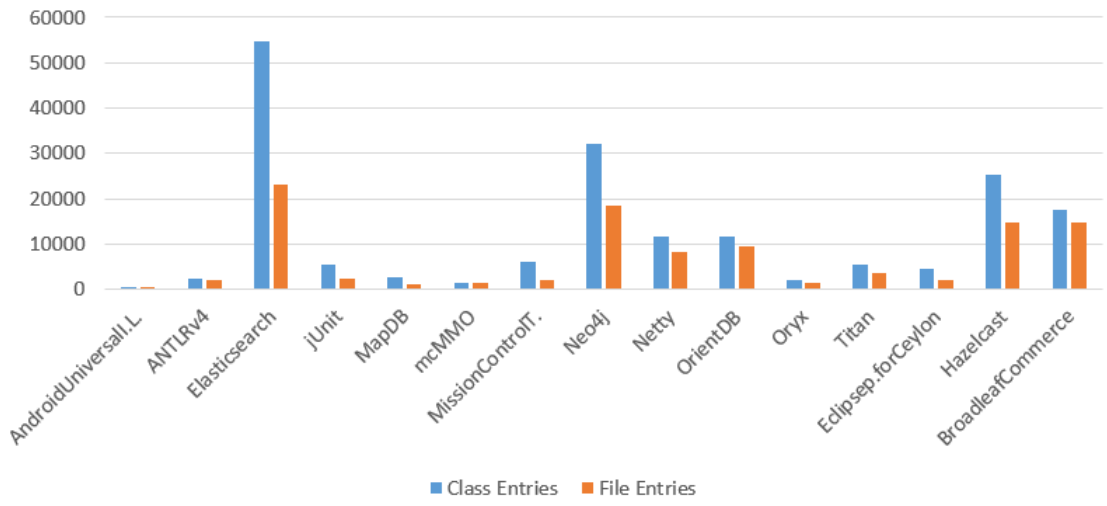

Figure 3.2. Number of entries distribution

For a better visualization, Figure 3.2 depicts the above mentioned entry numbers on a bar chart. Some projects have an outstanding number of class and file entries, however, we are going to present results on every project one-by-one by evaluating the best machine learning algorithms for different release versions. Out of the total 183,078 class level entries, Elasticsearch has 54,562 in 12 datasets, which is not surprising if we consider the size of the project (677 kLOC). However, Neo4J has the most commits (twice as much as the second project which is Hazelcast), it has considerably less bug reports that results in a smaller dataset. In general, the bigger the project and the more bug reports a project has the bigger dataset it results in.

3.5 Evaluation

In this section, we will give a detailed investigation in order to answer our research questions.

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

We evaluated our dataset by applying machine learning algorithms for all of the constructed datasets. The bug information in our dataset is present as the number of bugs. To apply machine learning classification, first we grouped the source code elements into two classes based on the occurrence of bugs in them. Instances with non-zero bug cardinality form a class (defective elements) and instances with zero bug number constitute the second separate class (non-defective elements), in other words, we labeled the source code elements as buggy or not buggy.

If we look at the ratio between the number of defective and the number of non-defective elements, we may notice that there are way more non-defective elements in a software version than defective. Considering that we are planning to apply machine learning algorithms, this could distort the results, because the non-buggy instances can get more emphasis during the training phase. To deal with this issue, we applied a random undersampling method to equalize the learning corpus [49, 99]. We randomly selected elements from the non-buggy class to match the size of the buggy category. This way we got a training set with the same number of positive and negative instances. We repeated this kind of learning (with random undersampling) 10 times and calculated an average. For the training, we used 10-fold cross validation and compared the results based on precision, recall, and F-measure metrics where these metrics are defined in the following way:

$$precision = \frac{TP}{TP + FP}$$

$$recall = \frac{TP}{TP + FN}$$

$$F - measure = 2 \cdot \frac{precision \cdot recall}{precision + recall}$$

where TP (True Positive) is the number of classes/files that were predicted as faulty and observed as faulty, FP (False Positive) is the number of classes/files that were predicted as faulty, but observed as not faulty, FN (False Negative) is the number of classes/files that were predicted as non-faulty, but observed as faulty. We carried out the training with the popular machine learning library called Weka.[3] It contains algorithms from different categories, for instance Bayesian methods, support vector machines, and decision trees. We used the following 13 algorithms:

- NaiveBayes (Bayes)
- NaiveBayesMultinomial (Bayes)
- Logistic (Function)
- SGD (Function)

[3] http://www.cs.waikato.ac.nz/ml/weka/

- SimpleLogistic (Function)
- SMO (Function)
- VotedPerceptron (Function)
- DecisionTable (Rule)
- OneR (Rule)
- PART (Rule)
- J48 – C4.5 (Tree)
- RandomForest (Tree)
- RandomTree (Tree)

We analyzed software versions with six-month intervals from 15 projects. In total, we selected 105 release versions. 80 of these versions contain bug information, which is the consequence of our applied method, as it was shown in Section 3.3. There can be versions, especially the last selected ones, where no bug information could be obtained. 5 out of the 80 versions contain too few buggy elements to apply machine learning. We ended up with 75 suitable versions for the training on class level. On file level, we got only 72, because in one buggy file there could be more than one buggy class, thus the size of the training set for a specific version could differ based on the granularity of the dataset.

Class level First, we investigated whether the class level datasets are suitable for bug prediction purposes. Presenting the results for all 15 projects using all 13 machine learning algorithms would end up in a giant table that human eyes would not be able to process, or at least could not focus on the most relevant parts. Consequently, we only present the best algorithms here to make it easier to overview and find the best ones. Furthermore, for each project we selected the interval which has the most buggy dataset entries to ensure the suitable size of the training corpus. Then, we used 10-fold cross-validation for that interval as described earlier. We chose the algorithms simply

Table 3.2. 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**

Table 3.3. F-measures at file level

Project	Logistic	Simple Logistic	PART	J48	Random Forest
Android Universal I. L.	0.5983	0.6230	0.6632	0.6215	0.6214
ANTLR v4	0.7638	**0.7941**	0.7443	**0.8267**	0.7645
Broadleaf Commerce	0.7244	0.7206	0.7736	0.7797	0.7875
Eclipse p. for Ceylon	0.6664	0.6403	0.7141	0.7026	0.6837
Elasticsearch	0.6303	0.6280	0.6718	0.7025	0.7169
Hazelcast	0.6883	0.6980	0.6742	0.6790	0.6946
jUnit	0.6950	0.6530	0.6142	0.6613	0.6591
MapDB	0.7466	0.7337	0.7702	0.7790	**0.8158**
mcMMO	0.6864	0.6717	0.6583	0.6509	0.6951
Mission Control T.	0.7039	0.6700	0.6287	0.6573	0.7049
Neo4j	0.6621	0.7154	0.6766	0.6504	0.7150
Netty	0.6483	0.6549	0.6646	0.6823	0.7120
OrientDB	0.6868	0.6772	0.7157	0.7182	0.7234
Oryx	0.5537	0.5687	0.6500	0.6569	0.7331
Titan	0.6590	0.6813	0.6595	0.6407	0.6919
Avg.	**0.6742**	**0.6753**	**0.6853**	**0.6939**	**0.7146**

by calculating the averages of F-measure values and considered the best 5 algorithms. Table 3.2 presents the F-measure values for these 5 algorithms at class level. As one can observe, values can differ significantly project by project, which can be caused by various reasons, such as the size of the constructed dataset. For instance, let us consider the Android Universal Image Loader and the Broadleaf Commerce projects. The Android project is the smallest in size, Broadleaf is one of the middle-sized projects. Android has 639 class level entries in total (6 DB files), however Broadleaf has 17,433 entries (11 DB files), which is more suitable for being a training corpus. Nevertheless, if we take a closer look at the results, we can see that the best F-measure values also occurred in small projects such as in Oryx or MCT, ergo we cannot generalize this conjecture to be true; however, further investigations should be done to prove that. Tree-, function- and rule-based models performed the best in this scenario. F-measure values are up to 0.8210, which is a promising result. Before answering the first research question, let us investigate the results at file level as well.

File level File level is different in some aspects from class level. For example, a completely distinct set of metrics (and also fewer) are calculated for file level entries. The best file level machine learning results are shown in Table 3.3. At first glance, one can see that the results are in a wider range than in the case of class level. However, RandomForest has the highest F-measure values in case of files too. Furthermore, another tree based algorithm (J48) also performs nicely in this case. Two function-based (Logistic and SimpleLogistic) and one rule-based algorithm are in the top. Considering these results, we can answer our research question.

Answering RQ 1: *Considering F-measure values for the chosen releases we can state that such datasets are suitable for bug prediction. In the bug prediction domain, the RandomForest performed the best in addition to function and rule based machine learning algorithms, thus one should consider these first to build prediction models using our datasets.*

After having insight into the bug prediction results, another question is put into words. The algorithms could perform better regarding recall, if they mark more classes/files buggy, of course with a decrease in precision. It is an important aspect to see how many bugs are covered by the marked classes/files, and what proportion of classes/files were marked as buggy. In other words, we investigate the precision and recall values in terms of bug coverage, and see how the F-Measure values are constructed.

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

Contrary to the investigation for the previous research question, in this context we cannot perform the same evaluation since we used random under sampling to equalize the number of buggy and non-buggy source code elements for the learning corpus, thus not all entries are included in the evaluation. For bug coverage, we used the previously built 10 models (for the balanced training sets - with random under sampling, and having the average as the result) 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 as faulty then we tagged it as faulty, otherwise we tagged it as non-faulty).

Table 3.4. Bug coverage at class level

Project	**Naive Bayes**	**PART**	**J48**	**Random Forest**	**Random Tree**
Android Universal I. L.	0.71 (0.21)	1.00 (0.39)	1.00 (0.47)	1.00 (0.42)	1.00 (0.42)
ANTLR v4	0.93 (0.20)	1.00 (0.35)	1.00 (0.26)	1.00 (0.27)	1.00 (0.27)
Broadleaf Commerce	0.60 (0.19)	1.00 (0.30)	0.94 (0.29)	1.00 (0.28)	1.00 (0.31)
Eclipse p. for Ceylon	0.79 (0.14)	1.00 (0.34)	0.98 (0.27)	1.00 (0.32)	1.00 (0.36)
Elasticsearch	0.86 (0.14)	1.00 (0.33)	1.00 (0.32)	1.00 (0.32)	1.00 (0.32)
Hazelcast	0.85 (0.14)	0.99 (0.32)	0.99 (0.31)	1.00 (0.31)	1.00 (0.32)
jUnit	0.82 (0.15)	1.00 (0.26)	1.00 (0.29)	1.00 (0.27)	1.00 (0.24)
MapDB	1.00 (0.25)	1.00 (0.29)	1.00 (0.21)	1.00 (0.26)	1.00 (0.26)
mcMMO	0.72 (0.18)	1.00 (0.40)	1.00 (0.39)	1.00 (0.41)	1.00 (0.36)
Mission Control T.	0.80 (0.21)	1.00 (0.22)	1.00 (0.32)	1.00 (0.18)	1.00 (0.17)
Neo4j	1.00 (0.14)	1.00 (0.34)	1.00 (0.27)	1.00 (0.36)	1.00 (0.39)
Netty	0.82 (0.18)	0.98 (0.34)	0.98 (0.32)	0.98 (0.35)	0.98 (0.33)
OrientDB	0.83 (0.18)	1.00 (0.31)	1.00 (0.32)	1.00 (0.31)	1.00 (0.33)
Oryx	0.92 (0.26)	1.00 (0.30)	0.93 (0.25)	1.00 (0.28)	1.00 (0.30)
Titan	0.66 (0.11)	0.94 (0.29)	0.94 (0.29)	0.94 (0.29)	0.94 (0.32)
Avg.	0.82 (0.18)	0.99 (0.32)	0.99 (0.31)	1.00 (0.31)	1.00 (0.31)

Table 3.4 and Table 3.5 show the bug coverage values (ratio of covered bugs) and the ratio of how many classes or files have been tagged as faulty to obtain that bug coverage ratio. Trees are performing the best, if we only consider the bug coverage, however, they tagged more than 31% of the source code elements as buggy in average. NaiveBayes is the other end of the story, since it has the lowest average in bug coverage values (0.82 can be acceptable), but tags the smallest amount of entries as buggy. Same results occurred at file level, but here we present some other algorithms (not the best five) to show the differences in machine learning algorithms. We can state that our dataset is useful for finding bugs in software systems with high bug coverage ratio, still tagging only a fair amount of entries as buggy.

Table 3.5. Bug coverage at file level

Project	Random Forest	Decision Table	SGD	Logistic	Naive Bayes
Android Universal I. L.	1.00 (0.46)	1.00 (0.68)	0.46 (0.10)	0.81 (0.27)	0.81 (0.33)
ANTLR v4	1.00 (0.32)	0.91 (0.30)	0.91 (0.20)	0.91 (0.24)	0.82 (0.18)
Broadleaf Commerce	1.00 (0.33)	0.88 (0.34)	0.78 (0.21)	0.80 (0.24)	0.69 (0.14)
Eclipse p. for Ceylon	1.00 (0.39)	1.00 (0.42)	0.76 (0.21)	0.83 (0.25)	0.65 (0.11)
Elasticsearch	1.00 (0.39)	0.94 (0.35)	0.82 (0.19)	0.83 (0.24)	0.73 (0.16)
Hazelcast	1.00 (0.38)	0.95 (0.37)	0.87 (0.21)	0.89 (0.28)	0.80 (0.12)
jUnit	1.00 (0.44)	0.94 (0.30)	0.83 (0.25)	0.83 (0.31)	0.89 (0.20)
MapDB	1.00 (0.33)	1.00 (0.36)	0.93 (0.19)	0.97 (0.28)	0.90 (0.25)
mcMMO	1.00 (0.42)	0.93 (0.44)	0.81 (0.27)	0.82 (0.29)	0.75 (0.21)
Mission Control T.	1.00 (0.25)	1.00 (0.38)	1.00 (0.24)	1.00 (0.24)	1.00 (0.19)
Neo4j	1.00 (0.30)	1.00 (0.38)	1.00 (0.24)	1.00 (0.25)	0.80 (0.12)
Netty	1.00 (0.44)	0.99 (0.60)	0.85 (0.34)	0.88 (0.36)	0.73 (0.14)
OrientDB	1.00 (0.41)	0.97 (0.49)	0.95 (0.42)	0.92 (0.38)	0.79 (0.14)
Oryx	1.00 (0.43)	1.00 (0.66)	0.64 (0.17)	0.73 (0.32)	0.36 (0.09)
Titan	1.00 (0.37)	1.00 (0.45)	1.00 (0.64)	0.89 (0.38)	0.72 (0.11)
Avg.	1.00 (0.38)	0.97 (0.43)	0.84 (0.26)	0.87 (0.29)	0.76 (0.17)

Answering RQ 2: *Tree based machine learning algorithms performed best in this scenario, with the highest bug coverage ratio. At class level, circa 31% of the elements were tagged as buggy, but the F-measure values are still high (averaged higher than 0.77). For file level, the values are lower (more entries have to be marked as buggy and the F-measure values are about 0.71), but in total the results are very similar to class level.*

Since there is a lack of space to introduce wide tables here, we present our whole set of results as an online appendix together with the full bug dataset at the following URL: `http://www.inf.u-szeged.hu/~ferenc/papers/GitHubBugDataSet/`

3.6 Summary

In this chapter, we proposed an approach for creating a bug dataset for selected release versions of 15 projects in an automatic way using the popular source code hosting platform named GitHub. We gathered the 15 Java projects from different domains to fulfill the need of generality. After constructing six-month-long release intervals, we collected bugs and the corresponding source code elements and organized them into datasets. Our dataset is differentiated from the previous ones by gathering data from GitHub, the approach for constructing the dataset is slightly different from previous ones, and we also included a wider set of static source code metrics. We made our new bug dataset public to augment the set of available datasets with a recent one. We applied 13 machine learning algorithms to investigate whether the dataset is usable for bug prediction purposes. We experienced quite good results for tree based algorithms (Random Forest, J48, Random Tree) with respect of F-measure values and bug coverage ratios. In case of F-measure, we could reach higher than 0.8 values in some cases at class level (averaged 0.77), and little lower, but similar values could be granted at file level (averaged 0.71). 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.

"I don't care if it works on your machine! We are not shipping your machine!"

— Vidiu Platon

4

A Unified Public Bug Dataset and Its Assessment in Bug Prediction

4.1 Overview

Finding and eliminating bugs in software systems has always been one of the most important issues in software engineering. Software testing is often limited because of the given resources, thus a more focused resource allocation should be applied. Bug localization is conducted when we want to find the exact locations of the occurring bugs. Bug localization is a crucial and very expensive part of software engineering, therefore, many researches have examined this topic and several different approaches were proposed that tried to reduce costs and create more powerful methods [106].

Bug or defect prediction is a process by which we try to learn from mistakes committed in the past and build a prediction model to leverage the location and amount of future bugs. Many research papers were published on bug prediction, that introduced new approaches that aimed to achieve better precision values [112, 109, 44, 101]. Unfortunately, a reported bug is rarely associated with the source code lines that caused it or with the corresponding source code elements (e.g. classes, methods). Therefore, to carry out such experiments, bugs have to be associated with source code (or with classes or methods) which in and of itself is a difficult task (this is where bug localization steps in). It is necessary to use a version control system and a bug tracking system properly during the development process, and even in this case it is still challenging to associate bugs with the problematic source code locations.

Although several algorithms were published on how to associate a reported bug with the relevant, corresponding defective source code [34, 105, 28], only few such bug association experiments were carried out. Furthermore, not all of these studies published the bug dataset or even if they did, closed source systems were used which limits the verifiability and reusability of the bug dataset. In spite of these facts, several bug datasets (containing information about open-source software systems) were published and made publicly available for further investigations or to replicate previous approaches [100, 90].

The main advantage of these bug datasets is that if someone wants to create a new bug prediction model or validate an existing one, it is enough to use a previously created bug dataset instead of building a new one, which is very resource consuming. It is common in these bug datasets that all of them store some specific information about the bugs, such as the containing source code element(s) with their source code metrics or any additional bug related information. Since different bug prediction approaches try to use various sources of information as predictors (independent variables), different bug datasets are constructed. Defect prediction approaches and hereby bug datasets can be categorized into larger groups based on the captured characteristics [37]:

- Datasets using process metrics [73, 75].
- Datasets using source code metrics [15, 23, 96].
- Datasets using previous defects [62, 82].

Different bug prediction approaches use various public or private bug datasets. Although these datasets seem very similar, they are often very different in some aspects, which is also true within the categories mentioned above. As in previous chapters, we will focus solely on datasets that use static source code metrics. Since this category itself has grown so immense, it is worth studying it as a separate unit. This category also has many dissimilarities between the existing datasets, including the granularity of the data (source code elements can be files, classes, or methods, depending on the purpose of the given research or on the capabilities of the tools used to extract data) and the representation of element names (different tools may use different notations). For the same reason, the set of metrics can be different as well. Even if the name or the abbreviation of a metric calculated by different tools is the same, it can have different meanings because it can be defined or calculated in a slightly different way. The bug related information given for a source code element can also be contrasting. An element can be labeled with whether or not it contains a bug, but it can also show how many bugs are related to that given source code element. From the information content perspective, it is less important, but not negligible that the format of the files containing the data can be CSV (Comma Separated Values), XML, or ARFF (which is the input format of Weka [43]), and these datasets can be found on different places on the Internet.

A constructed dataset can represent a good input for machine learning algorithms to build prediction models [14, 66, 70]. Some researchers argued that the used dataset is not as important as the applied machine learning algorithm [69]. However, the selection of software metrics from which a prediction model is built can severely influence the accuracy and the complexity of the model [86].

Finally, there is usually a lack of information about the reliability and no specification is given on how a given dataset should be used. On the other hand, it would be a difficult task and would require a lot of effort to validate the metric values and the number of bugs, especially for systems where the source code is not available for the public. In spite of all these drawbacks, researchers should consider using these bug datasets first, and not create new, specialized ones if it is possible. They can build new ones if needed, but first they should be attentive and try to use public datasets and further improve them. Our contributions can be listed as follows:

- Collection of the bug datasets and the source code of included projects.
- Unification of the collected bug datasets.

- Extension of the metrics suites.
- Assessment of the datasets.
- Making the results publicly available.

4.2 Data Collection

Gathered datasets were briefly introduced in Chapter 2. In the remaining of this chapter, we will go into the details and see how we collected and analyzed these datasets. Data collection can be divided into two parts, the first part is the collection and the evaluation of the literature review papers in the subject area. In the second part, we used the literature review papers, and the case studies presented in them, to collect the bug datasets themselves with the corresponding source code.

Collecting literature review papers Starting from the early 70's [88, 52] a large number of studies were introduced in connection with software faults. According to Yu et al. [110], 729 studies were published until 2005 and 1564 until 2015 on bug prediction (the number of studies has doubled in 10 years). From time to time, the enormous number of new publications in the topic of software faults made it unavoidable to collect the most important advances in literature review papers. By using the existing literature review papers, we were able to focus on the empirical aspects of the collected datasets.

Collecting bug datasets We went through the union of the references used in the review studies and filtered out the relevant papers based on keywords, title, abstract and the introduction. Then we collected all available information about the used bug datasets located in the remaining set of scientific papers. We took into consideration the following properties:

- Basic information (authors, title, date, publisher).
- Accessibility of the bug dataset (public, non public, partially public).
- Availability of the source code.

The latter two were extremely important when investigating the datasets, since we could not construct a unified dataset without obtaining the appropriate underlying data. As we collected the literature review papers, we created a list of the found datasets and repositories. Furthermore, we included a few additional papers which were published recently, so they were not included in any previous literature review. We considered the following list to check whether a dataset meets our requirements:

- the dataset is publicly available,
- source code is accessible for the included systems,
- source code elements are characterized by static source code metrics,
- bug information is provided,
- bugs are associated with the relevant source code elements,
- included projects were written in Java,
- the dataset provides bug information at file/class level, and

- the source code element names are provided and unambiguous (the referenced source code is clearly identifiable).

If any condition was missing then we had to exclude the subject system or the whole dataset from the study. The list of found public datasets were enumerated in Chapter 2. In our research, we could only include the following datasets:

- PROMISE [68]
- Eclipse Bug Dataset [113]
- Bug Prediction Dataset [36]
- Bugcatchers Bug Dataset [45]
- GitHub Bug Dataset [118] (our own dataset which we presented in Chapter 3)

Some datasets were only excluded from the unification because they were published after our study was already in progress. In the following sections, we will describe the chosen datasets in more details and investigate each dataset's peculiarities and we will also look for common characteristics, but before doing so, we will show some basic statistics about the datasets. Table 4.1 describes the size of the datasets and also presents the number of software systems included. Number of versions included in the datasets are also shown. For instance, the Eclipse Bug Dataset includes 3 versions of Eclipse. We used the cloc[1] program to measure the Lines of Code. We only considered Java files and we also neglected blank lines.

Table 4.1. Basic statistics about the public bug datasets

Dataset	Systems	Versions	Lines of Code
PROMISE	14	45	2,805,253
Eclipse Bug Dataset	1	3	3,087,826
Bug Prediction Dataset	5	5	1,171,220
Bugcatchers Bug Dataset	3	3	1,833,876
GitHub Bug Dataset	15	105	1,707,446

4.3 Data Processing

Although the found public datasets have similarities (e.g. containing source code metrics and bug information), they are very inhomogeneous. For example, they contain different metrics, which were calculated with different tools and for different kinds of code elements. The file formats are different as well, therefore, it is very difficult to use these datasets together. Consequently, our aim was to transform them into a unified format and to extend them with source code metrics that are calculated with the same tool for each system. In this section, we will describe the steps we performed to produce the unified bug dataset.

First, we transformed the existing datasets to a common format. This means that if a bug dataset for a system consists of separate files we conflated them into one file. Next, we changed the *CSV* separator in each file to *comma (,)* and renamed

[1] `https://www.npmjs.com/package/cloc`

the number of bug column in each dataset to '*bug*' and the source code element column name to '*filepath*' or '*classname*' depending on the granularity of the dataset. Finally, we transformed the source code element identifier into the standard form (e.g. *org.apache.tools.ant.AntClassLoader*).

4.3.1 Metrics Calculation

The bug datasets contain different kinds of metric sets, which were calculated with different tools, therefore, even if the same metric name appears in two or more different datasets, we cannot be sure they mean exactly the same metric. To eliminate this deficiency, we analyzed all the systems with the same tool. For this purpose, we used the free and open-source *OpenStaticAnalyzer*(OSA)[2] static source code analyzer tool that is able to analyze Java systems (among other languages). It calculates more than 50 different kinds (size, complexity, coupling, cohesion, inheritance, and documentation) of source code metrics for packages and class-level elements, about 30 metrics for methods, and a few ones for files. OpenStaticAnalyzer detects code duplications (Type-1 and Type-2 clones) as well, and calculates code duplication metrics for packages, classes, and methods. OpenStaticAnalyzer has two different kinds of textual outputs: the first one is an XML file that contains, among others, the whole structure of the source code (files, packages, classes, methods), their relationships and the metric values for each element (e.g. file, class, method). The other output format is CSV. Since different elements have different metrics, there is one CSV file for each kind of element (one for packages, one for classes, and so on).

For calculating the new metric values we needed the source code itself. Since all datasets belonged to a release version of a given software, therefore, if the software was open-source and the given release version was still available, we could manage to download and analyze it. This way, we obtained two results for each system: one from the downloaded bug datasets and one from the OpenStaticAnalyzer analysis.

4.3.2 Dataset Unification

We merged the original datasets with the results of OSA by using the "unique identifiers" of the elements (Java standard names at class level and paths at file level). More precisely, the basis of the unified dataset was our source code analysis result and it was extended with the data of the given bug dataset. This means that we went through all elements of the bug dataset and if the "unique identifier" of an element was found in our analysis result then these two elements were conjugated (paired the original dataset entry with the one found in the result of *OSA*), otherwise it was left out from the unified dataset. Table 4.2 and Table 4.3 show the results of this merging process: column *OSA* shows how many elements OpenStaticAnalyzer found in the analyzed systems, column *Orig.* presents the number of elements originally in the datasets, and column *Dropped* tells us how many elements of the bug datasets could not be paired, and so they were left out from the unified dataset. Although these numbers are very good, we had to "modify" a few systems to achieve this, but there were cases where we simply could not solve the inconsistencies. The details of the source code modifications and main reasons for the dropped elements were the following:

[2] `https://github.com/sed-inf-u-szeged/OpenStaticAnalyzer`

Table 4.2. Merging results – Class level datasets

Dataset	Name	OSA	Orig.	Dropped
PROMISE	Ant 1.3	530	125	0
	Ant 1.4	602	178	0
	Ant 1.5	945	293	0
	Ant 1.6	1,262	351	0
	Ant 1.7	1,576	745	0
	Camel 1.0	734	339	0
	Camel 1.2	1,348	608	13 (+5)
	Camel 1.4	2,339	872	0 (+31)
	Camel 1.6	3,174	965	0 (+38)
	Ckjm 1.8	9	10	1
	Forrest 0.6	159	6	0
	Forrest 0.7	76	29	0
	Forrest 0.8	53	32	0
	Ivy 1.4	421	241	0
	Ivy 2.0	637	352	0
	JEdit 3.2	552	272	0
	JEdit 4.0	647	306	0
	JEdit 4.1	722	312	0
	JEdit 4.2	888	367	0
	JEdit 4.3	1,181	492	0
	Log4J 1.0	180	135	0
	Log4J 1.1	217	109	0
	Log4J 1.2	410	205	0
	Lucene 2.0	758	195	1
	Lucene 2.2	1,394	247	1
	Lucene 2.4	1,522	340	1
	Pbeans 1	38	26	0
	Pbeans 2	77	51	0
	Poi 1.5	472	237	0
	Poi 2.0	667	314	0
	Poi 2.5	780	385	0
	Poi 3.0	1,508	442	0
	Synapse 1.0	319	157	0
	Synapse 1.1	491	222	0
	Synapse 1.2	618	256	0
	Velocity 1.4	275	196	0
	Velocity 1.5	377	214	1
	Velocity 1.6	458	229	1
	Xalan 2.4	906	723	0
	Xalan 2.5	992	803	0
	Xalan 2.6	1,217	885	0
	Xalan 2.7	1,249	909	0
	Xerces 1.2	564	440	0
	Xerces 1.3	596	453	0
	Xerces 1.4	782	588	42
Bug Prediction Dataset	Eclipse JDT Core 3.4	2,486	997	0
	Eclipse PDE UI 3.4.1	3,382	1,497	6
	Equinox 3.4	742	324	5
	Lucene 2.4	1,522	691	21
	Mylyn 3.1	3,238	1,862	457
GitHub Bug Dataset	Android U. I. L. 1.7.0	84	73	0
	ANTLR v4 4.2	525	479	0
	Elasticsearch 0.90.11	6,480	5,908	0
	jUnit 4.9	770	731	0
	MapDB 0.9.6	348	331	0
	mcMMO 1.4.06	329	301	0
	MCT 1.7b1	2,050	1,887	0
	Neo4j 1.9.7	6,705	5,899	0
	Netty 3.6.3	1,300	1,143	0
	OrientDB 1.6.2	2,098	1,847	0
	Oryx	562	533	0
	Titan 0.5.1	1,770	1,468	0
	Eclipse p. for Ceylon 1.1.0	1,651	1,610	0
	Hazelcast 3.3	3,765	3,412	0
	Broadleaf C. 3.0.10	2,094	1,593	0
Sum	**All**	**76,623**	**48,242**	**624**

Table 4.3. Merging results – File level datasets

Dataset	Name	OSA	Orig.	Dropped
Eclipse Bug Dataset	Eclipse 2.0	6,751	6,729	0
	Eclipse 2.1	7,909	7,888	0
	Eclipse 3.0	10,635	10,593	0
Bugcatchers Bug Dataset	Apache Commons	491	191	0
	ArgoUML 0.26 Beta	1,752	1,582	3
	Eclipse JDT Core 3.1	12,300	560	25
GitHub Bug Dataset	Android U. I. L. 1.7.0	63	63	0
	ANTLR v4 4.2	411	411	0
	Elasticsearch 0.90.11	3,540	3,035	0
	jUnit 4.9	308	308	0
	MapDB 0.9.6	137	137	0
	mcMMO 1.4.06	267	267	0
	MCT 1.7b1	1,064	413	0
	Neo4j 1.9.7	3,291	3,278	0
	Netty 3.6.3	914	913	0
	OrientDB 1.6.2	1,503	1,503	0
	Oryx	443	280	0
	Titan 0.5.1	981	975	0
	Ceylon for Eclipse 1.1.0	699	699	0
	Hazelcast 3.3	2,228	2,228	0
	Broadleaf C. 3.0.10	1,843	1,719	0
Sum	**All**	**57,530**	**43,772**	**28**

Camel 1.2: In the *org.apache.commons.logging* there were 13 classes in the original dataset that we did not find in the source code. There were 5 *package-info.java* files in the system, but these files never contain any Java classes, since they are used for package level Javadoc purposes, therefore, OpenStaticAnalyzer did not find such classes.
Camel 1.4: Besides the 7 *package-info.java* files, the original dataset contained information about 24 Scala files (they are also compiled to byte code), therefore, OpenStaticAnalyzer did not analyze them.
Camel 1.6: There were 8 *package-info.java* and 30 Scala files.
Ckjm 1.8: There was a class in the original dataset that did not exist in version 1.8.
Forrest-0.8: There were two different classes that appeared twice in the source code, therefore, we deleted the 2 copies from the *etc/test-whitespace* subdirectory.
Log4j: There was a *contribs* directory which contained the source code of different contributors. These files were put into the appropriate sub-directories as well (where they belonged according to their packages), which means that they occurred twice in the analysis and this prevented their merging. Therefore, in these cases we analyzed only those files that were in their appropriate subdirectories and excluded the files found in the *contribs* directory.
Lucene: In all three versions, there was an *org.apache.lucene.search.RemoteSearchable_Stub* class in the original dataset that did not exist in the source code.
Velocity: In versions 1.5 and 1.6 there were two *org.apache.velocity.app.event.implement.EscapeReference* classes in the source code, therefore, it was impossible to conjugate them by using their "unique identifiers" only.
Xerces 1.4.4: Although the name of the original dataset and the corresponding publication state that this is the result of Xerces 1.4.4 analysis, we found that 256 out of

the 588 elements did not exist in that version. We examined a few previous and following versions as well, and it turned out that the dataset is much closer to 2.0.0 than to 1.4.4, because only 42 elements could not be conjugated with the analysis result of 2.0.0. Although version 2.0.0 was still not matched perfectly, we did not find a "closer version", therefore, we used Xerces 2.0.0 in this case.

Eclipse JDT Core 3.4: There were a lot of classes which appeared twice in the source code: once in the "code" and once in the "test" directory, therefore we deleted the test directory.

Eclipse PDE UI 3.4.1: The missing 6 classes were not found in its source code.

Equinox 3.4: Three classes could not be conjugated, because they did not have a unique name (there are more classes with the same name in the system) while two classes were not found in the system.

Lucene 2.4 (BPD): 21 classes from the original dataset were not present in the source code of the analyzed system.

Mylyn 3.1: 457 classes were missing from our analysis that were in the original dataset, therefore, we downloaded different versions of Mylyn, but still could not find the matching source code. We could not achieve better result without knowing the proper version.

ArgoUML 0.26 Beta: There were 3 classes in the original dataset that did not exist in the source code.

Eclipse JDT Core 3.1: There were 25 classes that did not exist in the analyzed system.

GitHub Bug Dataset: Since OpenStaticAnalyzer is the open-source version of SourceMeter, the tool we used to construct the dataset presented in Chapter 3, we could easily merge the results. However, the class level bug datasets contained elements having the same "unique identifier" (since class names are not the standard Java names in that case), so this information was not enough to conjugate them. Luckily, the paths of the elements were also available and we used them as well, therefore, all elements could be conjugated. Since we performed a machine learning step on the versions that contain the most bugs, we decided to select these release versions and present the characteristics of these release versions. We also used these versions of the systems to include in the unified bug dataset.

As a result of this process, we obtained a unified bug dataset which contains all of the public datasets in a unified format, furthermore, they were extended with the same set of metrics provided by the OpenStaticAnalyzer tool. The last lines of Table 4.2 and Table 4.3 show that only 1.29% (624 out of 48,242) of the classes and 0.06% (28 out of 43,772) of the files could not be conjugated, which means that only 0.71% (652 out of 92,014) of the elements were left out from the unified dataset.

In many cases, the analysis results of OpenStaticAnalyzer contained more elements than the original datasets. Since we did not know how the bug datasets were produced, we could not give an exact explanation for the differences, but we list some possible causes:

- In some cases, we could not find the proper source code for the given system (e.g. Xerces 1.4.4 or Mylyn), so two different, but close versions of the same system might be conjugated.
- OpenStaticAnalyzer takes into account nested, local, and anonymous classes, while some datasets simply associated Java classes with files.

4.4 Original and Extended Metrics Suites

In this section we present the metrics proposed by each dataset. Additionally, we will show a metrics suite that is used by the unified dataset we have constructed.

4.4.1 Original Metric Suites

The authors [68] calculated the metrics of the PROMISE dataset with the tool called *ckjm*. All metrics, except McCabe's Cyclomatic Complexity (CC), are *class* level metrics. Besides the CK metrics they also calculated some additional metrics shown in Table 4.4.

Table 4.4. Metrics used in PROMISE dataset

Name	Abbr.
Weighted methods per class	WMC
Depth of Inheritance Tree	DIT
Number of Children	NOC
Coupling between object classes	CBO
Response for a Class	RFC
Lack of cohesion in methods	LCOM
Afferent couplings	Ca
Efferent couplings	Ce
Number of Public Methods	NPM
Lack of cohesion in methods (by Henderson-Sellers)	LCOM3
Lines of Code	LOC
Data Access Metric	DAM
Measure of Aggregation	MOA
Measure of Functional Abstraction	MFA
Cohesion Among Methods of Class	CAM
Inheritance Coupling	IC
Coupling Between Methods	CBM
Average Method Complexity	AMC
McCabe's cyclomatic complexity	CC
Maximum McCabe's cyclomatic complexity	MAX_CC
Average McCabe's cyclomatic complexity	AVG_CC
Number of files (compilation units)	NOCU

In the Eclipse Bug Dataset, there are two types of predictors. By parsing the structure of the obtained abstract syntax tree, they calculated the number of nodes for each type in a package and in a *file* (e.g. the number of return statements in a file) [113]. By implementing visitors to the Java parser of Eclipse, they also calculated various complexity metrics at method, class, file, and package level. Then they used avg, max, total avg, total max aggregation techniques to accumulate to file and package level, which are the final outputs of their approach. The complexity metrics used in the Eclipse dataset are listed in Table 4.5.

The Bug Prediction Dataset collects product and change (process) metrics. The authors [36] produced the corresponding product and process metrics at *class* level. Besides the classic CK metrics, they calculated some additional object-oriented metrics that are listed in Table 4.6.

The Bugcatchers Bug Dataset is a bit different from the previous datasets, since it does not contain traditional software metrics, but the number of bad smells for files. They used five bad smells, which are the following: Data Clumps, Message Chains, Middle Man, Speculative Generality, and Switch Statements. Besides, in the CSV file,

Table 4.5. Metrics used in Eclipse Bug Dataset

Name	Abbr.
Number of method calls	FOUT
Method lines of code	MLOC
Nested block depth	NBD
Number of parameters	PAR
McCabe cyclomatic complexity	VG
Number of field	NOF
Number of method	NOM
Number of static fields	NSF
Number of static methods	NSM
Number of anonymous type declarations	ACD
Number of interfaces	NOI
Number of classes	NOT
Total lines of code	TLOC
Number of files (compilation units)	NOCU

Table 4.6. Product metrics used in Bug Prediction Dataset

Name	Abbr.
Number of other classes that reference the class	FanIn
Number of other classes referenced by the class	FanOut
Number of attributes	NOA
Number of public attributes	NOPA
Number of private attributes	NOPRA
Number of attributes inherited	NOAI
Number of lines of code	LOC
Number of methods	NOM
Number of public methods	NOPM
Number of private methods	NOPRM
Number of methods inherited	NOMI

there are four source code metrics (blank, comment, code, codeLines), which are not explained in the corresponding publication [45].

The GitHub Bug Dataset used the SourceMeter static analyzer to calculate the static source code metrics, including software product metrics, code clone metrics, and rule violation metrics. The rule violation metrics were not used in our research, therefore, Table 4.7 shows only the list of the software product and code clone metrics.

4.4.2 Unified Bug Dataset

The unified dataset contains all the datasets with their original metrics and with further metrics that we calculated with OpenStaticAnalyzer. The set of metrics calculated by OpenStaticAnalyzer concurs with the metric set of the GitHub Bug Dataset because SourceMeter is a product based on the free and open-source OpenStaticAnalyzer tool. Therefore, all datasets in the Unified Bug Dataset are extended with the metrics listed in Table 4.7 except the GitHub Bug Dataset, because it contains the same metrics originally.

In spite of the fact, that several of the original metrics can be matched with the metrics calculated by OpenStaticAnalyzer, we decided to keep all the original metrics for every system included in the unified dataset, because they can differ in their definitions or in the way of their calculation. One can simply use the unified dataset and discard the metrics that were calculated by OpenStaticAnalyzer if they only want to

Table 4.7. Metrics used in GitHub Bug Dataset

Name	Abbr.	Name	Abbr.
API Documentation	AD	Number of Local Public Methods	NLPM
Clone Classes	CCL	Number of Local Setters	NLS
Clone Complexity	CCO	Number of Methods	NM
Clone Coverage	CC	Number of Outgoing Invocations	NOI
Clone Instances	CI	Number of Parents	NOP
Clone Line Coverage	CLC	Number of Public Attributes	NPA
Clone Logical Line Coverage	CLLC	Number of Public Methods	NPM
Comment Density	CD	Number of Setters	NS
Comment Lines of Code	CLOC	Number of Statements	NOS
Coupling Between Object classes	CBO	Public Documented API	PDA
Coupling Between Obj. classes Inv.	CBOI	Public Undocumented API	PUA
Depth of Inheritance Tree	DIT	Response set For Class	RFC
Documentation Lines of Code	DLOC	Total Comment Density	TCD
Lack of Cohesion in Methods 5	LCOM5	Total Comment Lines of Code	TCLOC
Lines of Code	LOC	Total Lines of Code	TLOC
Lines of Duplicated Code	LDC	Total Logical Lines of Code	TLLOC
Logical Lines of Code	LLOC	Total Number of Attributes	TNA
Logical Lines of Duplicated Code	LLDC	Total Number of Getters	TNG
Nesting Level	NL	Total Number of Local Attributes	TNLA
Nesting Level Else-If	NLE	Total Number of Local Getters	TNLG
Number of Ancestors	NOA	Total Number of Local Methods	TNLM
Number of Attributes	NA	Total Number of Local Public Attr.	TNLPA
Number of Children	NOC	Total Number of Local Public Meth.	TNLPM
Number of Descendants	NOD	Total Number of Local Setters	TNLS
Number of Getters	NG	Total Number of Methods	TNM
Number of Incoming Invocations	NII	Total Number of Public Attributes	TNPA
Number of Local Attributes	NLA	Total Number of Public Methods	TNPM
Number of Local Getters	NLG	Total Number of Setters	TNS
Number of Local Methods	NLM	Total Number of Statements	TNOS
Number of Local Public Attributes	NLPA	Weighted Methods per Class	WMC

work with the original metrics. Furthermore, this provides an opportunity to confront the original and the OpenStaticAnalyzer metrics.

Instead of presenting all the definitions of the metrics here, we give an external resource to show metric definitions because of a lack of space. All the metrics and their definitions can be found in the Unified Bug Dataset file reachable in the following URL: `http://www.inf.u-szeged.hu/~ferenc/papers/UnifiedBugDataSet`.

4.4.3 Comparison of the Metrics

In the unified dataset, each element has a number of metrics, but these values were calculated by different tools, therefore, we assessed them in more detail to get answers to questions like the following ones:

- Do two metrics with the same name have the same meaning?
- Do metrics with different names have the same definition?
- Can two metrics with the same definition be different?
- What are the root causes of the differences if the metrics share the definition?

Three out of the five datasets contain class level elements, but unfortunately, for each dataset a different analyzer tool was used to calculate the metrics (see Table 4.12). To be able to compare class level metrics calculated by all the tools used, we needed at

Table 4.8. Number of elements in the merged systems

Name	Merged	Remained elements
Bug Prediction Dataset	11,370	4,167
Eclipse Bug Dataset	25,295	25,210
Bugcatchers bug Dataset	14,543	2,305

least one dataset for which all metrics of all three tools are available. We were already familiar with the usage of the ckjm tool, so we chose to calculate the ckjm metrics for the Bug Prediction dataset. This way, we could assess all metrics of all tools, because the Bug Prediction dataset was originally created with Moose, so we have extended it with the OpenStaticAnalyzer metrics, and also – for the sake of this comparison – with ckjm metrics.

In the case of the three file level datasets, the used analyzer tools were unavailable, therefore, we could only compare the file level metrics of OpenStaticAnalyzer with the results of the other two tools separately on Eclipse and Bugcatchers Bug datasets.

In each comparison, we merged the different result files of each dataset into one, which contained the results of all systems in the given dataset and deleted those elements that did not contain all metric values. The resulting spreadsheet files can also be found at: `http://www.inf.u-szeged.hu/~ferenc/papers/UnifiedBugDataSet`. Table 4.8 shows how many classes or files were in the given dataset and how many of them remained.[3] We calculated the basic statistics (minimum, maximum, average, median, and standard deviation) of the examined metrics and compared them. Besides, we calculated the pairwise differences of the metrics for each element and examined its basic statistics as well. Furthermore, we performed a dependent, paired t-test to see if the calculated metrics of the different tools are the same, or they are significantly different. In our test, the H_0 hypothesis means that the two metric samples are equal and H_1 means that they are statistically different. Our t-test formula is the following:

$$t = \frac{\bar{X}_D - \mu}{\frac{S_D}{\sqrt{n}}}$$

, where $\bar{X}_D$ is the mean difference (the average of the differences), μ shows what we expect when the null hypothesis is true. In our case this is 0 to mark that no difference is present between the two samples. $\frac{S_D}{\sqrt{n}}$ is the average variation, which is the standard error of the difference scores: S_D is the standard deviation, and n gives the size of the sample. We used 95% confidence level in the tests to calculate the p-values.

Class level metrics The unified bug dataset contained the class level metrics of OpenStaticAnalyzer and Moose on Bug Prediction dataset. We downloaded the Java binaries of the systems in this dataset and used ckjm version 2.2 to calculate the metrics. The first difference is that while OpenStaticAnalyzer and Moose calculate metrics on

[3]The big differences between the number of merged and remained elements is explained in Section 4.3.2, see Table 4.2 and Table 4.3.

source code, ckjm uses Java bytecode and takes "external dependencies" into account, therefore, we expected differences, for instance, in the coupling metric values.

We compared the metric sets of the three tools and found that, for example, CBO and WMC have different definitions. On the other hand, efferent coupling metric is a good example for the metric which is calculated by all three tools, but with different names (see Table 4.9, CBO row). In the following paragraphs, we only examine those metrics whose definitions coincide in all three tools even if their names differ. Table 4.9 shows these metrics where the *Metric* column contains the abbreviation of the most widely used name of the metric. The *Tool* column presents the analyzer tools, in the *Metric name* column, the metric names are given using the notations of the different datasets. The "$tool_1 - tool_2$" means the pairwise difference where, for each element, we extracted the value of tool_2 from the value of tool_1 and the name of this "new metric" is diff. The rest of the columns present the basic statistics of the metrics and the p-values (where it is appropriate). Next, we will analyze their values one metric at a time.

WMC: This metric expresses the complexity of a class as the sum of the complexity of its methods. In its original definition, the method complexity is deliberately not defined exactly; and usually the uniform weight of 1 is used. In this case, this variant of WMC is calculated by all three tools. Its basic statistics are more or less the same and the pairwise values of OpenStaticAnalyzer and ckjm are also close to each other (see OSA−ckjm row), but they very much differ from Moose. Among the Moose results, there were several very low values where the other tools found a great number of methods and that caused the extreme difference (e.g. the max. value of OSA−Moose is 420).

CBO: In this definition, CBO counts the number of classes the given class depends on. Although it is a coupling metric, it counts efferent (outer) couplings, therefore, the metric values should have been similar. On the other hand, based on the statistical values and the pairwise comparison, we can say that these metrics differ significantly. The reasons can be, for example, that ckjm takes into account "external" dependencies (e.g. classes from *java.util*) or it counts coupling based on generated elements (e.g. generated default constructor), but further investigation would be required to determine all causes.

CBOI: It counts those classes that use the given class. Although, the basic statistics of *OSA* and *ckjm* are close to each other, its pairwise comparison suggests that they are different. Moreover, the metrics values of *Moose* are very different. The main reason can be, for example, that *OSA* found two times more classes, therefore, it is explicable that more classes use the given class or *ckjm* takes into account the generated classes and connections as well that exist in the bytecode, but not in the source code.

RFC: all three tools defined this metric in the same way but the comparison shows that the metric values are very different. The reasons for this are mainly the same as in case of the CBO metric.

DIT: Although the statistical values "hardly" differ compared to the previous ones, these values are usually small (as the max. values show), therefore, these differences are quite large. From the minimal values we can see that Moose probably counts Object too as the base class of all Java classes, while the other two tools neglect this.

NOC: The values of Moose and ckjm are close to each other. This is the only case, when the null hypothesis could be accepted, which means that the Moose and ckjm tools calculate the metric the same way.

LOC: Lines of code should be the most unambiguous metric, but it also differs a

Table 4.9. Comparison of the common class level metrics (Bug Prediction dataset)

Metric	Tool	Metric name	Min	Max	Avg	Med	Dev	p-value
WMC	OSA	NLM	0	426	11.04	7	18.12	-
	Moose	Methods	0	403	9.96	6	14.38	-
	ckjm	WMC	1	426	11.96	7	18.49	-
	OSA−Moose	diff	-4	**420**	1.08	0	9.40	1.42E-13
	OSA−ckjm	diff	-48	0	-0.91	0	1.94	3.73E-183
	Moose−ckjm	diff	**-421**	4	-1.99	-1	9.57	2.21E-40
CBO	OSA	CBO	0	214	8.86	5	12.25	-
	Moose	fanOut	0	93	6.22	4	7.79	-
	ckjm	Ce	0	213	13.78	8	16.88	-
	OSA−Moose	diff	-32	161	2.65	2	7.61	1.94E-105
	OSA−ckjm	diff	-120	83	-4.91	-1	9.72	4.53E-208
	Moose−ckjm	diff	-160	32	-7.56	-4	11.84	0.00E+00
CBOI	OSA	CBOI	0	607	9.38	3	26.14	-
	Moose	fanIn	0	355	4.69	1	14.30	-
	ckjm	Ca	0	611	7.64	2	22.13	-
	OSA−Moose	diff	-18	**607**	4.69	1	16.55	6.96E-72
	OSA−ckjm	diff	-100	189	1.74	0	11.02	3.85E-24
	Moose−ckjm	diff	**-611**	146	-2.95	-1	15.30	7.35E-35
RFC	OSA	RFC	0	600	22.82	12	34.53	-
	Moose	rfc	0	2,603	50.62	23	108.06	-
	ckjm	RFC	2	684	38.93	23	49.72	-
	OSA−Moose	diff	**-2,095**	600	-27.80	-8	83.70	8.47E-97
	OSA−ckjm	diff	-327	12	-16.11	-9	22.72	0.00E+00
	Moose−ckjm	diff	-673	**2,049**	11.69	-1	75.42	2.76E-23
DIT	OSA	DIT	0	8	1.31	1	1.63	-
	Moose	dit	**1**	9	2.08	2	1.44	-
	ckjm	DIT	0	5	0.38	0	0.60	-
	OSA−Moose	diff	-3	0	-0.76	-1	0.43	0.00E+00
	OSA−ckjm	diff	-5	8	0.94	1	1.96	6.97E-188
	Moose−ckjm	diff	-4	9	1.70	2	1.79	0.00E+00
NOC	OSA	NOC	0	107	0.73	0	3.27	-
	Moose	noc	0	49	0.64	0	2.55	-
	ckjm	NOC	0	107	0.64	0	2.95	-
	OSA−Moose	diff	-3	97	0.08	0	1.68	1.43E-03
	OSA−ckjm	diff	0	42	0.09	0	1.15	4.32E-07
	Moose−ckjm	diff	-97	34	0.01	0	1.81	**7.84E-01**
LOC	OSA	LLOC	2	8,746	131.99	56	357.39	-
	Moose	LinesOfCode	0	7,341	124.01	51	306.54	-
	ckjm	LOC	4	26,576	399.42	147	1142.60	-
	OSA−Moose	diff	-1,068	7,824	7.98	3	157.69	1.09E-03
	OSA−ckjm	diff	-19,150	112	**-267.43**	-91	791.30	5.34E-100
	Moose−ckjm	diff	-26,541	198	**-275.41**	-93	879.89	1.11E-86
NPM	OSA	NLPM	0	404	7.23	4	13.67	-
	Moose	PublicMethods	0	387	6.42	4	11.28	-
	ckjm	NPM	0	404	7.48	5	13.64	-
	OSA−Moose	diff	-4	236	0.81	0	6.55	1.47E-15
	OSA−ckjm	diff	-8	0	-0.25	0	0.45	1.75E-236
	Moose−ckjm	diff	-237	3	-1.06	0	6.55	2.97E-25

lot. Although this metric has several variants and it is not defined exactly how Moose and ckjm counts it, we used the closest one from OpenStaticAnalyzer based on the metric values. The very large value of ckjm is surprising, but it counts this value from the bytecode, therefore, it is not easy to validate. Besides, OpenStaticAnalyzer and

Moose have different values, in spite of the fact that both of them calculate LOC from source code. The 0 minimal value of Moose is also interesting and suggests that either Moose used a different definition or the algorithm was not good enough.

NPM: The number of public methods metrics of OpenStaticAnalyzer and ckjm are really close to each other, while Moose has different results in this case as well. However, the average difference is around 1, which can be caused by counting implicit methods (constructors, static init blocks) or not.

Considering the p-values, we can conclude that none of the metrics are calculated the same way except NOC (highlighted in the table). The comparison of the three tools revealed that, even though, they calculate the same metrics, the results are very divergent. A few of its reasons can be that *ckjm* calculates metrics from bytecode while the other two tools work on source code, or *ckjm* takes into account external code as well while *OSA* does not. Besides, we could not compare the detailed and precise definitions of the metrics to be sure that they are really calculated in the same way, therefore, it is possible that they differ slightly which causes the differences.

File level metrics Bugcatchers, Eclipse, and GitHub Bug Dataset are the ones that operate at file level (GitHub Bug Dataset contains class level too). Unfortunately, we could make only pairwise comparisons between file level metrics, since we could not replicate the measurements used in the Eclipse Bug Dataset (custom Eclipse JDT visitors were used) and in the Bugcatchers Bug Dataset (unknown bad smell detector was used).

In case of Bugcatchers Bug Dataset, we compared the results of OpenStaticAnalyzer and the original metrics which were produced by a code smell detector. Since OpenStaticAnalyzer only calculates a narrow set of file level metrics, Logical Lines of Code (LLOC) is the only metric we could use in this comparison. Table 4.10 presents the result of this comparison. Min, max, and median values are likely to be the same. Moreover, the average difference between LLOC values is less than 1 with a standard deviation of 6.05 which could be considered as insignificant in case of LLOC at file level, however, the t-test showed the opposite, i.e. these metrics values are significantly different. There is an additional common metric (CLOC) which is not listed in Table 4.10 since OpenStaticAnalyzer returned 0 values for all the files. This possible error in OpenStaticAnalyzer makes it superfluous to examine CLOC in further detail.

Table 4.10. Comparison of file level metrics on Bugcatchers

Metric	Tool	Met. name	Min	Max	Avg	Med	Dev	p-value
LLOC	OSA	LLOC	3	5,774	93.33	41	221.16	-
	Smell Detector	code	3	5,774	92.34	40	219.06	-
	OSA−Smell Detector	diff	-11	130	**0.98**	0	**6.05**	8.42E-15

In case of the Eclipse Bug Dataset, LLOC values are the same in most of the cases (see Table 4.11). OpenStaticAnalyzer counted one extra line in 10 cases out of 25,210 which is a negligible difference (as t-test also confirms this statement). Unfortunately, there is a serious sway in case of the McCabe's Cyclomatic Complexity. There is a case where the difference is 299 in the calculated values which is extremely high for this metric. We investigated these cases and found that OpenStaticAnalyzer does not include the number of methods in the final value. There are many cases when OpenStaticAnalyzer gives 1 as a result while the Eclipse Visitor calculates 0 as complexity.

It is probable that OpenStaticAnalyzer counts class definitions but not method definitions which could be specious. There are cases where OpenStaticAnalyzer has higher complexity values. It turned out that OpenStaticAnalyzer took the ternary operator (?:) into consideration, which is correct, since these statements also form conditions. Both calculation techniques seem to have some minor issues or at least we have to say that the metric definitions of cyclomatic complexity differ.

Table 4.11. Comparison of file level metrics on Eclipse

Metric	Tool	Metric name	Min	Max	Avg	Med	Dev	p-value
LLOC	OSA	LLOC	3	5,228	122.59	52	230.02	-
	Visitor	TLOC	3	5,228	122.59	52	230.02	-
	OSA−Visitor	diff	-7	1	**0.0001**	0	**0.048**	**6.96E-01**
McCC	OSA	McCC	1	1,198	19.55	5	48.27	-
	Visitor	VG_sum	0	1,479	28.06	10	**60.35**	-
	OSA−Visitor	diff	**-299**	123	-8.50	-4	15.83	0.00E+00

4.5 Evaluation

In the previous sections we described how we collected the bug datasets. We also gave a brief overview on how they have been used in literature and how we unified them. In this section, we will continue with further, detailed evaluation of the datasets.

4.5.1 Datasets and Bug Distribution

We gathered basic statistics about the projects, which includes the number of source code elements, the number of source code elements that contains at least one bug, the percentage of source code elements that contains at least one bug, the number of total bugs in the project, the size of the systems in thousands of logical lines of code, and the number of bugs per thousand lines in a system. These tables are too large in size, thus we attached the appropriate statistics at class and file level in the downloadable Unified Bug Dataset package. There are systems in the datasets with a wide variety of sizes from 2,636 *Logical Lines of Code (LLOC)* up to 1,594,471. There are projects

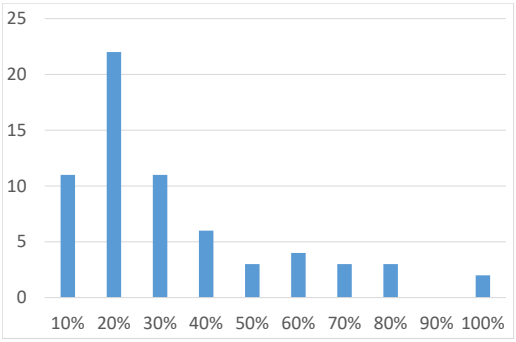

Figure 4.1. Fault distribution (classes)

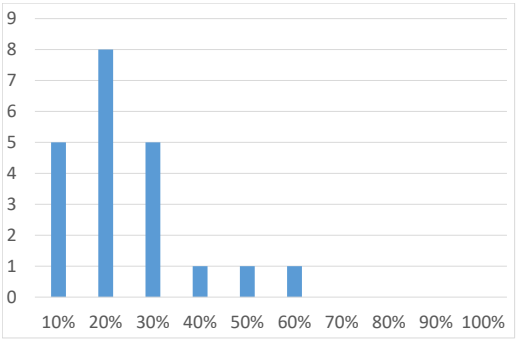

Figure 4.2. Fault distribution (files)

with very high (Xalan 2.7: 98.79%) and very low percentages (MCT: 0.48%) of buggy source code elements.

Figure 4.1 and Figure 4.2 show the distribution of the percentages of faulty source code elements for classes and files respectively. The percentages are shown horizontally. The first column means the number of systems (part of a dataset) that have between 0 and 10 percentages of their source code elements buggy (0 included, 10 excluded). In case of the systems that give bug information at class level, this number is 11 (5 at file level). In a project, usually 10%-40% of the source code elements contain at least one bug. Although there are fewer projects with file granularity, their fault distribution is similar to the class distribution, except the fact that there is no project whose files are more than 60% buggy.

4.5.2 Summary Meta Data

Table 4.12 lists some properties of the datasets, which show the circumstances of the unified dataset, rather than the data content. Our focus is on how the datasets were created, how reliable the used tools and the applied methods were. Since most of the information in the table was already described in previous sections (Analyzer, Granularity, Metrics, and Release), in this section we will only describe the Bug information row.

The Bug Prediction Dataset used the commit logs of SVN and the modification time of each file in CVS to collect co-changed files, authors and comments. Then they linked the files with bugs from Bugzilla and Jira using the bug id from the commit messages. Finally, they verified the consistency of timestamps. They filtered out inner classes and test classes.

The PROMISE dataset used Buginfo to collect whether an SVN or CVS commit is a bugfix or not. Buginfo uses regular expressions to detect commit messages that contain bug information.

The bug information of the Eclipse Bug Dataset was extracted from the CVS repository and from Bugzilla. In the first step, they identified the corrections or fixes in the version history by looking for patterns which are possible references to bug entries in Bugzilla. In the second step, they mapped the bug reports to versions using the version field of the bug report. Since the version of a bug report can change during the life

Table 4.12. Summary Meta Data

	Bug Prediction Dataset	**PROMISE**	**Eclipse Bug Dataset**	**Bugcatchers Bug Dataset**	**GitHub Bug Dataset**
Analyzer	inFusion Moose	ckjm	Visitors written for Java parser of Eclipse	Bad Smell Detector	SourceMeter
Granularity	Class	Class	File	File	Class, File
Bug information	CVS, SVN, Bugzilla, Jira	SVN, CVS	CVS, Bugzilla	CVS, SVN, Bugzilla, Jira	GitHub
Metrics	CK, process metrics	CK	Complexity, Structure of abstract syntax tree	Bad Smell	CK, Complexity, Clone, Rule violation
Release	post	pre	pre & post	pre	post

cycle of a bug they used the first version.

The Bugcatchers Bug Dataset followed the methodology of Zimmermann et al. (Eclipse Bug Dataset). They developed an Ant script that uses the SVN and CVS plugins to checkout the source code and associate each fault with a file.

In case of the GitHub bug dataset, we gathered the relevant versions to be analyzed from GitHub as we described in Chapter 3. Since GitHub can handle references between commits and issues, it was quite handy to use this information to match commits with bugs. We collected the number of bugs located in each file/class for the selected release versions (about 6 month long time intervals).

4.5.3 Functional Criteria

We evaluated the strength of bug prediction models we can build with the Weka [43] machine learning software using the dataset. For each subject software system in the Unified Bug Dataset, we created 3 versions of ARFF files (which is the default input format of Weka) for the experiments (containing only the original, only OpenStaticAnalyzer, and both set of metrics as predictors). In these files, we transformed the original bug occurrence values into two classes as follows: 0 bug $\rightarrow$ non buggy class, at least 1 bug occurrence $\rightarrow$ buggy class. Using these ARFF files we could run several tests about the effectiveness of fault prediction models built based on the dataset.

Capabilities of the original and the extended metrics suite

As described in Section 4.4, we extended the original datasets with the source code metrics of the OpenStaticAnalyzer tool and we created a unified bug dataset. We compared the bug prediction capabilities of the original metrics, the OpenStaticAnalyzer metrics, and the extended datasets. First, we handled each system individually, so we trained and tested on the same system data using ten-fold cross-validation. To build the bug prediction models, we used the J48 (C4.5 decision tree) algorithm with default parameters. We only used J48 since we did not focus on finding the best machine learning method, but we wanted rather to show a comparison of the different predictors' capabilities with this one algorithm (which has shown its power in case of the GitHub Bug Dataset). Using different machine learning algorithms (e.g. neural networks) might provide different results.

The weighted F-measure results can be seen in Table 4.13 for classes and in Table 4.14 for files. The tables contain two average values since the GitHub bug dataset used SourceMeter, which is based on OpenStaticAnalyzer to calculate the metrics. The results of OpenStaticAnalyzer and the Merged metrics would be the same. This could distort the averages, so we decided to detach this dataset from others when calculating the averages.

Results for classes need some deeper explanation for clear understanding. There are a few missing values, since there were less than 10 data entries; not enough to do the ten-fold cross-validation on (Ckjm and Forrest-0.6). The data cannot be used individually because of the lack of bug occurrences in them. The averages of the F-measure values let us conclude that there is no significant difference between the predictor sets (at least by using only J48), and there is only a slight increase when we use the original and the OSA metric together. Furthermore, there is no significant difference if we consider the systems one at a time either. However, we could achieve higher F-measure values in average in case of the GitHub Bug Dataset.

Table 4.13. Weighted F-Measure values in independent training at class level.

Dataset	**Original**	**OSA**	**Merged**
Eclipse JDT Core 3.4	0.665	0.819	0.817
Equinox 3.4	0.727	0.764	0.790
Lucene 2.4 BPD	0.891	0.878	0.879
Mylyn 3.1	0.932	0.806	0.813
PDE UI 3.4.1	0.856	0.820	0.816
Ant 1.3	0.799	0.846	0.827
Ant 1.4	0.724	0.717	0.726
Ant 1.5	0.887	0.854	0.871
Ant 1.6	0.775	0.752	0.745
Ant 1.7	0.794	0.788	0.788
Camel 1.0	0.943	0.946	0.932
Camel 1.2	0.626	0.673	0.686
Camel 1.4	0.805	0.806	0.814
Ckjm	-	-	-
Forrest 0.6	-	-	-
Forrest 0.7	0.793	0.676	0.676
Forrest 0.8	0.891	0.907	0.907
Ivy 1.4	0.897	0.915	0.899
Ivy 2.0	0.855	0.835	0.847
Jedit 3.2	0.727	0.703	0.724
Jedit 4.0	0.774	0.766	0.780
Jedit 4.1	0.747	0.778	0.781
Jedit 4.2	0.866	0.847	0.849
Jedit 4.3	0.967	0.965	0.780
Log4J 1.0	0.776	0.835	0.781
Log4J 1.1	0.739	0.743	0.849
Log4J 1.2	0.892	0.885	0.963
Lucene 2.0	0.610	0.659	0.634
Lucene 2.2	0.584	0.655	0.661
Lucene 2.4	0.698	0.656	0.663
Pbeans 1	0.813	0.769	0.813
Pbeans 2	0.727	0.686	0.740
Poi 1.5	0.743	0.753	0.782
Poi 2.0	0.851	0.841	0.831
Poi 2.5	0.787	0.809	0.789
Poi 3.0	0.768	0.768	0.764
Synapse 1.0	0.830	0.843	0.839
Synapse 1.1	0.758	0.767	0.749
Synapse 1.2	0.738	0.734	0.733
Velocity 1.4	0.827	0.824	0.856
Velocity 1.5	0.706	0.728	0.778
Velocity 1.6	0.703	0.774	0.748
Xalan 2.4	0.802	0.807	0.794
Xalan 2.5	0.654	0.693	0.699
Xalan 2.6	0.751	0.730	0.747
Xalan 2.7	0.994	0.992	0.992
Xerces 1.2	0.815	0.824	0.823
Xerces 1.3	0.860	0.837	0.836
Xerces 1.4	0.938	0.804	0.932
Average	**0.794**	**0.793**	**0.799**
Android U. I. L. 1.7.0	0.749	0.742	-
ANTLR v4 4.2	0.944	0.922	-
Broadleaf C. 3.0.10	0.898	0.881	-
Eclipse p. for Ceylon 1.1.0	0.942	0.939	-
Elasticsearch 0.90.11	0.871	0.874	-
Hazelcast 3.3	0.876	0.878	-
jUnit 4.9	0.927	0.928	-
MapDB 0.9.6	0.911	0.886	-
mcMMO 1.4.06	0.791	0.776	-
MCT 1.7b1	0.993	0.993	-
Neo4j 1.9.7	0.985	0.985	-
Netty 3.6.3	0.809	0.802	-
OrientDB 1.6.2	0.868	0.862	-
Oryx	0.889	0.898	-
Titan 0.5.1	0.922	0.926	-
Average	**0.892**	**0.886**	**-**

Table 4.14. Wighted F-Measure values in independent training at file level.

Dataset	Original	OSA	Merged
Apache Commons	0.779	0.454	0.712
Argo UML 0.26 Beta	0.899	0.832	0.880
Eclipse JDT Core 3.1	0.827	0.528	0.638
Eclipse 2.0	0.682	0.670	0.694
Eclipse 2.1	0.749	0.750	0.745
Eclipse 3.0	0.728	0.735	0.727
Average	**0.777**	**0.662**	**0.733**
Android U. I. L. 1.7.0	0.611	0.624	-
ANTLR v4 4.2	0.900	0.849	-
Broadleaf C. 3.0.10	0.862	0.823	-
Eclipse p. for Ceylon 1.1.0	0.879	0.874	-
Elasticsearch 0.90.11	0.775	0.778	-
Hazelcast 3.3	0.829	0.796	-
jUnit 4.9	0.800	0.801	-
MapDB 0.9.6	0.784	0.766	-
mcMMO 1.4.06	0.731	0.794	-
MCT 1.7b1	0.977	0.978	-
Neo4j 1.9.7	0.985	0.985	-
Netty 3.6.3	0.677	0.732	-
OrientDB 1.6.2	0.739	0.751	-
Oryx	0.771	0.861	-
Titan 0.5.1	0.894	0.894	-
Average	**0.814**	**0.820**	**-**

Results at file level are quite similar to class level results in terms of weighted F-measure. OpenStaticAnalyzer metrics did well on the GitHub Bug Dataset, but the original set of metrics are better when considering all the other file level datasets. The reason for this might be that, currently, there are only a few file level metrics provided by OpenStaticAnalyzer, and a possible contradiction in metrics can decrease the training capabilities (we saw that even LLOC values are very different).

Cross training and testing inside each dataset

As a second functional criteria, we trained a model using only one system from a dataset and tested it on all systems in the dataset. The result of the cross training is an NxN matrix where the rows and the columns of the matrix are the systems of the dataset and the value in the i^{th} row and j^{th} column shows how well the prediction model performed, which was trained on the i^{th} system and evaluated on the j^{th} one.

We used the OpenStaticAnalyzer metrics to test this criterion, but the bug occurrences are derived from the original datasets which are transformed to buggy and non buggy labels. The matrix for the whole Unified Bug Dataset would be too large to show here, thus we will only present submatrices. A submatrix for the PROMISE dataset (only one version presented for each project) can be seen in Table 4.15. The values of the matrix are weighted F-measure values, provided by the J48 algorithm. Higher F-measure values are indicated with darker colors, however, it is important to note that the coloring is done for each table individually (we introduce cross training for other datasets as well in the followings). Absolute white is the color for the lowest value (not necessarily 0.0) and the deepest gray encodes the highest value (not necessarily 1.0). Patterns are hardly noticeable, however, we can observe that there are some systems on which it is better to perform the training step. We can see matching coloring in rows that confirm this statement. For example, it is better to train the model on Log4J 1.0, than on PBeans-1. It does not match our expectations that different versions of the

Table 4.15. Cross training (PROMISE)

Project	ant 1.3	camel 1.0	ckjm 1.8	forrest 0.6	ivy 1.4	jedit 3.2	log4j 1.0	lucene 2.0	pbeans 1	poi 1.5	synapse 1.0	velocity 1.4	xalan 2.4	xerces 1.2
ant-1.3	0.967	0.830	0.776	0.776	0.785	0.784	0.794	0.754	0.718	0.706	0.674	0.660	0.659	0.596
camel-1.0	0.755	0.762	0.719	0.719	0.732	0.731	0.747	0.701	0.660	0.648	0.618	0.604	0.608	0.532
ckjm-1.8	0.307	0.442	0.488	0.488	0.481	0.482	0.496	0.496	0.500	0.503	0.507	0.514	0.511	0.505
forrest-0.6	0.767	0.741	0.697	0.697	0.712	0.710	0.728	0.681	0.637	0.623	0.587	0.573	0.576	0.496
ivy-1.4	0.807	0.756	0.714	0.714	0.733	0.735	0.748	0.707	0.671	0.658	0.622	0.610	0.613	0.540
jedit-3.2	0.733	0.746	0.719	0.718	0.728	0.751	0.750	0.721	0.696	0.691	0.678	0.665	0.664	0.589
log4j-1.0	0.753	0.731	0.722	0.722	0.733	0.734	0.741	0.718	0.703	0.696	0.677	0.672	0.671	0.627
lucene-2.0	0.750	0.718	0.696	0.696	0.697	0.698	0.700	0.685	0.672	0.671	0.665	0.655	0.656	0.620
pbeans-1	0.320	0.319	0.352	0.353	0.350	0.345	0.329	0.341	0.355	0.363	0.375	0.378	0.378	0.393
poi-1.5	0.394	0.538	0.555	0.555	0.550	0.554	0.578	0.569	0.564	0.573	0.575	0.576	0.572	0.557
synapse-1.0	0.772	0.745	0.706	0.706	0.720	0.720	0.734	0.691	0.652	0.638	0.604	0.594	0.596	0.516
velocity-1.4	0.223	0.209	0.232	0.233	0.229	0.229	0.252	0.267	0.277	0.280	0.298	0.321	0.316	0.343
xalan-2.4	0.792	0.767	0.718	0.718	0.732	0.741	0.760	0.721	0.684	0.677	0.655	0.642	0.653	0.604
xerces-1.2	0.734	0.723	0.677	0.677	0.692	0.693	0.711	0.671	0.632	0.618	0.587	0.576	0.579	0.526

same project do not show higher values (can be seen in the full attached matrix). There are also white lines in the matrix. Generally, these systems do not contain enough bug information to build efficient bug prediction models.

Table 4.16. 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

Table 4.16 shows the cross training values for the GitHub Bug Dataset. Values in the diagonal are higher in this case, as expected. Testing on Android Universal Image Loader is the weakest point in the matrix, as it is clearly visible. However, the values are not critical, the lowest value is 0.611, which can be acceptable. Elasticsearch did well in the role of a training set. This is probably because of the size of the system, the high amount of bugs, and the adequate number of entries in the dataset.

Table 4.17 shows the results of cross training for the Bug Prediction Dataset. Eclipse JDT Core passed the other systems in terms of training, which is unequivocally shown with the deep gray color in the table. Equinox performed the worst in the role of being a training set. The weighted F-measure values are relatively high, but the GitHub Bug Dataset generally has better values.

The above described results were calculated for class level datasets. Let us now consider the file level results. The three file level datasets are the GitHub, the Bugcatchers

Table 4.17. Cross training (Bug prediction dataset)

Train/Test	Eclipse JDT Core 3.4	Equinox 3.4	Lucene 2.4 BPD	Mylyn 3.1	PDE UI 3.4.1
Eclipse JDT Core 3.4	0.961	0.878	0.874	0.848	0.839
Equinox 3.4	0.451	0.570	0.614	0.664	0.641
Lucene 2.4 BPD	0.754	0.706	0.792	0.808	0.809
Mylyn 3.1	0.730	0.702	0.753	0.818	0.815
PDE UI 3.4.1	0.733	0.695	0.748	0.769	0.822

and the Eclipse bug datasets.

The GitHub Bug Dataset results can be seen in Table 4.18. As in the case of class level, the Android Universal Image Loader project performed the worst in the role of being the test set. Broadleaf, Hazelcast, and MapDB seemed to be the most powerful training sets, probably because of their size and the adequate number of bug entries. The class level dataset performed better than the file level one, which is likely because of the wider set of metrics defined for classes.

Table 4.18. Cross training (GitHub – File 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.	0.920	0.791	0.736	0.724	0.707	0.724	0.727	0.726	0.724	0.725	0.753	0.746	0.743	0.743	0.749
ANTLR v4	0.595	0.817	0.771	0.797	0.781	0.784	0.785	0.784	0.782	0.790	0.840	0.825	0.816	0.815	0.820
Broadleaf Commerce	0.631	0.832	0.836	0.842	0.803	0.803	0.803	0.803	0.800	0.806	0.848	0.835	0.826	0.825	0.829
Eclipse p. for Ceylon	0.595	0.841	0.797	0.828	0.804	0.805	0.805	0.806	0.803	0.809	0.850	0.835	0.829	0.828	0.832
Elasticsearch	0.595	0.817	0.771	0.797	0.781	0.784	0.785	0.784	0.782	0.790	0.840	0.825	0.816	0.815	0.820
Hazelcast	0.595	0.819	0.775	0.799	0.783	0.790	0.791	0.790	0.788	0.796	0.843	0.828	0.819	0.818	0.822
jUnit	0.722	0.813	0.759	0.767	0.759	0.766	0.769	0.768	0.766	0.771	0.802	0.793	0.788	0.788	0.793
MapDB	0.622	0.847	0.808	0.822	0.813	0.814	0.815	0.816	0.814	0.819	0.855	0.843	0.837	0.836	0.839
mcMMO	0.710	0.806	0.800	0.786	0.770	0.780	0.782	0.781	0.782	0.780	0.791	0.784	0.779	0.778	0.782
Mission Control T.	0.595	0.821	0.773	0.797	0.782	0.785	0.785	0.785	0.782	0.792	0.841	0.826	0.818	0.817	0.821
Neo4j	0.595	0.817	0.771	0.797	0.781	0.784	0.785	0.784	0.782	0.790	0.840	0.825	0.816	0.815	0.820
Netty	0.682	0.818	0.773	0.795	0.774	0.783	0.784	0.784	0.784	0.789	0.827	0.824	0.818	0.817	0.819
OrientDB	0.595	0.817	0.771	0.797	0.781	0.784	0.785	0.784	0.782	0.790	0.840	0.825	0.816	0.815	0.820
Oryx	0.637	0.830	0.768	0.795	0.787	0.789	0.791	0.790	0.787	0.794	0.839	0.825	0.816	0.818	0.822
Titan	0.595	0.817	0.771	0.797	0.781	0.784	0.785	0.784	0.782	0.790	0.840	0.825	0.816	0.815	0.820

The cross training results of the Bugcatchers Bug Dataset can be seen in Table 4.19. These results are lower than in the case of the GitHub Bug Dataset. ArgoUML is the system that is the easiest to evaluate on. Eclipse JDT Core performed the worst in the role of training set, however, it has better F-Measure values in the role of test data, but still a bit low. Apache Commons is the weakest as being the test data.

Table 4.19. Cross training (Bugcatchers)

Train/Test	Apache	ArgoUML	Eclipse JDT Core 3.1
Apache	0.510	0.800	0.676
ArgoUML	0.436	0.795	0.665
Eclipse JDT Core 3.1	0.388	0.234	0.308

The results of Eclipse Bug Dataset are shown in Table 4.20. The 2.0 version is the weakest one amongst the projects from the testing perspective. This is perhaps caused

by the fact that this version contains the least number of bug entries. Contrarily, the weakest model is built from version 3.0.

Table 4.20. Cross training (Eclipse)

Train/Test	Eclipse 2.0	Eclipse 2.1	Eclipse 3.0
Eclipse 2.0	0.709	0.705	0.700
Eclipse 2.1	0.638	0.712	0.725
Eclipse 3.0	0.604	0.676	0.701

We also performed a full cross-system experiment involving all systems from all datasets. This matrix is, however, too large to present here, consequently, it can be found online: `http://www.inf.u-szeged.hu/~ferenc/papers/UnifiedBugDataSet`. Minor differences can be observed for the datasets when considering the training capabilities. At class level, the GitHub and the Bug Prediction datasets performed better than the projects of the PROMISE dataset in the overall cross training. PROMISE is averaging 0.636 weighted F-measure, which is lower than the average of the GitHub Bug Dataset and the Bug Prediction Dataset with about 0.08. However, it is important to note that all the small projects are located in the PROMISE dataset and their training capabilities are limited because of their few bug entries.

Eclipse JDT Core performs the worst as a training set, and also showed poor results as testing data, but Apache Commons is even worse in this role at file level. The GitHub Bug Dataset is the most consistent dataset, only Android Universal Image Loader performed poor as a testing set. It seems that most of the projects could be used for prediction purposes, which is a good sign for the Unified Bug Dataset.

To sum up the previous findings, we can hardly tell whether the file or the class level metrics are better predictors. It may seem like file level metrics performed slightly better than class level predictors, however, we could not build sophisticated prediction models from the far more smaller metrics suite.

4.6 Threats to Validity

First of all, we accepted the collected datasets "as is", which means that we did not validate the data, we just used them to create the unified dataset and to examine the bug prediction capabilities of the different bug datasets. Since the bug datasets did not contain the source code, neither a step-by-step instruction on how to reproduce the bug datasets, we had to accept them, even if there were a few notable anomalies in them. For example, Camel 1.4 contains classes with LOC metrics of 0, or in the Bugcatchers dataset, there are two MessageChains metrics, and, in several cases, the two metric values are different.

Although, the version information was available for each system, in some cases there were notable differences between the result of OSA and the original result in the corresponding bug dataset. Even if the analyzers would parse the classes in different ways, the number of files should have been equal. If the analysis result of OSA contains the same number of elements or more, and (almost) all elements from the corresponding bug dataset could be paired, we can say that the unification is acceptable, because all elements of the bug dataset were put into the unified dataset. On the other hand, for a few systems, we could not find the proper source code version and we had to leave out a notable number of elements from the unified dataset.

Many systems were more than 10 years old when the actual Java version was 1.4 and these systems were analyzed according to that standard. The Java language has evolved a lot since then and we analyzed all systems according to the latest standard, which might have caused minor but negligible mistakes in the analysis results.

We used a heuristic method based on name matching to conjugate the elements of the datasets. Although there were cases where the conjugation was unsuccessful, we examined these cases manually and it turned out that the heuristics worked well and the cause of the problem originated from the differences of the two datasets (in Section 4.3 all cases are listed). We examined the successful conjugations as well and all of them were correct. Even though the heuristics could not handle elements having the same name during the conjugation, only a negligible amount of such cases happened.

Even when the matching heuristics worked well, the same class name could have different meanings in different datasets. For example, OpenStaticAnalyzer handles nested, local, and anonymous classes as different elements, while other datasets did not take into account such elements. Even more, the whole file was associated with its public class. This way, a bug found in a nested or inner class is associated with the public class in the bug datasets, but during the matching, this bug will be associated with the wrong element of the more detailed analysis result of OpenStaticAnalyzer.

4.7 Summary

There are several public bug datasets available in the literature, which characterize the bugs with static source code metrics. Our aim was to create one public unified bug dataset, which contains all the publicly available ones in a unified format. This dataset can provide researchers real value by offering a rich bug dataset for their new bug prediction experiments.

We considered five different public bug datasets, those are the PROMISE dataset, the Eclipse Bug Dataset, the Bug Prediction Dataset, the Bugcatchers Bug Dataset, and the GitHub Bug Dataset. We gave detailed information about each dataset, which contains, among others, their size, enumeration of the included software systems, used version control, and bug tracking systems.

We developed and executed a method on how to create the unified set of bug data, which encapsulates all the information that is available in the datasets. Different datasets use different metric suites, hence we collected the Java source code for all software systems of each dataset, and analyzed them with one particular static source code analyzer (OpenStaticAnalyzer) in order to have a common and uniform set of code metrics (bug predictors) for every system as well. We constructed the unified bug dataset from the gathered public datasets at file and class level and made this unified bug dataset publicly available to anyone for future use.

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 from Weka to build and evaluate bug prediction models per projects in the unified bug dataset. 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. Our experiments showed that the unified bug

dataset can be used effectively in bug prediction.

We encourage researchers to use this large and public unified bug dataset in their experiments and we also welcome new public bug datasets.

Part II

Methodology for Measuring Maintainability of RPG Software Systems

"The question of whether a computer can think is no more interesting than the question of whether a submarine can swim."

— Edsger W. Dijkstra

5

Brief Introduction to the RPG Programming Language

Before deep diving into different software quality assurance techniques for RPG legacy systems, a short background should be given about the RPG language itself and its basic concepts. Since legacy systems are understood by only a few, the programming languages used in them are no exceptions. As we will focus on the software quality of RPG legacy systems, we will give a brief description about the basic principles the language follows.

The RPG programming language is a popular language employed widely in IBM i mainframes nowadays. Legacy mainframe systems that evolved and survived the past decades are usually data intensive and even business critical applications. There are many legacy systems written for mainframe computers. These systems include critical elements such as bulk data processing, statistics, and transaction handling. In 1988, IBM introduced the very robust AS/400 platform, which became very popular at the end of the century (it was later renamed to IBM i). It has its own programming environment called ILE (Integrated Language Environment), which allows using programs written in ILE compatible languages, like the RPG programming language (Reporting Program Generator). Business applications developed for the IBM i platform usually use the RPG high-level programming language. In spite of the fact that the language was introduced in 1959, RPG is still widely used nowadays, and it is continuously evolving. It stands close to database systems and it is usually used for processing large amount of data. RPG is often used in the banking sector, since they do not want to change/migrate their systems because of the risk they would have to take.

Over the years, the data-intensive RPG language has evolved in many aspects. RPG III (released in 1978) already provided subroutines, modern structured constructs such as DO, DO-WHILE, IF. A great break-through was performed in 1994, when the first version of RPG IV was released. With the release of RPG IV, developers obtained new features like free-form blocks or definition specifications, and the set of available operations was extended as well. RPG supports different groups of data types, namely character data types (Character, Indicator, Graphic, UCS-2), numeric data

types (Binary Format, Float Format, Integer Format, Packed-Decimal Format, Unsigned Format, Zoned-Decimal Format), date, time and timestamp data types, object data type (user can even define Java objects), and one can define pointers to variables, procedures and programs. In RPG IV there are two types of functions. The first one is called subroutine which takes no parameter and has a limited visibility and usability. Procedures are more flexible constructs since they can have parameters and they can be called from other programs.

Listing 5.1. RPG IV sample code

```
....+....1....+....2....+....3....+....4....+....5....+..
     H DATEDIT(*DMY/)
     FEMPLOYEE  IF   E           K DISK
     D Pay             S               8P 2
      * Prototype Definition for subprocedure CalcPay
     D CalcPay         PR              8P 2
     D   Rate                          5P 2 VALUE
     D   Hours                        10U 0 VALUE
     D   Bonus                         5P 2 VALUE
      /free
        chain trn_number emp_rec;
        if %found(emp_rec);
          pay = CalcPay (emp_rate: trn_hours: trn_bonus);
        endif;
      /end-free
```

Listing 5.1 shows a sample RPG IV code fragment. The code snippet presents some features and capabilities of the RPG language. From the beginning, RPG uses the column dependent source format. In this way, programming requires the code elements to be placed in particular columns. For instance, the character in the 6th column means the specification type. In Listing 5.1, there are Control (H), File (F), and Definition (D) specifications. Calculation specifications (column-dependent) can be added to perform an appropriate sequence of operations. This specification is the one in which most of the processing is done. Since RPG IV, it is possible to use the free-form blocks for column-independent calculations. In this sample, payment is calculated by a procedure named CalcPay (prototype declaration can be found in the sample, but the definition is not presented here). In this chapter, our point is not to present all RPG language capabilities, but just to show the reader what it looks like. For details, see the ILE RPG Programmer's Guide on IBM's website [1].

In this thesis point, we will focus on the analysis and the maintainability of RPG programs. At this point, the reader might ask, why anybody would want to do something like this. This is not a made up topic; it is based on true industrial needs. We managed to address various RPG related research topics with the support of the Hungarian national grant GOP-1.1.1-11-2012-0323. In this grant, we cooperated with the R&R Software Ltd. who was the one with the industrial need of analysis on RPG software systems.

[1] https://tinyurl.com/yctgq6ax

"The most important property of a program is whether it accomplishes the intention of its user."

— C.A.R. Hoare

6

Evaluation of Existing Static Analysis Tools

6.1 Overview

Rapid development life cycles provided by 4GL languages resulted in a number of large software systems decades ago, that are mostly considered legacy systems nowadays. On the other hand, the role of quality assurance of these data intensive and often business critical systems is increasingly important. The IBM i platform – initially called AS/400 platform – became very popular towards the end of the last century. Business applications developed for the IBM i platform usually use the RPG high-level programming language (Reporting Program Generator), which is still widely employed, supported and evolving. In the early days of the appearance of 4GL (like RPG), several studies were published in favour of their use. The topics of these studies are mostly focused on predicting the size of a 4GL project and its development effort, for instance by calculating function points [104] or by combining 4GL metrics with metrics for database systems [85]. In the literature, only a few papers are available considering the software quality of these languages [63, 48, 79], while the main focus of current QA tools and techniques is on the more popular object-oriented languages.

In this chapter, we compare two state-of-the-art tools for RPG quality measurements by analyzing the capabilities of static analyzers. Several measurable aspects of the source code may affect higher level quality attributes, however, this comparison is based on five important aspects: analysis success, analysis depth, source code metrics, coding rule violations, and code duplications. The toolchain named SourceMeter for RPG is our product and we wanted to have an exhaustive comparison with other RPG analyzers. The quality of the static analyzer is key for further investigations, thus we needed to find the best tool available for that purpose.

This chapter is organized as follows. Related research is outlined in Section 6.2. Section 6.3 introduces our subject analyzer tools capable of measuring RPG quality. In depth comparison of the tools in terms of source code metrics, coding rule violations and clones is presented in Section 6.4, while the findings are discussed in Section 6.5. Finally, we conclude our work and list some threats to validity.

6.2 Related Work

Numerous studies have been published in the last decades focusing on different software metrics. Chidamber and Kemerer introduced a number of object oriented metric definitions [29]. Basili et al. validated these metrics by applying them on early software defect prediction [16]. A revalidation was done by Gyimothy et al. [42] to present fault prediction technique results of the open source Web and e-mail suite called Mozilla. Despite the fact that RPG is not located in OO domain, these studies are cornerstones for further investigations on software metrics.

At present, RPG and other early programming languages like COBOL are used by a narrowed set of developers since RPG programs cannot be run on personal computers (only via remote connection) and mainly newer languages are tutored, such as Java, JavaScript, Python, etc. Thus, much effort was put into researches dealing with effective migration mechanisms to transform RPG legacy programs into an object oriented environment, however there is no golden hammer or universal tool that can handle all the migration tasks and problems. A smaller step is presented by Canfora et al. that transforms RPG II and RPG III into RPG IV programs [27]. A migration technology was also proposed to handle COBOL legacy systems as Web-based applications [38] by applying wrapping techniques on them.

Research papers dealing with software metrics are commonly applied on widely used programming languages like C, C++ [108, 39], Java [26], C# [65, 50]. The Magic 4GL language has similar attributes to RPG, with similar need for quality assurance solutions ([78, 77]). Only a few studies focus on software metrics specialized for RPG. Hartman focused on McCabe and Halstead metrics [48] because of their usefulness in identifying modules containing a high number of errors. Another early research paper also focuses on the characteristics of programs written in RPG [79]. Naib conducted an experiment using environmental (varying with time) and internal (McCabe, Halstead, LOC that do not vary with time) factors and constructed a regression model to predict errors in the given systems. Bakker and Hirdes analyzed mainly legacy systems with more than 10 million lines of code written in COBOL, PL/I, C, C++, and RPG (1.8 million lines of code). They found that maintenance problems highly correlate with design issues. They recommended to re-design and re-structure rather than re-build applications since it is better worth it. Further maintenance difficulties, including improvement and extension, can occur, so a flowchart extraction method was made by Suntiparakoo and Limpiyakorn [97] that can serve as a quality assurance item supporting the understanding of RPG legacy code in maintenance processes. One can see that many approaches use software metrics as a low level component to produce or model a higher level characteristic (e.g fault-prone modules) describing a given system. Low level metrics can be applied for a wide variety of software quality purposes such as using quality models to characterize whole systems (often includes benchmarking). Different models have been proposed based on ISO/IEC 25010 [54], and its ancestor called ISO/IEC 9126 [53] to serve these purposes.

Due to focusing on high level characteristics, the above mentioned studies pay little attention to different low level software metrics and rules. In the following sections we will propose two state-of-the-art RPG static source code analyzers and compare their functionalities from the perspective of the users.

6.3 RPG Program Analyzer Tools

In this section, we provide in depth comparison of two software products for quality centric static analysis of RPG programs. Source code based quality measurements usually consider several aspects of the code, from which the most popular ones are architecture & design, comments, coding rules, potential bugs, complexity, duplications, and unit tests. The RPG language is not provided with such an extensive free tool support as in the case of object-oriented languages. In our comparison, we selected two recently announced and partially free / low cost software quality tools: SourceMeter for RPG version 7.0 and SonarQube RPG (version 4.5.4). Although the categorization of quality attributes are different in these tools, we found them comparable, as the results of the SourceMeter toolchain are integrated into the SonarQube framework.

6.3.1 SourceMeter for RPG

SourceMeter [95] is an innovative tool built for the precise static source code analysis of projects implemented in languages like Java, C/C++, Python or RPG [63]. This tool makes it possible to find the weak spots of a system under development from the source code itself without the need for simulating live conditions.

SourceMeter can analyze source code conforming to RPG/400 and RPG IV versions, including free-form as well. The input of the analysis can be specified as a raw source code file or a compiler listing. In case of using raw source code as an input, the analyzer could not calculate some code metrics, and detect various rule violations because the raw source contains less information than the compiler listing (for instance, cross references are detected using compiler listing entries). As it is recommended, we used compiler listing inputs in our work. For constructing RPG compiler listing files, we use RPG compiler with version V6R1M0.

SourceMeter is essentially a command line toolchain to flexibly produce raw results of static analysis. Visualization and further processing of these results are done in other tools like the QualityGate [87] software quality management platform and the SourceMeter plugin to integrate data into the SonarQube framework.

6.3.2 SonarQube RPG

SonarQube [94] is an open source quality management platform with several extensibility possibilities. In this platform, the concrete static analyzers of various programming languages are implemented as plugins as well. As it supports several languages, the depth and type of analysis results depend on the actual toolchain. The main starting point of the user interface is the so called Dashboard, however the interface can also be highly extended and customized. Figure 6.1 shows the SonarQube RPG dashboard, where all aspects of quality are represented. The SonarQube RPG analyzer is a commercial plugin, however, trial licence is available. The plugin supports the RPG IV language. However, no possibility is present to perform an analysis on RPG III (RPG/400) programs or to handle free-form code blocks in RPG IV.

6.3.3 SourceMeter for SonarQube Plugin

SourceMeter is bundled with a free SonarQube RPG analyzer plugin. The plugin conforms to the analysis process of the SonarQube and provides necessary data for

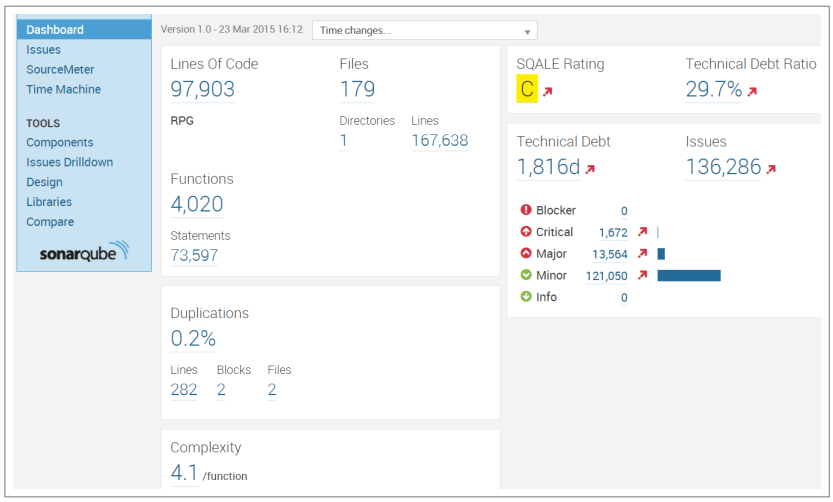

Figure 6.1. SonarQube dashboard

the integration. This way, analysis results (metrics, code clones, rule violations) of SourceMeter can be re-used and the SonarQube framework provides the user interface and the management layer of quality data.

In Figure 6.2, the dashboard can be seen with an analysis result done by SourceMeter. Results are different from the ones shown in Figure 6.1, for example, more coding rules and clones are found by the SourceMeter. In addition, there are several additional metrics, which are not presented in the dashboard, but can be found in the detailed views of SonarQube. The plugin provides, among others, a SourceMeter menu item with custom dashboard and an extended source code view with a metrics panel showing hands-on metric information next to the actual code element as shown in Figure 6.3.

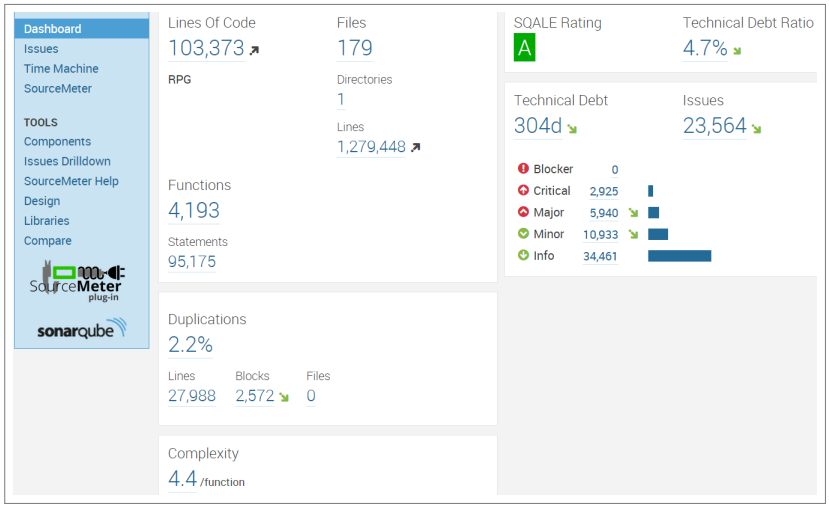

Figure 6.2. SourceMeter for RPG SonarQube plugin dashboard

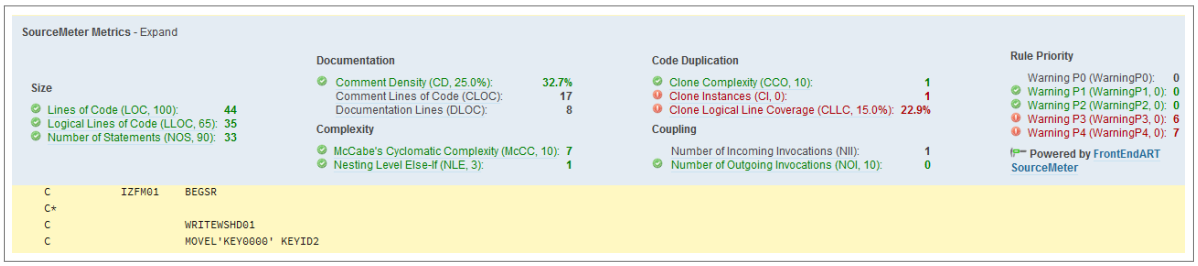

Figure 6.3. SourceMeter source code view with Metrics panel integrated in SonarQube

6.4 Comparative evaluation

We conducted experiments using 179 RPG programs containing around 100k lines of code. These programs belong to a software development company specialized for IBM i mainframe applications. While these programs are considered typical in their purpose, they are influenced by the coding style and the tradition of the company.

6.4.1 Comparison of Source Code Metrics

The SourceMeter tool provides a large variety of metrics at four levels of program elements: system, program, procedure, and subroutine levels. The SonarQube model is restricted to file level metrics, which we treat as program level metrics. In addition, system level summary is also available. On the other hand, the extensibility mechanism of SonarQube makes it possible to incorporate additional metrics into the user interface (as shown in Figure 6.3).

Table 6.1. System level metric values

	Files	LOC	Functions	Statements	Duplications	Complexity
SonarQube	179	97,903	4,020	73,597	0.2%	16,667
SourceMeter	179	103,373	4,193	95,175	2.2%	18,296
Difference	0	5,470	173	21,578	-	1,629
3 files	0	5,289	173	5054	-	1,113
Abs. Diff.	0	2 (181-179)	0	16524	-	516

Table 6.1 shows the high level metric values of the analyzed system, which metrics are available in both tools. Each tool is able to calculate the number of files, lines of code, number of functions, number of statements, the percentage of duplicated code portion, and the global complexity. Considering the indicated values, one can see that many of them are not the same. Further investigations showed that SonarQube could not analyze three files, thus the metric values are also not calculated and aggregated. Metric values that are calculated for these three files by SourceMeter is also shown in the table. SourceMeter counts the last empty line into the LOC metric, so absolute difference can be caused by the distinct calculating methods used by each tool, moreover, it is based on there being no previous unified baselines when dealing with software metrics related to RPG programming language. Different operations can be taken into consideration when calculating complexity or number of statements.

We summarized the available metrics of both tools in Table 6.2. SourceMeter definitely operates with a more comprehensive set of metrics. SourceMeter handles subroutines as basic code elements and propagates the calculated metric values to

Table 6.2. Defined Metrics

Level	Category	SourceMeter for RPG	SonarQube RPG
System	Coupling	TNF	TNF
	Documentation	TCD, TCLOC, TDLOC	-
	Complexity	-	McCC
	Size	TLLOC, TLOC, TNOS, TNPC, TNPG, TNSR, TNNC, TNDS	TNOS, TNSR, TLOC, TLLOC
Program/File	Coupling	TNOI, NF, TNF, NIR, NOR	-
	Documentation	CD, CLOC, DLOC, TCD, TCLOC, TDLOC	CLOC, CD
	Complexity	NL, NLE	McCC
	Size	LLOC, LOC, NOS, NUMPAR, TLLOC, TLOC, TNOS, TNPC, TNSR, NNC, TNNC, NDS, TNDS	TNSR, TNOS,LOC, LLOC
Procedure	Coupling	NOI, TNOI, NF	-
	Documentation	CD, CLOC, DLOC, TCD, TCLOC, TDLOC	-
	Complexity	McCC, NL, NLE	-
	Size	LLOC, LOC, NOS, NUMPAR, TLLOC, TLOC, TNOS, TNSR, NNC, NDS	-
Subroutine	Coupling	NII, NOI	-
	Documentation	CD, CLOC, DLOC	-
	Complexity	McCC, NL, NLE	-
	Size	LLOC, LOC, NOS	-

higher levels (procedures and programs can contain subroutines). SonarQube focuses only on file (program) and system levels and also works with a narrowed set of metrics. For detailed descriptions of the computed metrics we refer to the websites and user's guides of the tools.

6.4.2 Comparison of Coding Rules

The lists of coding rules of the two analysis tools have a significant common part. Figure 6.4 shows the distribution of coding rules between each of the following categories: common rules checked by both tools, SourceMeter-only rules, SonarQube-only rules. SourceMeter also provides a set of rules for validating metric values by specifying an upper and/or lower bound for each metric shown in Table 6.2. Precisely set values can help developers focus on code segments that are possible weak spots. SonarQube does not support rules like this. Many rules are implemented in both tools (31% $\approx$ 30 rules), which confirms that a similar set of rules are considered important by the developers of each tool.

Table 6.3 shows a comparison of the implemented rules in both tools. Based on the different implementation, many rule violation occurrence numbers are not equal. In the following sections, we mainly wanted to focus on the rule violation occurrence values that differ. The rule dealing with comment density is not the same in these tools, since SourceMeter desires comment lines after x lines, where x is an upper threshold, on the

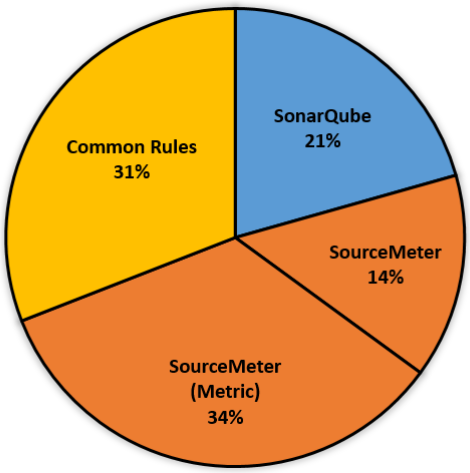

Figure 6.4. Distribution of common and unique coding rules

contrary, SonarQube only examines the Comment density metric (CD) of a program. None of the tools found any subroutines without documentation. The reason for this is that the RPG sources are generated from compiler listing files that contain comments, since the compiler automatically places comments before subroutines. In the given source files, no naming convention was applied on subroutine names, so SourceMeter detects all the 4193 subroutines (found), except the one whose name starts with SR which is expected by the rule. However, the list contains the *INZSR subroutines (172), which is not correct, since the initialization subroutine must be named exactly like that and should be skipped. The remaining 173 rule violations come from the three unanalyzed files. Copyright checks are not sufficient by SonarQube (found no violation), contrary to SourceMeter that found 28 cases when copyright is not located in the source code. Some random cases were validated manually and SourceMeter triggers correctly. Nesting different control flow statements – like do, do-while, if, select – too deeply may result in complexity problems. SonarQube located 264 deep nesting cases, while SourceMeter detected 352. A possible reason for this can be the different parameter setting for the maximum nesting level. SourceMeter should use a better default parameter for subroutine complexity since it detects numerous subroutines with high complexity. "/EJECT" compiler directive should be used after F, D, and C specification sections. We empirically validated the fact that SonarQube does not detect all the possibilities (after C specifications, it does not require an /EJECT directive). When dealing with unused subroutines, SonarQube counts the initialization subroutine as one of the never called ones, although it is called automatically (there are cases when explicit call is used). SonarQube detects commented out code sections, but SourceMeter locates commented out statements. SourceMeter explores avoid ("testn" – occurred 4 times) and forbidden ("leave" operation – occurred once) operations (only the priority differs) and does not use a particular rule only for GOTO operation. SonarQube handles all occurrences of '0' or '1' as a possible rule violation and asks the developer to change it for *ON or *OFF (not only for indicators that causes many false positive violations). SourceMeter requires the presence of *NODEBUGIO option in the header as well, not only the *SRCSTMT keyword. Missing error handling rule violations differ only because of the three unanalyzed files. "Code blocks (IF,

Table 6.3. Rules implemented in both tools

Group by SonarQube	SC Occ.	SonarQube Rule Description	SM Occ.	Group by SourceMeter
	0	Source files should have a sufficient density of comment lines	185	Documentation
	0	Subroutines should be documented	0	Documentation
convention	3847	Subroutine names should comply with a naming convention	4192	Naming
	0	Copyright and license headers should be defined	28	Security
	0	"E" (externally described) indicator should be found in F spec lines	0	Design
	7	Numeric fields should be defined as odd length packed fields	7	Design
brain-overload	264	Control flow statements "IF", "FOR", "DO", ... should not be nested too deeply	352	Design
brain-overload	31	Subroutines should not be too complex	1,454	Design
	1	Line count data should be retrieved from the file information data structure	1	Design
convention	334	"/EJECT" should be used after "F", "D" and "C" specification sections	520	Basic
	305	The first parameter of a "CHAIN/READx" statement should be a "KLIST"	315	Design
unused	227	Unused subroutines should be removed	80	Unused Code
unused	130	Sections of code should not be "commented out"	185	Unused Code
	0	Certain operation codes should not be used	4 + 1	Basic
brain-overload	0	"GOTO" statement should not be used	0	Basic
cwe, security	0	Debugging statements "DEBUG(*YES)" and "DUMP" should not be used	0	Basic
	0	The correct "ENDxx" statement should always be used	0	Basic
	0	"IF" statements should not be conditioned on Indicators	0	Basic
cwe	0	All opened "USROPN" files should be explicitly closed	0	Basic
	3	An indicator should be used on a "CHAIN" statement	1	Basic
	1111	Standard figurative constants *ON, *OFF and *BLANK should be used in place of '1', '0' and ' '	17	Basic
	0	The "*SRCSTMT" header option should be used	4	Basic
error-handling	699	Error handling should be defined in F spec	749	Basic
cert	0	"IF ELSEIF" constructs shall be terminated with an "ELSE" clause	0	Basic
brain-overload	643	"WHEN" clauses should not have too many lines	308	Size
brain-overload	58	Files should not have too many lines	120	Size
brain-overload	55	"DO" blocks should not have too many lines	17	Size
brain-overload	145	"IF" blocks should not have too many lines	151	Size
	0	"/COPY" should be avoided	1	Design
brain-overload	0	Subroutines should not have too many lines	0	Size

DO, Files, WHEN clauses) containing too many lines" rules possibly have different occurrence values, since the default parameter differs. SourceMeter has a similar rule for limiting the usage of the /COPY compiler directive, but it operates with a nesting level limit (currently one level of copy is allowed). However, SonarQube does not detect the forbidden copy operations.

Table 6.4 presents the SourceMeter-only rules. Rules can be found like subroutine circular call detection, different naming conventions, constantly false conditional state-

Table 6.4. Rules implemented only in SourceMeter (without metrics-based rules)

SourceMeter Rule Description	Group by SourceMeter	Occ.
Uncommented conditional operation	Documentation	4,629
File uses prefixed name	Naming	0
Too short name	Naming	22
Too long name	Naming	271
Character variable names should begin with '$'.	Naming	0
Numeric variable names should begin with '#'.	Naming	0
Lower case letter in the name of called program or procedure	Naming	0
Large static array	Design	33
Circular reference between subroutines	Design	1
Variable only referenced from an unused subroutine	Unused Code	18
Conditional expression is always false	Unused Code	1
Numeric operands of MOVE(L) are not compatible	Type	2
Call operand is a variable	Basic	3
Complete Conditional Operation Needed	Basic	179

ments (like 1 equals to 2). A bad programming practice is when a variable is given as an operand of call operation since it makes the debugging process more difficult.

Table 6.5 shows the list of SonarQube-only rules and the number of triggers. Some rules have a very high trigger value. Uppercase form was not used in 107,768 cases that can seriously distort the technical dept information (however, any rule can be turned on or off).

Table 6.5. Rules implemented only in SonarQube

SonarQube Rule Description	Group by SonarQube	Occ.
Variables used in only one subprocedure should not be global	pitfall	0
"/COPY" statements should include specification letters	convention	185
"CONST" should be used for parameters that are not modified		2
Columns to be read with a SELECT statement should be clearly defined	sql	0
Comment lines should not be too long	convention	9,891
Expressions should not be too complex	brain-overload	86
LIKE keyword should be used to define work fields		703
Nested blocks of code should not be left empty	bug	32
Operation codes and reserved words should be in upper case	convention	107,768
Prototypes should be used	convention, obsolete	1,423
Record formats should be cleared before each use	bug	973
Source files should not have any duplicated blocks		2
SQL statements should not join too many tables	performance, sql	0
Subprocedures should be used instead of subroutines	obsolete	4,019
Subprocedures should not reference global variables	brain-overload	0
The data area structure for "IN" should be defined in D spec lines.		148
Parameters of "CALL"/"CALLB" statements should be defined as "PLIST"		68
Non-input files should be accessed with the no lock option		0
Unused variables should be removed	unused	14
String literals should not be duplicated		2872

6.4.3 Comparison of Duplicated Code Sections

SonarQube contains a rule for noting suspicious duplicated code sections. Sonar can show duplicated lines in the source files, however, no grouping can be obtained that makes it hard to understand code clones. Sonar only deals with Type-1 clones that means the traditional copy-paste programming habit, so, every character must be the

same in the clone instances. A clone class encapsulates the same code portions (clone instances) from different source locations into a group. SonarCube considered 0.2% of the whole RPG code as code duplication (2 duplicated section with 141 lines). SourceMeter has a poor display technique in Sonar environment, namely no highlighting on affected lines is done. In a different context, SourceMeter supports a kind of well-defined format for marking various clone classes and the relevant code instances. The tool is also capable of finding Type-2 clones (e.g variable names may differ) that is confirmed by the found 2.2% of code fragment that play a role in code duplications. Its clone detection algorithm tries to match similar code sections (syntax-based) based on source code elements (subroutine, procedure). On the contrary, SonarQube only uses textual similarities to detect clones, but no structural information is used in clone detection. For example, clone instances containing one and a half subroutines may be produced, however, they should be split into two clone instances (holds more information when considering refactoring). Another advantage of SourceMeter is that it accepts parameters such as the minimum lines of code contained by a clone instance.

While SonarQube shows duplicated code locally in the inspected program, SourceMeter extends its capabilities with a separate code duplication view, where clone instances belonging to the same clone class can be investigated easily.

6.5 Discussion

In this section, we are summarizing and concluding our results. Then we show some insights about the quality indices in terms of the previously presented aspects (metrics, coding rule violations and code clones).

6.5.1 Summary of results

Analysis success and depth The program analysis went almost without problems with both tools. While SourceMeter successfully analyzed all source files, SonarQube RPG failed to analyze three of them. Although this is not considered as a blocker problem in its use. On the other hand, SonarQube works at file level, while SourceMeter analyzer works at finer levels of details (like procedure, subroutine level), which provides a more detailed view of the analyzed system.

Source code metrics SourceMeter provides a wider range of metrics, and works even at procedure and subroutine levels. SonarQube provides a limited set of metrics, which restricts the building of further higher level models on top of low level metrics.

Coding rules The set of common or similar coding rules is large in size. SonarQube has slightly more unique rules implemented, but SourceMeter provides a wide set for validating metric value based rules. Generally, the tools are balanced in functionality.

Code duplications SourceMeter found significantly more duplicated code fragments with better granularity. SourceMeter detects Type-2 clones (syntax-based), SonarQube only deals with copy-pasted clones. SourceMeter extends SonarQube with an improved display of code clones.

Table 6.6. 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

Table 6.6 summarizes our findings with a short explanation of the result of our experiments. During the comparison of our subject tools, we experienced that coding rules for the RPG language in general need to be evolved, compared to similar solutions of other popular languages. Given that both tools appeared recently on the market, we foresee extended versions in the future.

6.5.2 Effect on Quality Indices

Low level, measurable attributes, such as code metrics, rule violations and code duplications contribute to higher level code quality indices. Such quality indices give an overall picture of the analyzed project, helping stakeholders to take actions in case of low or decreasing quality. SonarQube operates with two concepts to assess higher level quality: technical debt and SQALE rating.

Technical debt is a metaphor for doing things in a quick but dirty way, which makes future maintenance harder. If the debt is not paid back (e.g. software quality is not considered as an important aim in the development process), it will keep accumulating interest – similarly to a financial debt. In case of SonarQube, the technical debt is measured purely based on coding rule violations. Each coding rule has an estimated time to correct it. The overall technical debt is the sum of the estimated correction time of all rule violation instances. The SQALE rating is based on the technical debt, as such, it is based on coding rules as well. Hence, other quality attributes, like various metrics (e.g. complexity, coupling) and code duplications do not affect these quality indices. We provide dashboard data of quality indices computed in case of all rules checked (Figure 6.5) and the dashboard for an analysis when only the common rules were active (Figure 6.6). On the other hand, we recommend quality model that rely on more quality attributes, like the QualityGate [87] models.

6.5.3 Threats to Validity

We identified a few threats to validity in our study. The validation of the results was done manually on selected metrics/rules. The initial plan was to export the whole list of rule violations and filter, at least the common results, automatically. While SourceMeter is a command line tool-chain that can produces csv outputs, we did not manage to obtain the full list from SonarQube. It is possible to obtain a report from SonarQube, but that is not a complete list of rule violations. Although exhaustive manual validation is not feasible, the current study involves three aspects of quality measurements. We believe these three aspects are of high importance (technical debt

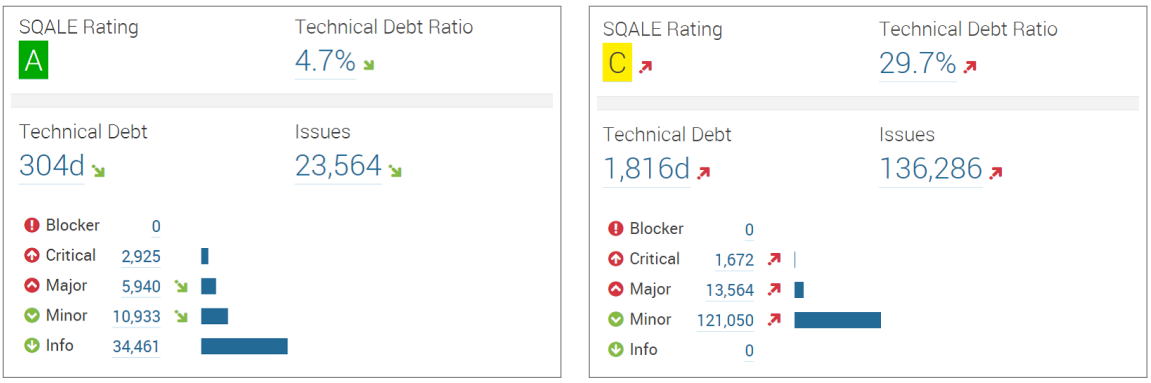

Figure 6.5. Quality indexes based on SourceMeter for RPG analyzer (left) and SonarQube RPG analyzer (right) – computed using all coding rules

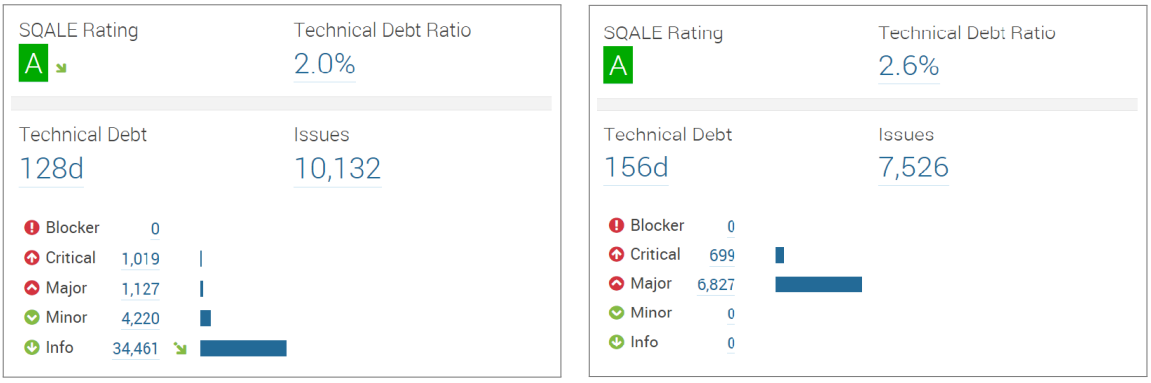

Figure 6.6. Quality indexes based on SourceMeter for RPG analyzer (left) and SonarQube RPG analyzer (right) – computed using common coding rules only

is computed based on only one aspect), however, adding other viewpoints or even dynamic analysis results would increase the validity of the results. The measured RPG programs belong to the same domain and are implemented by developers of the same software house who followed the coding policies of the company. Further experiments are needed with larger RPG codebase from various domains and developers. Although we identified this threat, we note that the measured RPG programs are part of legacy, data intensive applications typical in IBM i mainframes.

6.6 Summary

In this chapter, we experimented with the static analyzers of quality management tools for the RPG programming language employed on the IBM i mainframe. We compared the SourceMeter for RPG command line toolchain together with its SonarQube plugin to the RPG analyzer of the SonarQube framework. Five important aspects of quality measurements were examined: analysis success, analysis depth, source code metrics, coding rules, and code duplications. SonarQube can not handle some source files, moreover, the depth of analysis is limited to system and file level. SourceMeter can perform analysis in finer granularity (procedures, subroutines). We found that, from the metrics point of view, the SourceMeter tool provides a much wider range of possibilities, while the handling of coding rules is balanced, since the common set of coding rules is relatively large. SonarQube detects clones using copy-paste (Type-1) clones,

SourceMeter can detect Type-2 clones since it uses a syntax-based mechanism and also takes into consideration the bounds of source code elements (subroutines, procedures).

Technical Dept and SQALE rating are higher level quality attributes provided by the SonarQube platform. Unfortunately, these quality metrics only based on the coding rule violations, which can distort the overall quality of the system. We suggest and will present possible solutions for this problem in the following chapters in which we will use SourceMeter for the purpose of calculating low level quality attributes.

"Before software can be reusable it first has to be usable."

— Ralph Johnson

7

Integrating Continuous Quality Monitoring Into Existing Workflow – A Case Study

7.1 Overview

The IBM i mainframe was designed to manage business applications for which reliability and quality are matters of national security. The RPG programming language is the most frequently used one on this platform. The maintainability of the source code has a big influence on the development costs, which is probably the reason why it is one of the most attractive, observed and evaluated quality characteristic of all. For improving, or at least preserving, the maintainability level of software, it is necessary to evaluate it regularly. In this chapter, we present a quality model based on the ISO/IEC 25010 international standard for evaluating the maintainability of software systems written in RPG. As an evaluation step of the quality model, we show a case study in which we explain how we integrated the quality model as a continuous quality monitoring tool into the business processes of a mid-size software company, which has more than twenty years of experience in developing RPG applications.

Our main contribution in this work is the introduction of a continuous quality management approach specialized to RPG. According to our best knowledge, no or only very little effort was invested into researching state-of-the-art quality assurance techniques for RPG. To explore RPG source code and detect components which carry high risks, we used the SourceMeter for RPG static code analyzer tool, which seemed to have a deeper analysis capability as shown in Chapter 6. As we have presented, the analyzer provides three types of measurements, which are the following:

- *Software Metrics:* A software metric in this context is a measure of some property of the source code [29]. There is a growing need in software development to define quantitative and objective measurements, so software metrics are calculated for RPG language elements (namely for subroutines, procedures, programs, and for the whole system).

- *Rule violations:* Rule violations [21] can reveal code segments that are prone to errors. These can be coding problems that are introduced e.g. accidentally or because of low-skill programmers, and code smells that can be symptoms in the source code that possibly mean a deeper design problem in the code and can cause incorrect operation at a later stage. Code smells are not equal to bugs, but their presence increases the risk of bugs or failures in the future. Rules are defined for RPG to indicate the existence of possible code smells. Furthermore, coding rules are excellent means for defining company coding standards.

- *Code duplications:* Duplicated code segments usually come from the very productive, but at the same time dangerous source code copy-paste habit. Code duplications [18] could be handled as a code smell; however, they cover an important separable part of quality assurance.

We used these low-level measurements as an input to provide a diagnosis about an RPG system. This diagnosis, which characterizes the quality of a whole RPG system, is provided by the ISO/IEC 25010 [54] standard based, new quality model introduced in this chapter. Building and testing a quality model like this needs specialists in this specific domain. So, the company called R&R Software[1] (mid-sized company with more than 100 software developers) was involved to help the calibration and evaluation of the quality model with their deep knowledge in developing software systems written in RPG. They clarified the importance of each quality aspect, and also provided a collection of industrial programs written in RPG. As a validation of our approach, we show in a case study how we managed to integrate our method into their development processes, and how they used it for refactoring purposes.

The chapter is organized as follows. In the next section, we summarize some related studies. Next, we briefly describe our approach in Section 7.3. Afterwards, in Section 7.4 we present a case study, which shows how our approach works in a real life situation. In Section 7.5, we collect some limitations and threats to validity. Finally, in Section 7.6 we sum up our findings.

7.2 Related work

In this section we provide an overview about the most important studies about software quality measurement and its relationship with RPG static code analysis.

7.2.1 Quality Assurance for RPG

Papers focusing on legacy systems and/or RPG quality analysis are rarely published, however, there are some notable research papers presented during the decades. Kan et al. [60] were dealing with software quality management in AS/400 environment. They identified the key elements of the quality management method, such as customer satisfaction, product quality, continuous process improvement and people. Based on empirical data, the progress in several quality parameters of the AS/400 software system were examined. They presented a quality action road map that describes the various quality actions that were deployed.

[1] http://www.rrsoftware.hu/

Bakker and Hirdes [8] described some project experiences using software product metrics. They analyzed more than 10 million lines of code in COBOL, PL/I, Pascal, C, C++, and RPG (about 1.8 million lines of code). The goals of the projects written in RPG were maintenance and risk analysis. They found that the problems in legacy systems are caused by the design, not by the structure of the code. Furthermore, re-designing and re-structuring existing systems are less costly and safer solutions to these problems than re-building.

7.2.2 Quality Model

Once we have quality indicators, like software metrics, which are capable to characterize the software quality from various points of view, it is necessary to standardize this information somehow [4]. This is the reason why the ISO/IEC 25010 [54], and its ancestor, the ISO/IEC 9126 [53] international standards have been created. The research community reacted quickly to the appearance of these standards, and several papers have been published in connection to them. Jung and Kim [56] validated the structure of the ISO/IEC 9126 standard based on the answers of a widespread survey. They focused on the connections between the subcharacteristics and the characteristics. They grouped subcharacteristics together based on the high correlation in their values according to the evaluators. The authors found that most of the evolved groups referred to a characteristic defined by the standard.

Since the standards do not provide details about how these characteristics can be determined, numerous adaptations and various solutions have been developed for calculating and assessing the values of the defined properties in the standard [9, 7, 13].

Many research papers use benchmarking and quality models to provide higher level information about software systems. Benchmarking in software engineering is proven to be an effective technique to determine how good a metric value is. Alves et al. [5] presented a method to determine the threshold values more precisely based on a benchmark repository [32] holding the analysis results of other systems. The model has a calibration phase [7] for tuning the threshold values of the quality model in such a way that for each lowest level quality attribute they get a desired symmetrical distribution. They used the $\langle 5, 30, 30, 30, 5 \rangle$ percentage-wise distributions over 5 levels of quality. To convert the software metrics into quality indices we also used a benchmark with a large amount of evaluations, but we applied it in a different way. During the calibration, instead of calculating threshold values we approximate a normal distribution function called benchmark characteristic, which is used to determine how good a software is.

7.3 Approach

In this section, we provide a brief overview about the probabilistic software quality model called ColumbusQM [9] and show how we implemented the general approach for the RPG language.

The ISO/IEC 25010 international standard defines the product quality characteristics that are widely accepted both by industrial experts and academic researchers. These characteristics are: functional suitability, performance efficiency, compatibility, usability, reliability, security, maintainability and portability. In this study, we focused on maintainability because of its obvious and direct connection with the costs of altering the behavior of the software. The standard defines the subcharacteristics

of maintainability as well, but it does not provide further details on how we could calculate these characteristics and subcharacteristics.

The basic idea behind the ColumbusQM is splitting the complex problem of calculating a high-level quality characteristic into less complex sub-problems. In the quality model, the relations between the lower level metrics, which can be readily obtained from the source code, and the higher level quality characteristics can be described with an acyclic directed graph, called the *attribute dependency graph (ADG)*. The developed ADG for RPG language is shown in Figure 7.1.

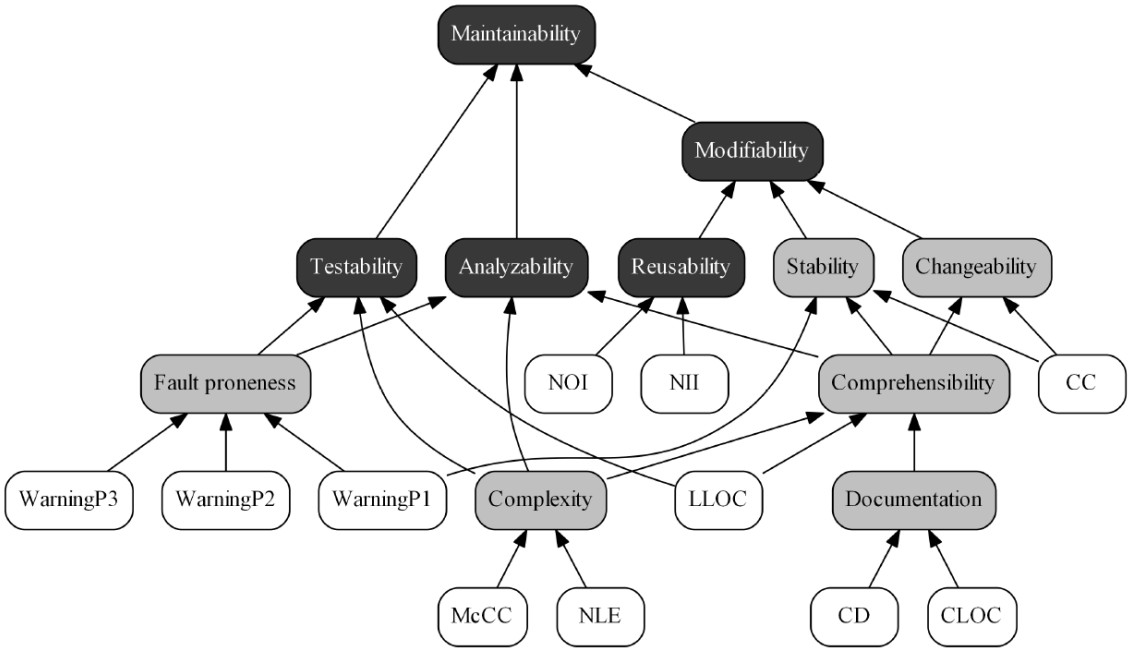

Figure 7.1. RPG quality model ADG

The black nodes in the model are defined by the ISO/IEC 25010 international standard, the white nodes are the source code metrics calculated by the SourceMeter for RPG tool, and finally, the gray nodes are the inner nodes defined by us to help revealing the dependencies in the model. We call the source code metric nodes as *Sensor nodes* and the other nodes as *Aggregated nodes*. Table 7.1 contains the descriptions of all nodes.

An essential part of the approach, besides the quality model, is the benchmark, which contains several RPG systems. By comparing the system with the applications from the benchmark, the approach converts the low-level source code metrics into quality indices. For the edges of the *ADG*, a survey was prepared. In this survey, the developers were asked to assign weights to the edges, based on how they felt about the importance of the dependency.

Table 7.1. Description of the nodes in the RPG Quality Model.

Sensor nodes	
CC	Clone coverage. The percentage of copied and pasted source code parts, computed for the subroutine, procedure, program, and the system itself.
CLOC	Comment Lines of Code. Number of comment and documentation code lines of the subroutine/procedure/program. Subroutines' CLOC are not added to the containing procedure's CLOC. Similarly, subroutines' and procedures' CLOC are not added to the containing program's CLOC.
CD	Comment Density. The ratio of comment lines compared to the sum of its comment and logical lines of code (CLOC and LLOC). Calculated for subroutines, procedures, and programs.
LLOC	Logical Lines of Code. Number of code lines of the subroutine/procedure/program, excluding empty and comment lines. In case of a procedure, the contained subroutines' LLOC is not counted. Similarly, in case of a program, the contained subroutines' and procedures' LLOC is not counted.
McCC	McCabe's Cyclomatic Complexity [67] of the subroutine/procedure. The number of decisions within the specified subroutine/procedure plus 1, where each if, else-if, for, do, do-while, do-until, when, on-error counts once. Subroutines' McCC are not added to the containing procedure's McCC.
NLE	Nesting Level Else-If. Complexity of the subroutine/procedure/program expressed as the depth of the maximum embeddedness of its conditional and iteration block scopes, where in the if-else-if construct only the first if instruction is considered. The following RPG language items are taken into consideration: if, for, do, do-while, do-until, select, and monitor. The else-if, else, when, other, and on-error operations do not increase the value of the metric; however, they can contain elements defined above, which increase NLE. Subroutines' NLE are not added to the containing procedure's NLE. Similarly, subroutines' and procedures' NLE are not added to the containing program's NLE.
NII	Number of Incoming Invocations. Number of other subroutines which directly call the subroutine.
NOI	Number of Outgoing Invocations. Number of other subroutines which are directly called by the subroutine.
Warning P1	The number of critical rule violations in the subroutine/procedure/program. They can be potential root causes of system faults.
Warning P2	The number of major rule violations in the subroutine/procedure/program. Serious coding issues which makes the code hard to understand, and can decrease efficiency.
Warning P3	The number of minor rule violations in the subroutine/procedure/program. These are only minor coding style issues, makes the source code harder to comprehend.
Aggregated nodes	
Changeability	The capability of the software product to enable a specified modification to be implemented, where implementation includes coding, designing and documenting changes.
Complexity	Represents the overall complexity of the source code. It is represented by the McCabe's Cyclomatic Complexity and the Nested level of the subroutines.
Comprehensibility	Expresses how easy it is to understand the source code. It involves the complexity, documentation and size of the source code.
Documentation	Expresses how well the source code is documented. It is represented by the density and the amount comment lines of code in a subroutine.
Fault proneness	Represents the possibility of having a faulty code segment. Represented by the number of minor, major and critical rule violations.
Stability	The capability of the software product to avoid unexpected effects from modifications of the software.
Aggregated nodes defined by the ISO/IEC 25010 standard	
Maintainability	Degree of effectiveness and efficiency with which a product or system can be modified to improve it, correct it or adapt it to changes in environment, and in requirements.
Analyzability	Degree of effectiveness and efficiency with which it is possible to assess the impact on a product or system of an intended change to one or more of its parts, or to diagnose a product for deficiencies or causes of failures, or to identify parts to be modified.
Modifiability	Degree to which a product or system can be effectively and efficiently modified without introducing defects or degrading existing product quality.
Testability	Degree of effectiveness and efficiency with which test criteria can be established for a system, product or component and tests can be performed to determine whether those criteria have been met.
Reusability	The degree to which an asset can be used in more than one system, or in building other assets.

7.3.1 Integration with QualityGate

Since our intention was to provide valuable information for the managers and developers on a daily basis, it was necessary to integrate our approach into a continuous quality management tool. The SourceMeter for RPG tool and the ColumbusQM approach was integrated into a tool called QualityGate [11]. As a comprehensive software quality management platform, QualityGate is capable of calculating quality values using the ColumbusQM from source code, using a wide range of software quality metrics provided by the SourceMeter for RPG tool. It is empowered by a built-in quality model conforming to the ISO/IEC 25010 standard and has a benchmark containing analysis data from a large number of RPG systems. This makes it possible to calculate objective quality attributes and estimate upcoming development costs [10].

7.4 Case Study

In this section, we present our empirical results about how we managed to integrate the introduced quality model for RPG approach and the QualityGate tool suite into the life of a mid-sized software company.

We organized the case study into four consecutive phases. In the first, *initial phase*, we calibrated the SourceMeter for RPG tool and created a benchmark and a quality model for RPG. In the next, *integration phase*, we prepared the IBM i mainframe to regularly generate spool files and upload them into a subversion repository. In the *refactoring phase*, a software module was improved based on the guidelines of the approach. Finally, the last part of the case study is the *discussion phase*, where we discussed our experiences about the approach.

7.4.1 Initial Phase

In the initial phase, we personalized our approach to the R&R Software company. First, we went through every software metric, rule violation and clone property and calibrated them in cooperation with RPG experts.

We created a survey to set the priorities of each rule violation. Developers classified each rule violation as minor, major or critical. We also implemented some new rule violations, which seemed useful based on their opinions. We also asked them to set the limits of specific metric based rule violations, for example, what is the interval for Nesting Level (NLE) value which is considered acceptable for a subroutine.

To build a benchmark, we used large, real-life RPG modules provided by the company. The basic statistics about the modules can be found in Table 7.2. The table shows that the modules vary between 18K and 129K in total logical lines of code (TLLOC) with thousands of subroutines. Since we planned to compare the results of the given application to another applications, we used these four modules as benchmark.

Finally, we created and weighted a quality model in cooperation with the RPG experts at R&R company. This quality model was presented earlier in this chapter and can be seen in Figure 7.1.

Table 7.2. Basic statistics of the benchmark systems.

System	TLLOC	Program num.	Subroutine num.
AB	128,146	264	5,355
IV	18,378	62	850
LG	65,655	163	2,336
PR	129,484	288	5,944

7.4.2 Integration Phase

In the second phase, we integrated our approach into their IBM i programming environment. The flowchart of the whole approach is shown in Figure 7.2.

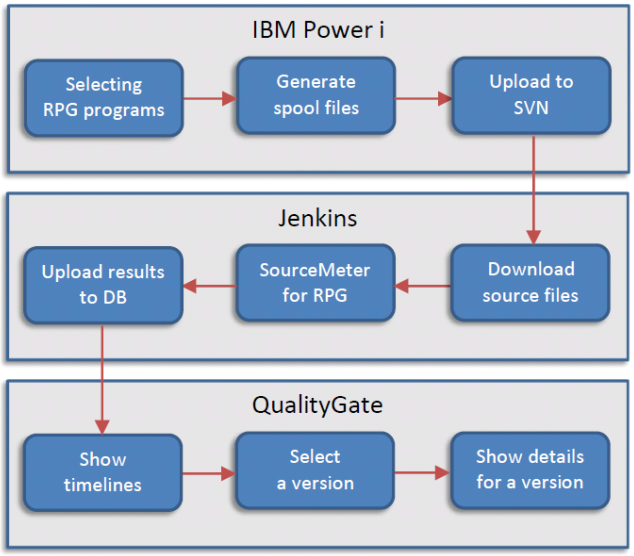

Figure 7.2. Process chain of the approach.

First of all, we had to program the IBM i mainframe to generate the spool files from the chosen system on a daily basis. Spool files are the result of the RPG compilation and, besides the source files, they contain several useful information about the compilation. SourceMeter for RPG supports both spool files and raw source files, but since spool files contain more information it is recommend to use them. After the spool file generation is finished, the IBM i starts a script that uploads the spool files to an SVN version control system.

We used an extendable open source continuous integration server called Jenkins[2] to analyze every version in the subversion repository. Jenkins makes it possible to extend the build system with various plugins. We implemented a Jenkins plugin, which provides a user interface where the user can set the specific properties of the evaluation, like the used quality model, subversion repository, or frequency of the analysis. By using the plugin, when a new version is committed to the subversion repository, Jenkins automatically downloads the source files and executes the SourceMeter for RPG tool and right after it uploads the results to the database of QualityGate.

[2]https://jenkins.io/

Finally, QualityGate shows the quality of the evaluated versions on different timeline and spider charts. On the timelines, the developers can choose a specific version and find the root cause for why the given quality characteristic has been changed.

7.4.3 Refactoring Phase

As a result of the previous phases, the continuous quality monitoring tool was ready to be used. R&R selected a part of the LG module for refactoring, which they wanted to improve. The basic statistics before and after the refactoring of this submodule can be seen in Table 7.3. As it was hoped and expected, during the refactoring the overall maintainability of the module increased. The number of critical rule violations (WarningP1) halved and only about one third of the major rule violations (WarningP2) remained in the code. The clone coverage of the module did not change, since no effort was put into reducing clones. Also, the developers did not reduce the number of minor rule violations (WarningP3).

Table 7.3. Basic statistics of the LG submodule before and after the refactoring.

Property	Before refactoring	After refactoring
TLLOC	5,266	5,319
Program num.	9	9
Subroutine num.	226	233
Maintainability	4.87	5.41
WarningP1 num.	109	53
WarningP2 num.	238	80
WarningP3 num.	262	262
Clone Coverage	0.17	0.17

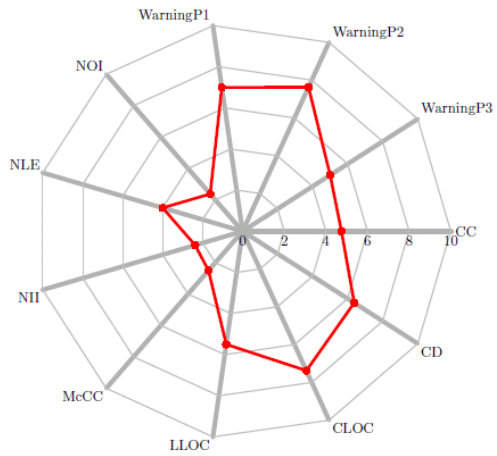

Figure 7.3. Low-level quality results.

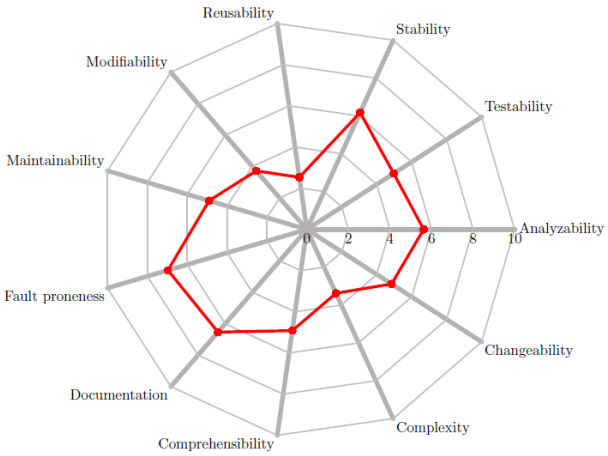

Figure 7.4. High-level quality results.

The initial low and high-level quality results of the submodule can be seen in Figure 7.3 and Figure 7.4. The maintainability of this system before the refactoring was 4.87 on the scale of [0,10] (larger is better). Since the average is 5, this means the initial

maintainability of the selected LG submodule is slightly under the average (based on the benchmark). Based on Figure 7.3, we can assume that this is mostly because of the NII, NOI, McCC and NLE metrics, because their goodness value is the lowest in the model. According to the RPG quality model, the metrics influence the goodness of the Complexity and Reusability aggregated nodes, e.g. Complexity is calculated from McCC and NLE, and it also has a very low 3.38 goodness value.

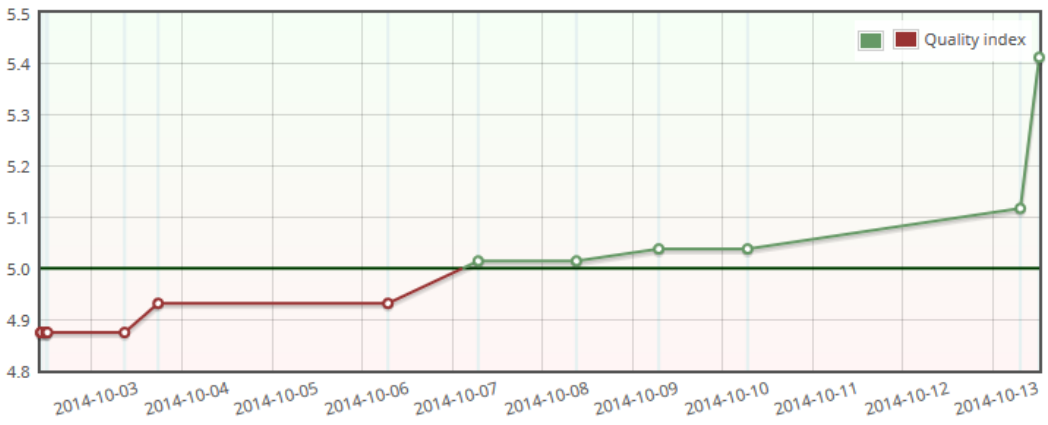

Figure 7.5. Maintainability quality timeline.

The business process of a refactoring task is quite simple with QualityGate. The lead developer needs to find a coding issue based on the guidelines of the approach. This issue can be, for example, a badly maintainable subroutine, a rule violation or a code duplication. The next step is to assign the issue as a ticket to a suitable developer with some comment. Finally, the developer has to fix the specified issue. At the time the developer committed the fix for the issue, the new version will be uploaded to QualityGate. If the issue was a rule violation or a code duplication, the validation of the fix is done automatically, since it disappeared from the new version.

The progress of the maintainability quality characteristic of the system during the whole refactoring process can be seen in Figure 7.5. As it was expected, the maintainability of the submodule increased with almost every new version. After a week of refactoring, the quality of the submodule went above 5. It means it became more maintainable than the average according to the benchmark. As we can see in Figure 7.5, at this time the color of the timeline changed from red to green, meaning the quality of the system reached a specific threshold value. This previously set threshold value determines whether the system is in NO-GO or GO state. This way, managers can easily set the target quality of a specific application.

In Figure 7.6, we can see that no effort was put into improving the McCC complexity of the module in the first week of the refactoring. Although, at the end of the refactoring process it increased by 0.5, it remained still under 3. The complexity of the system is still critical, but during the refactoring its goodness increased from 3.38 to 3.69.

In Table 7.3, we saw that the number of critical rule violations (WarningP1) decreased a lot during the refactoring. This improvement can be seen on the quality timeline of the WarningP1 node in Figure 7.7. If we compare it with the maintainability timeline, we can see that fixing critical rule violations had an important role in the two highest improvements on the maintainability timeline. These improvements were mainly caused by the elimination of rule violations.

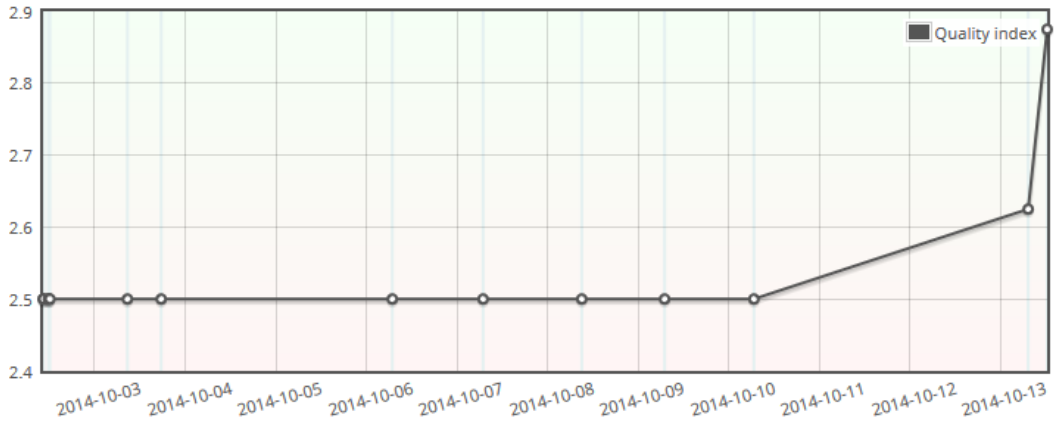

Figure 7.6. McCC quality timeline.

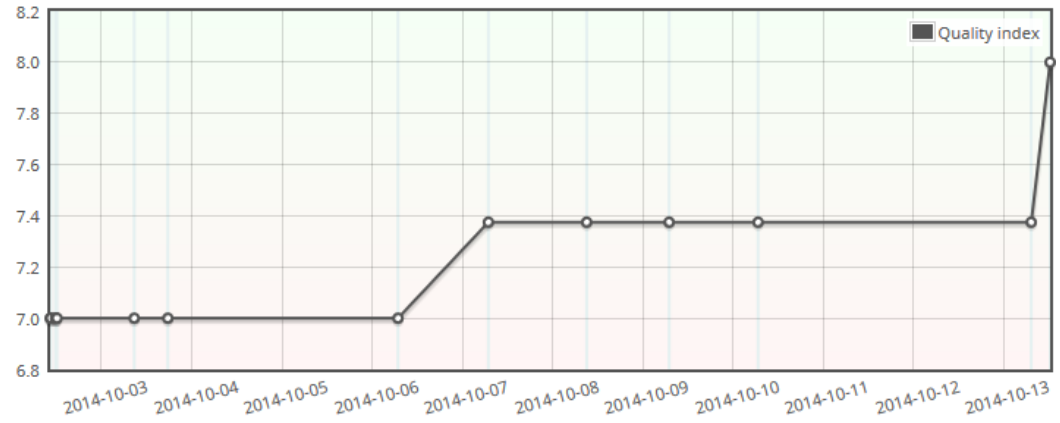

Figure 7.7. WarningP1 quality timeline.

In the followings, two simple cases will be shown to demonstrate the refactoring mechanism applied on the elimination of rule violations. Companies have different standards for the allowed set of operations to be used in the code. Since the given set can differ from company to company, the list of operations to avoid in code is customizable. At R&R Software, *ADDDUR* is one of the operations to avoid. Using such an operation is forbidden according to the company's standard.

Look at the code snippet from the LG module, shown in Listing 7.1. A subroutine definition can be seen named *fk0601*. In the first step, a checking subroutine is called, then, if no error was found in the date format, an assignment is done, and finally another check is performed by calling *ckfm01*. The ADDDUR operation adds the duration specified in the second operand called factor 2 (*"7:*days"* which means that the number of days is increased by seven) to a date or time and places the resulting Date, Time or Timestamp in the result field (*@@date*). Since the ADDDUR operation is on the avoid operation list, a critical rule violation will occur pointing to the relevant code location.

To eliminate ADDDUR from the code, refactoring should be performed on the code. In Listing 7.2 the same *fk0601* subroutine is shown with some modifications. After checking whether the date is in the desired format, the two *eval* and the *adddur* operations were combined into one single *eval* operation that has the same functionality.

Listing 7.1. Using avoid operation in a subroutine

```
....+....1....+....2....+....3....+....4....+....5....+..
     C     fk0601         begsr
     C                    exsr       ckdtfr
      * No error --> Process
     C     *in99          ifeq       *off
     C                    eval       @@date = %date(wsdtfr)
     C                    adddur     7:*days       @@date
     C                    eval       wsdtfr = %dec(@@date)
     C                    exsr       ckfm01
     C                    endif
     C                    endsr
```

Listing 7.2. Eliminate avoid operation rule violation

```
....+....1....+....2....+....3....+....4....+....5....+..
     C     fk0601         begsr
     C                    exsr       ckdtfr
      * No error --> Process
     C     *in99          ifeq       *off
     C                    eval       wsdtfr = %dec(%date(↩
       wsdtfr)
                                                   +%days↩
                                                     (7))
     C                    exsr       ckfm01
     C                    endif
```

Another critical rule violation is when a programmer does not specify an error handling subroutine on a File Specification line, thus error(s) that occur during file management will not be handled correctly. In Listing 7.3, a file named *lro1000c* is specified as an external file and also a fully procedural file is stored on a local disk for only input purposes (in this program). A line continuation is added with a keyword *rename* to reference the file more easily from code. In this case, to handle input errors on file *lro1000c*, we need to add another keyword that is *infsr*. The given parameter of the *infsr* keyword is the name of the subroutine that will handle occurring input errors. The corrected source snippet can be seen in Listing 7.4. A keyword has been added that marks *srinfs* as the error handling subroutine.

Listing 7.3. Missing error handling on File Specification

```
....1....+....2....+....3....+....4....+....5....+...
     Flro1000c  if   e           k disk
     F                                     rename(rorc:↩
       ro1000cr)
```

Listing 7.4. Eliminate missing error handling on File Specification

```
....1....+....2....+....3....+....4....+....5....+...
     Flro1000c  if   e           k disk
     F                                     infsr(srinfs)
     F                                     rename(rorc:↩
        ro1000cr)
```

By eliminating these critical rule violations, it is very likely that the quality of the system will improve. Nevertheless, as we have seen, the quality index is calculated from a wide set of metrics. Consider a case when a programmer is told to eliminate a code clone. So, the developer extracts the cloned code parts into a procedure and calls it from the right places where formerly the clones were located (assume that clone instances were located in two subroutines). In this case, the value of the CC (Clone Coverage) metric will be lower, but the NOI (Number of Outgoing Invocations) values will increase for both subroutines. To sum up, improving one characteristic of the system does not necessarily result in the improvement of the quality index of the system.

7.4.4 Discussion Phase

The last phase was the discussion phase, where we concluded our experiences about the approach in a workshop. Altogether they found our approach useful, easy to adapt and use. We organized their opinions into the following four points.

- According to them, one of the most important features of the approach is that they were able to check the RPG coding conventions inside the company using the precisely calibrated rule violations and metrics. This feature helped them to force their developers to avoid undesirable solutions. This makes the source code more maintainable, since when a few months later a different developer would like to maintain the code part it will be much easier.

- The customization of the approach can be done easily. QualityGate provides user interfaces to create new benchmarks and quality models. In the future, they would like to use this feature to validate and upgrade their source code generator with a quality model which is not designed for maintainability, but for their special task.

- The best time to use the method is before testing, since the tool could help prevent unnecessary test cycles by revealing possible faults. Another good occasion to use the method is when a new project is started or an application needs to be upgraded.

- A long term benefit of using this approach is that developers will learn what the common patterns for solving different problems are and what constructs should be avoided. This way, not only the company will benefit using the approach, but the developers as well.

7.5 Threats to Validity, Limitations

Overall, the results are promising, but we have to highlight the limitations and the threats to validity of our approach as well.

One major limitation of our approach is generalizability, since the entire process was designed within a cooperation with one software company. The quality model reflects their and our opinion, the benchmark was created from their software systems and they designed several rule violations. The approach can be used in other companies as well, but it is highly recommended to create a new benchmark and calibrate the weights of the used quality model.

The validation of our method was based only on the opinions of the developers about the improvement of one submodule. Examining more refactoring processes, collecting exact values, and running statistical tests would have provided more precise results, but only a few developers were working on the refactoring project and we could not run such tests because of the lack of data.

A big question about software metrics and rule violations affects the reliability of our approach as well. We know that they are useful, they have proven to be good indicators for some aspects of software quality, but maintainability is still a subjective, high-level characteristic, and we cannot be sure that we took every important aspect into consideration.

7.6 Summary

In this chapter, we introduced a software quality model for the RPG programming language, which conforms to the ISO/IEC 25010 international standard, and integrated it into the continuous software quality monitoring tool called QualityGate. We used software metrics, rule violations, and code duplications to estimate the maintainability of RPG software systems and to reveal various source code issues.

As a validation of our approach, we presented a case study in cooperation with the R&R Software development company. We organized the case study into four phases. In the initial phase we calibrated the parameters of the tools, and in the second phase we set up the IBM i compiler to generate spool files into a subversion repository on a daily basis. The third phase of the case study was to refactor one of their submodules chosen by the company. Finally, as a conclusion, we discussed their experiences in using our approach.

Based on the opinions of the developers, the industrial application of our method was a success. During the refactoring phase, the number of critical and major rule violations were halved. Despite the fact that the complexity goodness of the examined system is still under the average, its maintainability value increased and crossed the baseline value, so the state of the project had changed from NO-GO to GO.

"Avoid complexities. Make everything as simple as possible. "

— Henry Maudslay

8

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

8.1 Overview

ISO 25010 standard describes maintainability as one of the eight quality characteristics. Maintainability has become the most important trait related to quality, as the cost of maintaining a software system adds up to 40-60% of the total costs of a software[24, 83]. This is why researchers focus on maintainability and try to discover relationships between maintainability and different characteristics of the system.

There are numerous business applications having their core written in RPG. Maintainability of legacy business applications are likely to be more important. More effort should be put into research studies that deal with legacy systems' maintainability to prevent software erosion.

Halstead metrics are the first complexity measures that were defined by Maurice Halstead[46]. He thought that many characteristics of a software system can be expressed by only using the number of operands and operators occurred in a software. Halstead complexity metrics were first calculated for IBM RPG systems in 1982 presented by Hartman[48]. At that time, the calculation was performed on RPG II and RPG III systems that are very rare nowadays.

In this chapter, we propose a definition of Halstead complexity metrics for newer versions of RPG, namely IBM RPG/400 and RPG IV. RPG IV brought new core language features that makes the calculation of Halstead complexity metrics absolutely different than before. Free-form block (column independent) constructions have the most impact on the methodology. We extended our static analysis tool, SourceMeter, to calculate Halstead complexity measures for RPG. We used this tool to calculate the metrics for 348 RPG programs containing 7475 subroutines. We also applied four Maintainability Index (MI) metrics that are widely used to express the overall maintainability of a software. For instance, Microsoft's Visual Studio is currently using a Maintainability Index definition to provide an overall maintainability/quality measurement for a system. Maintainability Index depends upon Halstead's Volume, which

motivates us to investigate the Halstead's Complexity metrics to get a deeper insight into how they are related to other software metrics. Similarly, maintainability models are constructed to gain an overall maintainability score by aggregating low-level metric values. In chapter 7, we defined a quality model for RPG[115].

To determine which metrics form groups that have strong inner connections, researchers often use the concept of Principal Component Analysis (PCA). PCA is also used to reveal hidden connections in the dataset and to reduce the dimensionality of the data. Based on the Principal Component Analysis, we determined how our previous model could be extended to involve more metrics, thus ensuring a stronger descriptive behavior of our maintainability model.

The rest of this chapter is structured as follows. In Section 8.2, we present the most important studies that are related to our research domain. In Section 8.3, we present the definitions of the used metrics. Section 8.4 shows the results of the Principal Component Analysis and we make suggestions for extending the quality model. Finally, we end the chapter with enumerating the threats to validity and summarizing the results.

8.2 Related Work

In this section, we present the most important studies that relate to static source code analysis in RPG systems and software metrics defined for RPG language. In the literature there is a lack of studies that focus on RPG legacy software systems, hence we can only enumerate a very limited number of papers that have RPG related research topics. We can hardly identify groups of research areas when RPG is in the spotlight. The first studies that are related to IBM RPG are from 1982. Naib investigates internal (not varying with time - McCabe, Halstead, Lines of Code) and external (varying with time - number of users) metrics on two large RPG packages to see whether the metrics have correlation with error rates.[79]. Different internal measures are calculated at module level for which Naib used Hartman's counting tool to support the identification of fault-prone RPG II and RPG III modules[48]. Hartman used the original definitions to calculate McCabe's Cyclomatic Complexity[67] and the Halstead's complexity metrics[46].

The usefulness of metrics are mostly accepted, however, sometimes the metrics are criticized rather to pinpoint the weaknesses and add a gentle indication to change or modify the directions of the research areas [91]. These kind of studies often reflect the misuse of metrics in different models. Halstead's complexity metric family as being one of the first complexity metric set is sometimes handled as a golden hammer [25] that is obviously a bad practice. Consequently, more metrics were defined and used for empirical analysis to show different characteristics of the subject systems [72, 17]. For evaluating new complexity metrics, sometimes different frameworks are used [71]. Maintainability Index (MI) was first introduced by Oman et al. in 1994 [80, 30]. MI was designed to express the maintainability of a system (as its name reflects) with a single value. Its power has become its weakness, since it does not provide any information on how the metric value was made up (maybe only one lower level metric is critical) or what changes should be made to improve the system's maintainability [51]. As ISO 25010 describes, maintainability is a derived quality indicator which comprises modularity, reusability, analyzability, modifiability, and testability. However, Maintainability Index is an ideal measurement when one would like to compare the overall maintainability

of different software systems. Maintainability models were proposed to overcome the above mentioned problems [74, 61] and soon more complex quality models were being created [81, 89, 9].

Bakota et al. presented a probabilistic software quality model where the overall maintainability is derived from analyzability, changeability, stability, testability [9]. They used the ISO 9126 standard, which is the ancestor of ISO 25010, thus this model has become quite outdated and needs to be updated, however, it is still usable. In case of RPG, we have proposed a similar quality model in Chapter 7 which is based on the results of the probabilistic software quality model. In this chapter, we would like to give recommendations for extending the RPG quality model to involve more measurements that reflect the overall quality of a system in a more precise way.

8.3 Computing Halstead Metrics and Maintainability Index for RPG

Halstead Complexity metrics[46] are likely to be forgotten, which is undeserving in many cases. For instance, Maintainablity Index[30] shows the strength of Halstead's metrics. Coleman et al. used Halstead's Effort metric amongst others to derive the original Maintainability Index (MI) metric. At that time, Halstead's volume and effort metrics were considered to be the best indicators for predicting the maintainability of a software system.

To produce the necessary metric values, we first present the list of definitions for Halstead metrics. Let us consider the following notations:

- η_1 = number of distinct operators
- η_2 = number of distinct operands
- N_1 = total number of operators
- N_2 = total number of operands

Now we have the definitions of the four basic metrics we will use in our further formulas, however, there is no intention or concept that describes what should be considered as an operand and an operation. This problem can cause inconsistencies between research papers since they use different interpretations. Furthermore, the calculation of operands and operators can significantly differ by programming languages (mainly coming from the dissimilarities of the languages). Fortunately, in case of RPG we do not have to dig deep to figure out how different source code elements should be treated. We calculated the Halstead's complexity metrics similarly to how it was presented by Hartman for RPG III, thus we concentrate on the peculiarities of RPG IV. Now we will present the different source code elements that should be included in the calculations.

Table 8.1 summarizes the source code elements in different RPG versions to be counted as operators. Calculation specification is the place where we can specify the operations to be done on the given operands. In RPG IV, we use free-form to avoid column-sensitive programming. In the free-form section, we can use different operators such as infix operators (+,-,*,/, < ,>, ...), member selection (data structure field select), array subscription (to get elements from an array), parentheses (to modify the

Table 8.1. List of source code elements to be counted as operators

Specification name	Construct name	RPG version
Calculation	Operator	RPG/400 RPG IV
Free-form (C)	Infix expressions	RPG IV
Free-form (C)	Member Selection	RPG IV
Free-form (C)	Array Subscript	RPG IV
Free-form (C)	Parentheses	RPG IV
Free-form (C)	Prefix Expressions	RPG IV

operation precedence), and also prefix operations. Most of the free-form statements can be written in calculation specifications, some cannot.

Table 8.2. List of source code elements to be counted as operands

Specification name	Construct name	RPG version
Calculation	Factor 1	RPG/400 RPG IV
Calculation	Factor 2	RPG/400 RPG IV
Calculation	Result Field	RPG/400 RPG IV
Definition	(Variable) Name	RPG IV
Input	Program Field	RPG/400, RPG IV
Input	Data Structure	RPG/400, RPG IV
Input	Data Structure Subfield	RPG/400, RPG IV
Input	External Record	RPG/400, RPG IV
Input	External Field	RPG/400, RPG IV
Input	Data Structure	RPG/400, RPG IV
Input	Data Structure	RPG/400, RPG IV
Input	Named Constant	RPG/400
Free-form	Literal	RPG IV
Free-form	Identifier	RPG IV
Output	Output External Record	RPG/400, RPG IV
Output	Output External Field	RPG/400, RPG IV
Output	Output Program Field	RPG/400, RPG IV

Table 8.2 shows the RPG constructions to be counted as operands. When we use an operator in Calculation Specification we have to specify operand(s) (if needed) to perform the operation on. These operands should be specified in columns named factor 1 and factor 2. The result of the operation is stored in the given result field. In RPG IV we can use Definition Specification to define variables and constants. We use Input and Output Specification to declare the appropriate input and output data structures and their fields (also constants in RPG/400). In RPG IV we can also use literals and identifiers in free-form section which are also counted as operands.

Table 8.3 introduces the Halstead metrics that are aggregated from the basic ones (η_1, η_2, N_1, N_2). Table 8.4 presents the different variants of Maintainability Index (MI) metrics.

In RPG, we have 3 levels of abstraction, namely subroutine, procedure, and program. We can define a subroutine by writing code between the BEGSR and ENDSR

Table 8.3. List of the used Halstead metrics

Metric Name	**Formula**
Program Vocabulary (HPV)	$\eta = \eta_1 + \eta_2$
Program Length (HPL)	$N = N_1 + N_2$
Calculated Program Length (HCPL)	$\hat{N} = \eta_1 \cdot log_2 \eta_1 + \eta_2 \cdot log_2 \eta_2$
Volume (HVOL)	$V = N \times log_2 \eta$
Difficulty (HDIF)	$D = \frac{\eta_1}{2} \times \frac{N_2}{\eta_2}$
Effort (HEFF)	$E = D \times V$
Time required to program (HTRP)	$T = \frac{E}{18}$
Number of delivered bugs (HNDB)	$B = \frac{E^{\frac{2}{3}}}{3000}$

Table 8.4. List of the used Maintainability Index metrics

MI variant	**Formula**
Original (MI)	$171 - 5.2 \times ln(HVOL) - 0.23 \times McCC - 16.2 \times ln(LLOC)$
SEI (MISEI)	$171 - 5.2 \times log_2(HVOL) - 0.23 \times McCC - 16.2 \times log_2(LLOC) + 50 \times sin(\sqrt{2.4 * CD})$
Visual Studio (MIMS)	$max(0, 100 \times \frac{171 - 5.2 \times ln(HVOL) - 0.23 \times McCC - 16.2 \times ln(LLOC)}{171})$
SourceMeter (MISM)	$max(0, 100 \times (\frac{171 - 5.2 \times log_2(HVOL) - 0.23 \times McCC}{171} - \frac{16.2 \times log_2(LLOC) + 50 \times sin(\sqrt{2.4 * CD})}{171}))$

operation codes. To call a subroutine we have to use the EXSR operation and specify the name of the subroutine to be called. Unlike subroutines, procedures can be prototyped and have parameters, thus supporting a more flexible way to reuse code portions. Programs are larger building blocks that encapsulate subroutines and procedures as well. In this chapter, we will only examine subroutines and programs because we accessed a limited set of source code files that mainly contains subroutines instead of procedures.

8.4 Evaluating the usefulness of Halstead's and MI metrics

Principal Component Analysis (PCA) [55] is widely used in many domains to accomplish dimensionality reduction and uncover patterns from the data [19, 64]. PCA determines which dimensions are the most important ones and which ones represent the most variation in the data. PCA takes a dataset (a set of metrics in our case) as input and outputs principal components (uncorrelated dimensions) that span the direction of the $1^{st}, 2^{nd}, 3^{rd}, \ldots$ largest variations.

We have performed PCA both at program (RPG file) and subroutine level to see the difference between these levels if any exists. We investigated 348 RPG programs (185 RPG IV and 163 RPG/400 programs) and 7475 RPG subroutines.

We first present the correlation matrices that can be seen in Table 8.5 and Table 8.6. We included the Halstead, Maintainability Index metrics, and the sensor metrics that are used by the quality model in the correlation matrix to investigate the relationship between them. Values in the tables are mapped with color codes to help better understand the correlations between metrics. The color interpolation has three base points: -1, 0, 1. The greater the correlation between two metrics (negative or positive correlation) the grayer the cell is (1 and -1 values imply dark gray color). White means that two variables are not correlated. One can see clear groups of metrics that correlation coefficients are very high inside the group. In case of programs, Halstead metrics form such a group, which is not surprising since many of them are calculated with the help of others, furthermore, all metrics can be derived from the four basic ones (See Table 8.3). Maintainability Index metrics have the same characteristics. Their lowest correlation is 0.946 (program level) and 0.997 (subroutine level) that are both very high values. High correlation is caused by the fact that each variant has almost the same core in their formula. A relatively high correlation can be seen in case of the different warnings (avg. correlation: 0.774) at program level, but the same cannot be said about subroutines.

It is promising that the correlations between Halstead metrics and warnings are high (avg. correlation is 0.812) since we can use the Halstead metrics to predict warnings in the system (at program level). Unfortunately, no valuable correlation was found at subroutine level between these metrics. The Halstead complexity metrics are also highly correlated with the McCC metric (we use the Program Complexity (PC) terminology at program level), which means that each complexity measure can express the other. This is partly true at subroutine level since HCPL, HPL, HPV and HVOL have poor correlations with McCC. At program level, the McCabe's Complexity metric can also be used to express the warnings in the system, since it has 0.836 avg. correlation coefficient with the warning metrics.

PCA constructs 25 dimensions (factors) from 26 dimensions that is not the best case scenario. However, using the first ten factors will give back 96.865 (program level) and 96.366 (subroutine level) percent of the total variability. Figure 8.1 and Figure 8.2 depict the eigenvalues for all the 25 factors and the cumulative variability at program level and subroutine level respectively. The cumulative variability is slightly steeper in case of programs; meaning that we can reconstruct the original data by using less dimensions (factors).

Table 8.5. Correlation between metrics (Program Level)

Variables	CC	HCPL	HDIF	HEFF	HNDB	HPL	HPV	HTRP	HVOL	MI	MIMS	MISEI	MISM	NLE	PC	NOI	CD	CLOC	TCD	LLOC	WarningInfo	Clone Metric Rules	Complexity Metric Rules	Coupling Metric Rules	Doc. Metric Rules	Size Metric Rules
CC	1	0,073	0,162	0,033	0,067	0,066	0,087	0,033	0,055	-0,186	-0,186	-0,156	-0,149	0,099	0,061	0,074	0,083	0,207	0,009	0,145	0,435	0,485	0,058	0,020	0,007	0,079
HCPL	0,073	1	0,811	0,891	0,947	0,970	0,996	0,891	0,965	-0,698	-0,698	-0,662	-0,667	0,521	0,951	0,215	0,076	0,461	-0,417	0,513	0,785	0,712	0,889	0,834	0,789	0,933
HDIF	0,162	0,811	1	0,736	0,865	0,841	0,835	0,736	0,812	-0,673	-0,673	-0,610	-0,627	0,645	0,836	0,346	0,162	0,463	-0,400	0,253	0,771	0,720	0,774	0,669	0,619	0,829
HEFF	0,033	0,891	0,736	1	0,967	0,953	0,858	1,000	0,966	-0,464	-0,464	-0,438	-0,447	0,370	0,949	0,082	0,057	0,502	-0,255	0,334	0,745	0,675	0,871	0,825	0,815	0,847
HNDB	0,067	0,947	0,865	0,967	1	0,992	0,933	0,967	0,990	-0,578	-0,578	-0,539	-0,551	0,481	0,984	0,172	0,091	0,510	-0,330	0,346	0,813	0,744	0,910	0,850	0,813	0,911
HPL	0,066	0,970	0,841	0,953	0,992	1	0,956	0,953	0,998	-0,606	-0,606	-0,567	-0,578	0,483	0,987	0,189	0,090	0,515	-0,350	0,379	0,818	0,746	0,918	0,866	0,821	0,928
HPV	0,087	0,996	0,835	0,858	0,933	0,956	1	0,858	0,946	-0,740	-0,740	-0,700	-0,706	0,554	0,935	0,256	0,088	0,451	-0,442	0,521	0,786	0,715	0,871	0,813	0,764	0,937
HTRP	0,033	0,891	0,736	1,000	0,967	0,953	0,858	1	0,966	-0,464	-0,464	-0,438	-0,447	0,370	0,949	0,082	0,057	0,502	-0,255	0,334	0,745	0,675	0,871	0,825	0,815	0,847
HVOL	0,055	0,965	0,812	0,966	0,990	0,998	0,946	0,966	1	-0,579	-0,579	-0,543	-0,554	0,459	0,986	0,162	0,081	0,514	-0,332	0,380	0,806	0,733	0,919	0,869	0,827	0,918
MI	-0,186	-0,698	-0,673	-0,464	-0,578	-0,606	-0,740	-0,464	-0,579	1	1,000	0,946	0,950	-0,631	-0,572	-0,482	-0,121	-0,360	0,654	-0,680	-0,540	-0,492	-0,546	-0,432	-0,486	-0,716
MIMS	-0,186	-0,698	-0,673	-0,464	-0,578	-0,606	-0,740	-0,464	-0,579	1,000	1	0,946	0,950	-0,631	-0,572	-0,482	-0,121	-0,360	0,654	-0,680	-0,540	-0,492	-0,546	-0,432	-0,486	-0,716
MISEI	-0,156	-0,662	-0,610	-0,438	-0,539	-0,567	-0,700	-0,438	-0,543	0,946	0,946	1	0,995	-0,578	-0,535	-0,421	0,209	-0,254	0,743	-0,701	-0,496	-0,449	-0,511	-0,406	-0,512	-0,670
MISM	-0,149	-0,667	-0,627	-0,447	-0,551	-0,578	-0,706	-0,447	-0,554	0,950	0,950	0,995	1	-0,593	-0,546	-0,438	0,180	-0,269	0,743	-0,665	-0,507	-0,459	-0,520	-0,415	-0,510	-0,682
NLE	0,099	0,521	0,645	0,370	0,481	0,483	0,554	0,370	0,459	-0,631	-0,631	-0,578	-0,593	1	0,472	0,380	0,133	0,276	-0,461	0,318	0,443	0,398	0,570	0,366	0,337	0,554
PC	0,061	0,951	0,836	0,949	0,984	0,987	0,935	0,949	0,986	-0,572	-0,572	-0,535	-0,546	0,472	1	0,164	0,088	0,479	-0,319	0,352	0,796	0,724	0,915	0,865	0,807	0,911
NOI	0,074	0,215	0,346	0,082	0,172	0,189	0,256	0,082	0,162	-0,482	-0,482	-0,421	-0,438	0,380	0,164	1	0,165	0,282	-0,265	0,135	0,163	0,142	0,165	0,068	0,112	0,314
CD	0,083	0,076	0,162	0,057	0,091	0,090	0,088	0,057	0,081	-0,121	-0,121	0,209	0,180	0,133	0,088	0,165	1	0,305	0,301	-0,094	0,110	0,111	0,084	0,059	-0,102	0,109
CLOC	0,207	0,461	0,463	0,502	0,510	0,515	0,451	0,502	0,514	-0,360	-0,360	-0,254	-0,269	0,276	0,479	0,282	0,305	1	-0,052	0,171	0,584	0,574	0,432	0,428	0,384	0,496
TCD	0,009	-0,417	-0,400	-0,255	-0,330	-0,350	-0,442	-0,255	-0,332	0,654	0,654	0,743	0,743	-0,461	-0,319	-0,265	0,301	-0,052	1	-0,511	-0,228	-0,178	-0,345	-0,222	-0,455	-0,460
LLOC	0,145	0,513	0,253	0,334	0,346	0,379	0,521	0,334	0,380	-0,680	-0,680	-0,701	-0,665	0,318	0,352	0,135	-0,094	0,171	-0,511	1	0,272	0,225	0,385	0,251	0,467	0,449
WarningInfo	0,435	0,785	0,771	0,745	0,813	0,818	0,786	0,745	0,806	-0,540	-0,540	-0,496	-0,507	0,443	0,796	0,163	0,110	0,584	-0,228	0,272	1	0,992	0,737	0,715	0,599	0,748
Clone Metric Rules	0,485	0,712	0,720	0,675	0,744	0,746	0,715	0,675	0,733	-0,492	-0,492	-0,449	-0,459	0,398	0,724	0,142	0,111	0,574	-0,178	0,225	0,992	1	0,654	0,657	0,516	0,668
Complexity Metric Rules	0,058	0,889	0,774	0,871	0,910	0,918	0,871	0,871	0,919	-0,546	-0,546	-0,511	-0,520	0,570	0,915	0,165	0,084	0,432	-0,345	0,385	0,737	0,654	1	0,788	0,801	0,857
Coupling Metric Rules	0,020	0,834	0,669	0,825	0,850	0,866	0,813	0,825	0,869	-0,432	-0,432	-0,406	-0,415	0,366	0,865	0,068	0,059	0,428	-0,222	0,251	0,715	0,657	0,788	1	0,657	0,755
Doc. Metric Rules	0,007	0,789	0,619	0,815	0,813	0,821	0,764	0,815	0,827	-0,486	-0,486	-0,512	-0,510	0,337	0,807	0,112	-0,102	0,384	-0,455	0,467	0,599	0,516	0,801	0,657	1	0,780
Size Metric Rules	0,079	0,933	0,829	0,847	0,911	0,928	0,937	0,847	0,918	-0,716	-0,716	-0,670	-0,682	0,554	0,911	0,314	0,109	0,496	-0,460	0,449	0,748	0,668	0,857	0,755	0,780	1

Table 8.6. Correlation between metrics (Subroutine Level)

Variables	CC	HCPL	HDIF	HEFF	HNDB	HPL	HPV	HTRP	HVOL	MI	MIMS	MISEI	MISM	McCC	NLE	NII	NOI	CD	CLOC	LLOC	Warning Info	Clone Metric Rules	Complexity Metric Rules	Coupling Metric Rules	Doc. Metric Rules	Size Metric Rules
CC	1	-0,089	-0,047	-0,016	-0,045	-0,064	-0,102	-0,016	-0,058	0,100	0,100	0,105	0,105	-0,026	-0,135	0,015	0,027	0,142	-0,026	-0,063	0,825	0,902	-0,023	0,026	-0,021	-0,050
HCPL	-0,089	1	0,358	0,607	0,728	0,954	0,982	0,607	0,982	-0,684	-0,684	-0,680	-0,666	0,306	0,188	-0,092	0,132	-0,509	0,859	0,952	0,265	0,101	0,268	0,076	0,301	0,671
HDIF	-0,047	0,358	1	0,636	0,798	0,438	0,455	0,636	0,353	-0,689	-0,688	-0,691	-0,690	0,743	0,650	0,031	0,167	-0,633	0,316	0,451	0,280	0,123	0,622	0,050	0,384	0,358
HEFF	-0,016	0,607	0,636	1	0,922	0,740	1,000	0,710	0,710	-0,492	-0,487	-0,481	-0,462	0,701	0,228	-0,018	0,104	-0,332	0,600	0,734	0,281	0,116	0,388	0,080	0,347	0,580
HNDB	-0,045	0,728	0,798	0,922	1	0,827	0,763	0,922	0,768	-0,722	-0,720	-0,716	-0,704	0,750	0,387	-0,031	0,158	-0,553	0,680	0,828	0,352	0,145	0,537	0,087	0,444	0,690
HPL	-0,064	0,954	0,438	0,740	0,827	1	0,925	0,740	0,989	-0,669	-0,668	-0,662	-0,645	0,408	0,162	-0,099	0,098	-0,475	0,888	0,995	0,298	0,121	0,301	0,067	0,331	0,722
HPV	-0,102	0,982	0,455	0,601	0,763	0,925	1	0,601	0,901	-0,798	-0,798	-0,795	-0,784	0,387	0,289	-0,088	0,177	-0,630	0,831	0,928	0,290	0,115	0,336	0,094	0,331	0,679
HTRP	-0,016	0,607	0,636	1,000	0,922	0,740	0,601	1	0,710	-0,492	-0,487	-0,481	-0,462	0,701	0,228	-0,018	0,104	-0,332	0,600	0,734	0,281	0,116	0,388	0,080	0,347	0,580
HVOL	-0,058	0,952	0,353	0,710	0,768	0,989	0,901	0,710	1	-0,580	-0,579	-0,573	-0,554	0,330	0,111	-0,094	0,074	-0,393	0,883	0,983	0,263	0,105	0,240	0,054	0,288	0,670
MI	0,100	-0,684	-0,689	-0,492	-0,722	-0,669	-0,798	-0,492	-0,580	1	1,000	0,998	0,997	-0,584	-0,543	0,061	-0,292	0,893	-0,591	-0,687	-0,318	-0,151	-0,472	-0,144	-0,347	-0,528
MIMS	0,100	-0,684	-0,688	-0,487	-0,720	-0,668	-0,798	-0,487	-0,579	1,000	1	0,998	0,998	-0,580	-0,543	0,061	-0,292	0,894	-0,591	-0,686	-0,317	-0,151	-0,471	-0,143	-0,347	-0,528
MISEI	0,105	-0,680	-0,691	-0,481	-0,716	-0,662	-0,795	-0,481	-0,573	0,998	0,998	1	1,000	-0,574	-0,549	0,058	-0,271	0,914	-0,566	-0,680	-0,309	-0,143	-0,469	-0,130	-0,368	-0,519
MISM	0,105	-0,666	-0,690	-0,462	-0,704	-0,645	-0,784	-0,462	-0,554	0,997	0,998	1,000	1	-0,568	-0,552	0,058	-0,273	0,916	-0,551	-0,663	-0,308	-0,143	-0,469	-0,129	-0,367	-0,515
McCC	-0,026	0,306	0,743	0,701	0,750	0,408	0,387	0,701	0,330	-0,584	-0,580	-0,574	-0,568	1	0,531	0,066	0,333	-0,474	0,307	0,428	0,296	0,120	0,663	0,232	0,383	0,393
NLE	-0,135	0,188	0,650	0,228	0,387	0,162	0,289	0,228	0,111	-0,543	-0,543	-0,549	-0,552	0,531	1	0,062	0,261	-0,587	0,101	0,198	0,092	-0,007	0,591	0,082	0,185	0,050
NII	0,015	-0,092	0,031	-0,018	-0,031	-0,099	-0,088	-0,018	-0,094	0,061	0,061	0,058	0,058	0,066	0,062	1	0,091	0,019	-0,119	-0,092	-0,014	-0,008	0,068	0,053	0,016	-0,102
NOI	0,027	0,132	0,167	0,104	0,158	0,098	0,177	0,104	0,074	-0,292	-0,292	-0,271	-0,273	0,333	0,261	0,091	1	-0,124	0,248	0,127	0,200	0,137	0,266	0,655	0,035	0,073
CD	0,142	-0,509	-0,633	-0,332	-0,553	-0,475	-0,630	-0,332	-0,303	0,893	0,894	0,914	0,916	-0,474	-0,587	0,019	-0,124	1	-0,268	-0,493	-0,187	-0,062	-0,385	-0,036	-0,371	-0,327
CLOC	-0,026	0,859	0,316	0,600	0,680	0,888	0,831	0,600	0,883	-0,591	-0,591	-0,566	-0,551	0,307	0,101	-0,119	0,248	-0,268	1	0,890	0,303	0,158	0,237	0,169	0,109	0,652
LLOC	-0,063	0,952	0,451	0,734	0,828	0,995	0,928	0,734	0,983	-0,687	-0,686	-0,680	-0,663	0,428	0,198	-0,092	0,127	-0,493	0,890	1	0,305	0,125	0,324	0,083	0,340	0,722
WarningInfo	0,825	0,265	0,280	0,281	0,352	0,298	0,290	0,281	0,263	-0,318	-0,317	-0,309	-0,308	0,296	0,092	-0,014	0,200	-0,187	0,303	0,305	1	0,961	0,288	0,162	0,201	0,311
Clone Metric Rules	0,902	0,101	0,123	0,116	0,145	0,121	0,115	0,116	0,105	-0,151	-0,151	-0,143	-0,143	0,120	-0,007	-0,008	0,137	-0,062	0,158	0,125	0,961	1	0,093	0,093	0,045	0,087
Complexity Metric Rules	-0,023	0,268	0,622	0,388	0,537	0,301	0,336	0,388	0,240	-0,472	-0,471	-0,469	-0,469	0,663	0,591	0,068	0,266	-0,385	0,237	0,324	0,288	0,093	1	0,208	0,325	0,275
Coupling Metric Rules	0,026	0,076	0,050	0,080	0,087	0,067	0,094	0,080	0,054	-0,144	-0,143	-0,130	-0,129	0,232	0,082	0,053	0,655	-0,036	0,169	0,083	0,162	0,093	0,208	1	0,000	0,066
Doc. Metric Rules	-0,021	0,301	0,384	0,347	0,444	0,331	0,331	0,347	0,288	-0,347	-0,347	-0,368	-0,367	0,383	0,185	0,016	0,035	-0,371	0,109	0,340	0,201	0,045	0,325	0,000	1	0,340
Size Metric Rules	-0,050	0,671	0,358	0,580	0,690	0,722	0,679	0,580	0,670	-0,528	-0,528	-0,519	-0,515	0,393	0,050	-0,102	0,073	-0,327	0,652	0,722	0,311	0,087	0,275	0,066	0,340	1

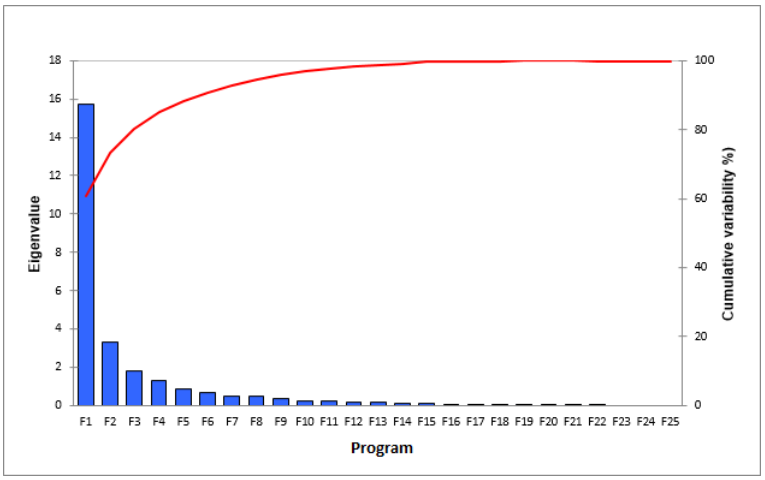

Figure 8.1. Eigenvalues and variability of principal components (Program level)

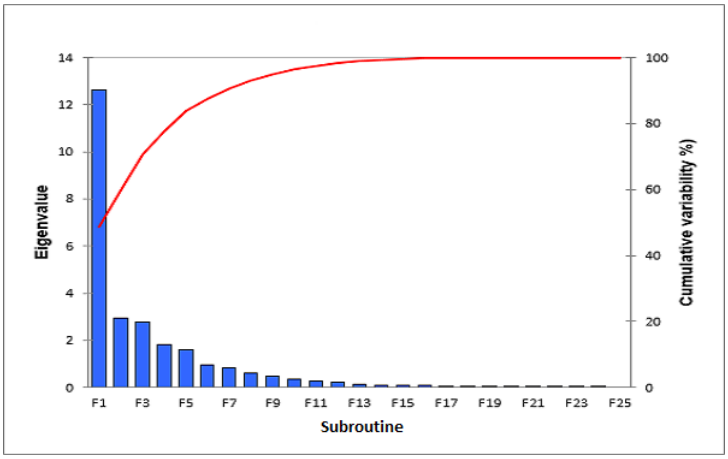

Figure 8.2. Eigenvalues and variability of principal components (Subroutine level)

Factors are constructed from the original metrics with linear combination. It is important to examine the so called factor loadings which give us the linear combinations for each factor. We analyzed the factor loadings for only the first five factors since they retrieve 88.204 and 83.814 percent of the whole variability at program and subroutine level respectively, thus analyzing the most dominant factors is enough to detect the most dominant original metrics. Table 8.7 shows the factor loadings for the first five factors, both at program and subroutine levels. Values higher than 0.7 are highlighted. It is clearly visible that the first factors are made up from many metrics to capture the maximum possible variability. Both in case of program and subroutine levels, the Halstead metrics are the most prominent ones that contribute with the largest weights meaning that they are the most descriptive metrics. Maintainability Index variants are combined with negative weights, but they are also significant ones. Further dominant metrics are different at program and subroutine level. The McCabe Cyclomatic Complexity is as strong as the warning occurrence metrics at program level. At subroutine level, the CD, CLOC and LLOC metrics are stronger besides the Halstead and MI metrics which are absolutely dominating.

Table 8.7. 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

8.4.1 Extend Quality Model For RPG

Consider the maintainability model presented in Figure 7.1.We can enhance the expressiveness of the model by involving further metrics based on the results of the Principal Component Analysis. PCA showed that the Halstead complexity metrics form an independent group that captures the most information of the system (has the largest weights in factor loadings). Considering the correlation matrix we suggest to involve the 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 suggest to include the 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.

8.5 Threats to Validity

It is a very challenging task to find any open source software system written in RPG (since RPG is used in business applications). Consequently, it is hard to gather RPG source code sets from different domains that would guarantee the generalizability. We only have source code from one company we have previously introduced in Chapter 7, and they mostly use subroutines that obviously limits the generalizability.

8.6 Summary

We have applied the Halstead Complexity metrics for RPG/400 and RPG IV programming languages that has never been done before. Furthermore, we have a prototype implementation for these metrics. We also worked out four different Maintainability Index variants in our static source code analyzer. We performed a Principal Component Analysis on 348 RPG programs and on 7475 subroutines and we investigated the relationships between the calculated metrics. We experienced that the Halstead's complexity metrics form a disjoint group that can be used to characterize the warning occurrences in the system at program level. Moreover, Halstead metrics can be involved in a maintainability model to improve its usefulness and compactness. We suggest using the Halstead's Program Vocabulary (HPV) and the Halstead's Number of Delivered Bugs metrics in the model, since these two metrics expand the model the best based on our observations.

"When you reach the end of your rope, tie a knot in it and hang on."

— Franklin D. Roosevelt

9 Conclusions

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.

Future Work

Despite the results we achieved, there are numerous possibilities for extending our work.

In the case of bug datasets, we are planning to make our bug dataset extraction tool open-source, so anybody will be able to use or even improve our method. We plan to do more experiments with our models on other projects. We will try to identify (with statistical methods) connections between the usefulness of the datasets and other descriptors, such as the size of the projects or the amount of reported bugs. We would like to keep the unified bug dataset up-to-date and extend it with public datasets which were published recently (i.e. Had-oops dataset[47], Mutation bug dataset[22], ELFF dataset[93]) and others, which will be published in the future.

There are possible future works in the area of RPG quality assurance, as well. Having a more diverse set of benchmark programs is always desirable. Collecting more source code from more companies would also provide a great opportunity to investigate the behavior of Halstead's metrics at procedure level and a larger benchmark would also be beneficial for testing the quality model itself. Additional low level quality attributes could help in making a more exhaustive comparison and they could also make the quality model better and more general. In the case of the quality model, we would like to create domain specific benchmarks, since it can further improve the quality and the confidence of them. Collecting more votes on the edges (finer weight adjustment) of the quality model can also improve the generality of our method. Since we are able to measure the quality of the original and the migrated versions of a legacy system, it would be interesting to study their quality differences and using these experiences to provide better migration algorithms.

Epilogue

I have always been very keen on promoting quality as the most important aspect since I have started programming. Often, it is not satisfying to have a working prototype as soon as possible. The code will soon become unmanageable and hard to maintain. The old phrase: "Learn from your and from others' mistakes" tends to be true as we can train strong bug prediction models by revisiting mistakes committed in the past. Scanning through discussions under issues made me understand the importance of writing high quality code, and the advantages of quality control. As my code began to be cleaner and more flexible, this approach was applied in my daily life as well. I have become stricter with myself not to choose the easier way but to provide quality results at the end.

Appendices

A

Summary in English

Considering any aspect of life where humans are involved, it is more likely for faults to occur regularly. The thesis focuses on two main topics that emphasize the importance of finding and eliminating software defects in an early stage. In the first part, we present new public bug datasets in order to help building bug prediction models and locate bugs early. The second part presents a more crucial domain (often used in the banking sector) and provides a detailed methodology for measuring the maintainability of RPG software systems. The resulting statements are grouped into two major thesis points. The relation between the underlying supporting publications and the thesis points is shown in Table A.1.

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

The contributions of this thesis point – related to public bug datasets for Java – are discussed in chapters 2, 3, and 4.

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. Furthermore, existing datasets often operate with a narrowed set of software metrics. Based on our findings, we constructed a new dataset for 15 projects selected from GitHub. 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. Contrary to previous bug datasets, we aggregated bugs for the preceding release versions not for the succeeding ones. We performed our dataset creation both at file and class level. Besides the bug dataset being our main contribution, we evaluated its bug prediction capabilities by building 13 different prediction models using this dataset. Tree-based machine

learning algorithms 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, and 0.71 at file level. File level results are lower due to the narrow set of source code metrics. We also investigated the bug coverage to extend our findings. A nearly perfect bug coverage could also be achieved by tagging about 31 percent of the source code elements as faulty both at class and file level. If precision is preferred over recall then using Naive Bayes could be a good option.

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 our own GitHub Bug Dataset, and merged them into one grandiose 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 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). 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. 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 ones, and 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.

This thesis point focuses on the public bug datasets and on their enhancement. We created a state-of-the-art, source code metric based bug dataset, the GitHub Bug Dataset. As a following step, we joined 5 public datasets and included them in one large dataset to provide a strong input for future bug prediction models that use static source code metrics as predictors. Both datasets were made available for the public.

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.

II. Methodology for Measuring Maintainability of RPG Software Systems

The contributions of this thesis point – related to measuring maintainability in RPG systems – are discussed in chapters 5, 6, 7, and 8. 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. 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 following research areas, which use low-level quality characteristics to calculate sophisticated high-level quality indices.

Integrating Continuous Quality Monitoring Into Existing Workflow – A Case Study The goal of this research was to create a general and flexible quality model 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, analysability, modifiability, and testability as the subcharacteristics contributing to maintainability. These characteristics are made up from low-level quality attributes, which are provided by SourceMeter. After defining a quality model, we carried out the 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*. In the second, *integration phase*, 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. 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 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. We found that Halstead's complexity metrics form a strong metric group that can be used to reveal more details about RPG software systems. 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.

This thesis point involves investigating, comparing, evaluating, and extending different static source code analyzers and methods for the RPG programming language. SourceMeter seemed to perform better in case of the calculation of software metrics, code clones, and it also performed a deeper and finer analysis. We built a quality model for RPG that uses the results of SourceMeter. Integrating a continuous quality monitoring into a development cycle of a mid-sized company was successful. Motivated by this success, we wanted to further investigate and improve the quality model itself by recommending Halstead's Complexity metrics to be included in the model. The final proposition, which is based on a Principal Component Analysis, suggests to involve HNDB (Halstead's Number of Delivered Bugs) and HPV (Halstead's Program Vocabulary) metrics into the model.

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: 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.

Table A.1 summarizes the main publications and how they relate to our thesis points.

№	[118]	[114]	[119]	[115]	[117]
I.	♦	♦			
II.			♦	♦	♦

Table A.1. Thesis contributions and supporting publications

B

Magyar nyelvű összefoglaló

Az élet bármely területén, ahol az emberek is szerves részét képezik a folyamatoknak, ott nagyobb valószínűséggel fognak hibák rendszeresen jelentkezni. Ennek tükrében, jelen disszertáció két fő területre fókuszál, melyek hangsúlyozzák a szoftver hibák korai felderítésének és eliminálásának fontosságát. Az első szekcióban új hiba adatbázisok kerülnek bemutatásra, melyek használatával hatékony hiba előrejelző modelleket építhetünk, így segítve a hibák korai felderítését. A második szekcióban egy módszertant ismertetünk, melynek segítségével RPG rendszerek karbantarthatóságát mérhetjük, melyek általában banki rendszerek mélyén ketyegnek, így ezek feltérképezése kritikus fontosságú. A szekciókban megfogalmazott állítások egyben a disszertáció két tézispontját is alkotják. A tézispontokhoz tartozó publikációkat a B.1. táblázat foglalja össze.

I. Új adatbázisok hiba előrejelző modellek építéséhez

A tézispontban a Java környezetű publikus hiba adatbázisokkal foglalkozunk, melynek eredményeit a 2., 3. és 4. fejezetek tárgyalják.

Új publikus hiba adatbázis és kiértékelése a hiba előrejelzésben: Ennek a kutatásnak a középpontjában a létező – statikus forráskód metrikákat használó – publikus hiba adatbázisok gyenge pontjainak feltérképezése állt, illetve ezek alapján egy új hiba adatbázis megalkotása, mely a gyenge pontokat vagy hiányosságot foltozza be. Annak ellenére, hogy a nyílt forráskódú rendszerek közkedvelt hoszt platformja a GitHub, egyetlen egy létező hiba adatbázis sem használja azt forrásként. Az elérhető adatbázisok igencsak korosnak számítanak, így az általuk tartalmazott projektet is idősek. Továbbá ezek az adatbázisok gyakran csak egy szűk forráskód metrikahalmazzal dolgoznak. Az észrevételeink alapján elkészítettünk egy új hiba adatbázist, mely 15 nyílt forráskódú, GitHub-on hosztolt projektet foglal magában. A hibákat leíró adatokat automatizált úton állítottuk elő. Az egyes projektekhez körülbelül 6 hónap hosszú időintervallumokat alkottunk, melyek kezdő- és végpontjai egy-egy hivatalos kiadással esnek egybe, és a hiba információkat ezeken az intervallumokon akkumuláltan gyűjtöttük össze. A létező adatbázisokhoz képest fontos különbség, hogy a hiba adatokat nem a hiba

kijavítását követő verzióhoz propagáltuk, hanem a megelőző kiadási verzióhoz. Az így előállított adatbázis mind osztály és fájl szinten nyújt információkat. Az elkészült hiba adatbázis mellett – mely a fő kontribúciónk is egyben – kiértékeltük az adatbázis hiba előrejelző képességeit, melyhez 13 gépi tanulási modellt építettünk és futtattunk. A tanulási feladatban a fa-alapú algoritmusok bizonyultak a legsikeresebbnek, melyet a magas F-measure értékek támasztanak alá. 0,77-es és 0,71-es átlagos F-measure értéket sikerült elérnünk, rendre osztály és fájl szinten. A fájl szintű értékek alacsonyabbak, mivel ezen a szinten kevesebb forráskód metrika áll a rendelkezésünkre, így kevésbé szofisztikált modelleket tudtunk csak építeni. Ahhoz, hogy átfogó eredményeket kapjunk, a modellek hiba lefedettségi képességét is megvizsgáltuk. Közel tökéletes lefedettséget tudtunk elérni úgy, hogy a forráskód elemek mindössze 31%-át jelöltük meg hibásként mind osztály, mind fájl szinten. Amennyiben a *precision* érték fontosabb, mint a *recall*, akkor érdemesebb lehet a Naive Bayes módszert használni a hiba előrejelzéshez.

Egységesített hiba adatbázis és annak felhasználása hiba előrejelzéshez: A létező hiba adatbázisok feltérképezése közben megfogalmazódott bennünk egy igény, miszerint szükség volna egy egységesített hiba adatbázisra, annak érdekében, hogy a hiba előrejelző módszereket egy sokkal általánosabb, sokrétűbb adathalmazon értékelhessük ki. Így egy kimerítő szakirodalom kutatásba kezdtünk, hogy az összes lehetséges hiba adatbázist felderítsük. 5 jelöltet sikerült találnunk, melyek között a saját GitHub hiba adatbázisunk is megtalálható. Ezeket az adatbázisokat fésültük össze egy nagy és egységes hiba adatbázisba. Az OpenStaticAnalyzer (OSA) nyílt forráskódú statikus elemzőt használtuk, hogy a több, mint 50 statikus metrikát kinyerjük a különböző rendszerekre. A létrejövő hiba adatbázisban, így a kiszámított metrikák halmaza egységes lett, ugyanakkor a forráskód elemekhez meghagytuk az eredeti hibaszámokat (az eredeti elemeket az új elemzés eredményeivel a standard Java osztály nevek alapján, míg a fájlokat egyszerűen a nevük alapján párosítottuk). A forráskód elemek párosítása közben fellépő anomáliákat manuális ellenőriztük az okok felderítéséhez. Az egyik fő ok az olyan elemek jelenléte az eredeti adatbázisokban, melyek nem is igazi Java forráskódot tartalmazó állományok (Scala forrásfájlok, package-info fájlok). 624 osztály és 28 fájl szintű adatbázis bejegyzést nem tudtunk párosítani az új megfelelőjükkel. Az arányok végett, 48 242 osztály és 43 722 fájl szintű bejegyzés volt megtalálható az eredeti adatbázisokban. Ez összességében azt jelenti, hogy az elemek 0,71%-át (652 a 92 014-ből) kellett kihagynunk az egységesített hiba adatbázisból. Az egységesítés folyamata közben megvizsgáltuk az eredeti és az új metrika halmazok közötti átfedéseket (a közös metrikákat), hogy képet kapjunk azok különbözőségeiről. Feltártuk a különböző nevezéktani, illetve definíciós eltéréseket is egyaránt. Szignifikáns eltéréseket találtunk a legegyszerűbb metrikák között is – mint amilyen például a logikai kódsorok száma (LLOC) – mely az elemzőeszközök eltéréséből fakad, ugyanis egyes eszközök forráskódon dolgoznak, míg mások a bájt kódot használják fel bemenetként. Az adatbázisokat megvizsgáltuk a meta adatok szempontjából is, illetve funkcionális követelmények alapján is. A meta adatok tartalmazzák a használt statikus elemzőt, a granularitást (osztály vagy fájl szintű elemeket tartalmaz az adatbázis), a használt hiba- és verziókövető rendszert, illetve a számított forráskód metrikák halmazát. Egy funkcionális követelményként összehasonlítottuk az eredeti és az új metrikák hiba előrejelző képességét, illetve azt is megvizsgáltuk, hogy ehhez képest együttesen hogyan

teljesítenek. A vizsgálathoz a J48 döntési fát (a C4.5 döntési fa egy implementációja) használtuk a Weka gépi tanulási keretrendszerből. A hiba előrejelző modelleket külön-külön tanítottuk az egyes projekteken, melynek során 10-szeres keresztvalidációt alkalmaztunk. Egy további funkcionális követelményként különböző rendszereket használtunk fel tanításhoz, mind pedig kiértékeléshez, melyet kereszt projektes tanításnak nevezünk. A hiba adatbázisban szereplő összes projekt párosításra elvégeztük ezt a kísérletet. Az eredmények közül a GitHub hiba adatbázis projektjeire vett részhalmaz a leginkább konzisztens. Összességében a kísérletek megmutatták, hogy az egységes hiba adatbázis hatékonyan használható hiba előrejelzéshez, melyet az elért magas F-measure értékek (0,8 és a fölötti) támasztanak alá.

Jelen tézispont a publikus hiba adatbázisokra és azok kiterjesztésére fókuszál. Létrehoztunk egy korszerű, statikus forráskód metrikákat használó, publikus adatabázist, a GitHub hiba adatbázist. Következő lépésként, 5 publikusan elérhető hiba adatbázist egyesítettünk egy nagy közös adatbázisba, annak érdekében, hogy egy erős alapot biztosítsunk a statikus forráskód metrika alapú hiba előrejelző modellek építéséhez és teszteléséhez. A tézispontban bemutatott új hiba adatbázisok nyilvánosan elérhetőek.

A szerző hozzájárulása

A disszertáció szerzője tervezte meg a GitHub-ról történő hiba adatok kinyerésére irányuló módszert, illetve az ezen adatokból történő GitHub hiba adatbázis felépítésének módszerét. A szerző kutatta fel a publikus hiba adatbázisokkal kapcsolatos szakirodalmat, illetve gyűjtötte össze az adatbázisokhoz tartozó mindennemű információt. A GitHub hiba adatbázisba beválogatni kívánt projektekkel kapcsolatos elvárások megfogalmazásában is aktívan szerepet vállalt. A szerző készítette el a statisztikákat az egységesített hiba adatbázisban szereplő összes projekthez. Ezenfelül, a statikus forráskód elemzését is ő végezte el az egységes hiba adatbázisban található projektekre. A különböző adatbázisokban használt metrika halmazok összegyűjtése és összehasonlítása, továbbá a meta adatok feltérképezése a szerző saját munkáját képezik. A szerző aktívan részt vett továbbá a hiba előrejelző modellek építésében, mind a GitHub, mind az egységesített hiba adatbázisok esetében. A végső gépi tanulások eredményeit a szerző állította elő, továbbá az azokból levonható következtetéseket is ő fogalmazta meg. A tézisponthoz tartozó alátámasztó publikációk a következőek:

- **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.

II. Egy módszertan az RPG szoftverrendszerek karbantarthatóságának mérésére

A tézispontban az RPG hatyaték rendszerek karbantarthatóságának mérésével foglalkozunk, melynek eredményeit a 5., 6., 7. és 8. fejezetek tárgyalják. Az RPG rendszerek elemzése valós ipari igényen alapszik. Ezen kutatást az R&R Software Kft.-vel közösen vittük véghez, amely cég rendkívül nagy múlttal rendelkezik az RPG rendszerek tervezésében és megvalósításában. A cég vetette fel igényét az RPG nyelvű rendszereik statikus forráskód elemzésére, illetve a rendszerek karbantarthatóságának szofisztikált és megbízható mérésére. A tézispontban található kutatási eredményeket a GOP-1.1.1-11-2012-0323 témaszámú pályázat keretein belül készítettük el.

Szoftverminőség méréshez használható RPG statikus forráskód elemzők összehasonlítása: A kutatás célja, hogy a jelenlegi legkorszerűbb RPG statikus forráskód elemzőket mélyrehatóan és objektíven összehasonlítsa. A kutatási terület a statikus elemzőkből kinyert adatokra fókuszál, amelyeket később magasabb szintű minőségleíró attribútumok meghatározására használhatunk. A SourceMeter egy saját fejlesztésű parancssori eszköz, mely a forráskód különböző tulajdonságait adja eredményül, mint például statikus forráskód metrikákat, kódolási szabálysértéseket és kód duplikációkat. A SonarQube egy minőség menedzsment platform, mely rendelkezik RPG nyelvi támogatással is. Az objektív összehasonlítás megkönnyítése érdekében, a *SourceMeter for RPG* SonarQube-os beépülő modulját (plugin) használtuk, amely kiterjeszti a SonarQube funkcionalitását a SourceMeter képességeivel. Ilyen módon a vizsgált rendszerek egy közös interfésszel rendelkeztek, amely jelentősen megkönnyítette az összehasonlítás elvégzését. Az összehasonlítás bementét 179 rendszer képezte. Az összehasonlításban szereplő szempontok között megtalálható az elemzés sikeressége és mélysége, a forráskód metrikák halmaza, a támogatott kódolási szabálysértések halmaza, illetve a kód duplikációk felfedezésének képessége. A SourceMeter eszköz gond nélkül elemezte az összes forráskódot, míg a SonarQube esetében 3 fájlt nem sikerült a rendszernek analizálni, mivel azok nem támogatott nyelvi konstrukciókat tartalmaztak (például free-form blokkot). A SourceMeter 4 szinten szolgáltatja az eredményeket: *rendszer, program (fájl), procedúra* és *szubrutin* szinten. A SonarQube ehhez képest csak *rendszer* és *fájl* szinteket támogat. A SonarQube limitált mennyiségű statikus forráskód metrikát szolgáltat a SourceMeterhez képest, mely utóbbi a metrikákat alacsonyabb szinteken is számolja (szubrutin és procedúra). A közösen támogatott szabályok halmaza nagy, továbbá mindkét eszköz biztosít olyan kódolási szabályokat, melyeket a másik nem. A SourceMeter eszköz rendelkezik metrika-alapú szabályokkal is, melyek alapján szabálysértést generál a rendszer, ha egy adott forráskód elemhez tartozó metrika értéke kisebb vagy nagyobb, mint az engedélyezett intervallum alsó vagy felső végpontja. A SonarQube eszköz képes az 1-es típusú kód klónok felderítésére, melyek másolás és beillesztés technikával jöttek létre. A SourceMeter ugyanakkor egy absztrakt szintaxis fát (Abstract Syntax Tree - AST) épít, mely a klón detektálás bemenetét képezi, így az képes a 2-es típusú klónok felderítésére is (például a különböző változónevek használatával nem lehet megkerülni a klón megtalálását). Az eszközök kiértékelése után, az alacsony szintű karakterisztikák magasabb szintű jellemzőkre gyakorolt hatását vizsgáltuk meg a platformon, azaz a minőségi indexeket vettük górcső alá. A *technical dept* és a *SQALE* metikákat vettük figyelembe,

melyeket a SonarQube szolgáltat. Ezen jellemzők erősen építenek a szabálysértésekre, viszont más alacsony szintű jellemzőket (a forráskód metrikákat és a kód duplikációkat) nem vesznek figyelembe, így ezek a jellemzők nem képesek megfelelő módon tükrözni a rendszer teljes minőségét. A SourceMeter eszközt találtuk fejlettebbnek az elemzés mélységében, a metrikák és a kód klónok tekintetében, míg a kódolási szabályok és az elemzés sikerességét tekintve teljesítményük közel azonos. Ezen faktorokat tekintve a SourceMeter eszközt választottuk ki arra a célra, hogy alacsony szintű jellemzőket biztosítson a szofisztikáltabb, magasabb szintű tulajdonságok meghatározásához.

Folyamatos minőség monitorozás integrációja egy meglévő fejlesztési folyamatba – egy esettanulmány: A kutatás célja egy olyan általános és rugalmas minőség modell megalkotása az RPG nyelvre, melyet aztán sikeresen alkalmazhatunk az esettanulmányunkban. Egy jó minőség modell egyet jelent olyan magas szintű tulajdonságokkal, mint amilyenek a minőségi index vagy a karbantarthatósági index, melyek többet árulnak el a rendszerről. A kreált minőség modell az ISO/IEC 25010-es szabványon alapszik, melyben a karbantarthatóság, mint kulcs tényező játszik szerepet egy rendszer minőségének meghatározásában (ilyen módon a minőség modell és karbantarthatóság modell kifejezéseket felváltva használjuk a karbantarthatósági modellre, mely valójában csak egy részét képezi a legfelsőbb szintű minőség jellemzőnek). A modell erősen épít az újrafelhasználhatóságára, elemezhetőségére, módosíthatóságára és tesztelhetőségére, mint altulajdonságokra, melyek meghatározzák a rendszer tényleges karbantarthatóságát. Ezek a magasabb szintű jellemzők az alacsony szintú tulajdonságokat használják fel, melyeket a SourceMeter szolgáltat. A minőség modell definiálása után, kiviteleztünk egy esettanulmányt, melyben a modellt beépítettük egy közepes méretű szoftverfejlesztő cég, az R&R Software Kft., fejlesztési folyamataiba. Az *inicializációs fázisban* a SourceMeter finomhangolása (például a metrika alapú szabályok lehetséges intervallumainak beállítása, tiltott operációk meghatározása) és a minőség modell paraméterezése után (súlyok meghatározása), létrehoztunk egy benchmark-ot. A második, *integrációs fázisban* adaptáltuk a minőség monitorozó módszerünket, mely észrevétlenül épül be a vállalat fejlesztési folyamataiba. Az új változtatások során (commit) automatikusan lefut a forráskód elemzés, melynek eredményeit adatbázisban tároljuk, biztosítva azok hosszú távú elérhetőséget. A vállalat egy megadott modul minőségén szeretett volna javítani, így egy *refaktorálási fázisban* a vállalat néhány fejlesztője csökkentette a rendszerben lévő kritikus és súlyos szabálysértések számát. Ezen tevékenység segítette, hogy a rendszer karbantarthatósága verzióról verzióra növekedhessen. Az *értékelés fázisban* levonhattuk a konklúziót, mely szerint a karbantarthatóság javult a kiválasztott modulon, így elérve a „GO" állapotot, vagyis a megadott minimális karbantarthatósági szintet, amellyel a rendszer már kiadható. A vállalat fejlesztői és menedzserei szerint a módszerünk ipari környeztben történő alkalmazása sikeresnek bizonyult. A vállalaton belül sikeresen tudták ellenőrizni a kódolási konvenciók követését, amely kikényszeríti, hogy a fejlesztők elkerüljenek bizonyos rossz megoldásokat, helyettük bevett technkikákat tanuljanak meg és alkalmazzák azokat a gyakori problémákra. Ezen felül a megközelítésünket kellőképpen rugalmasnak és testre szabhatónak találták (például a benchmark létrehozás és a minőség modell súlyozása gyorsan megtanulható). A fejlesztők szerint a karbantartási munkálatokat hatékonyan tudták elvégezni, melyet a SourceMeter és

QualityGate eszközöknek köszönhetnek.

Halstead komplexitás metrikák és a karbantarthatósági index alkalmazása RPG környezetben: A kutatás célja, hogy a minőség modellbe további lehetséges jellemzőket vonjunk be, melyek továbbfejlesztik a modell kifejező képességét. Első lépésként meg kellett határoznunk a Halstead komplexitás metrikák RPG/400 és RPG IV környezetre értelmezhető definícióját. Sajnos nincs standard módja ezen metrikák számításának, mivel a különböző programozási nyelvek különböző nyelvi konstrukciókat tartalmaznak, melyekről sokszor nehéz megállapítani, hogy operandusként avagy operátorként számoljuk őket. RPG esetében egy korábbi definíciós halmaz már létezett az RPG II és RPG III nyelvekre, így ezeket terjesztettük ki az RPG/400-ban és az RPG IV-ben bemutatott számos újabb nyelvi konstrukcióra. Következő lépésként megvizsgáltuk a Halstead komplexitás metrikákat, illetve a karbantarthatósági index négy variánsát, hogy ezek milyen korrelációban állnak a többi metrikával (melyek a minőség modellben szerepelnek), továbbá kerestük ezen metrikák alkalmazhatóságát a minőség modell kiterjesztésére, hogy az még jobban le tudja írni a szóban forgó rendszer minőségét. Ehhez a feladathoz főkomponens analízist (Principal Component Analysis - PCA) alkalmaztunk. Az analízis során azt tapasztaltuk, hogy a Halstead komplexitás metrikák egy erősen összefüggő csoportot alkotnak a metrikák halmazán belül, továbbá jól felhasználhatóak az elemzett rendszer jobb leírásához. Végső konklúzióként a Halstead Number of Delivered Bugs (HNDB) metrikát javasoljuk belevenni a minőség modellbe, azon belül is a hibahajlam (fault proneness) belső pont gyermekeként, mivel a legnagyobb korrelációs együtthatót a lehetséges hibákkal (warning) produkálta ez a metrika. A HNDB metrika mellett a Halstead Program Vocabulary (HPV) metrikát is ajánlatos belevenni a minőség modell komplexitás belső pontja alá, mivel alacsony korrelációt mutat a McCabe-féle komplexitás metrikával (szubrutinok szintjén), ugyanakkor nagy súllyal szerepel a faktoranalízis lineáris kombinációjának első, azaz legdominánsabb komponensében. Ilyen módon a rendszer komplexitását a McCabe-féle komplexitás, az NLE (elágazások egymásba ágyazottsága) és a HPV metrikák megfelelő súlyozásával alkotott kombinációja alkotnák, melyek összességében jobban, több szemszögből írnák le a rendszert.

Ez a tézispont az RPG programozási nyelven készült rendszerekre alkalmazható statikus forráskód elemezőkkel és arra épülő minőségbiztosítási módszerekkel foglalkozik. A SourceMeter eszköz bizonyult megfelelő választásnak a statikus szoftver metrikák, a kód klónok, valamint az elemzés mélysége, részletessége szempontjából, így annak eredményeit használtuk fel a magasabb szintű minőségi jellemzők kinyerésére. Ezen magasabb szintű jellemzők előállítása végett egy minőség modell definiáltunk az RPG nyelvre. A folyamatos minőség monitorozás integrációja – egy közepes méretű vállalat fejlesztési folyamataiba – sikeresnek bizonyult. Az integráció sikerességén felbuzdulva, tovább szerettük volna javítani a karbantarthatóság mérésére adott módszerünket, így ajánlásokat tettünk a minőség modell továbbfejlesztésére. A főkomponens analízis eredményei alapján megfogalmaztunk egy végső javaslatot, mely szerint a HNDB (Halstead Number of Delivered Bugs) és a HPV (Halstead Program Vocabulary) metrikákat ajánljuk a modellbe építeni, mivel ezek növelik a leghatékonyabban annak minőségleíró képességét a vizsgált lehetséges metrikák közül.

A szerző hozzájárulása

A *SourceMeter for RPG* statikus forráskód elemző kifejlesztése, melynek segítségével RPG/400-as és RPG IV-es rendszereket is elemezhetünk a szerző munkája. Ez az elemző eszköz képezi az egyik legfontosabb alapját az összehasonlításnak, illetve a minőség modellnek is egyaránt. A szerző gyűjtötte össze és dolgozta fel az RPG szoftver rendszerek minőségbiztosításával foglalkozó szakirodalmat. Az elemző eszközök összehasonlításához, illetve a benchmark építéshez használt programokat a szerző gyűjtötte össze. Az összes statikus elemzést a szerző végezte el, illetve azok eredményeit a szerző gyűjtötte össze és dolgozta fel (mind a statikus elemzők összehasonlításánál, mind a minőség modell kiterjesztésénél). Részt vett továbbá az esettanulmány megszervezésében és kivitelezésében. A főkomponens analízis elvégzése és annak eredményei alapján a minőség modell kiterjesztésére vonatkozó javaslatok megfogalmazása a szerző munkáját képezik. A téziponthoz kapcsolódó alátámasztó publikációk:

- **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: 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.

A tézispontokat és a kapcsolódó publikációkat a B.1. táblázat összegzi.

№	[118]	[114]	[119]	[115]	[117]
I.	♦	♦			
II.			♦	♦	♦

B.1. táblázat. A tézispontokhoz kapcsolódó publikációk

Acknowledgement

This thesis was supported by the EU-funded Hungarian national grant GINOP-2.3.2-15-2016-00037 titled “Internet of Living Things”.

Bibliography

[1] JHawk. http://www.virtualmachinery.com/jhawkprod.htm, 2018. Accessed: 2018-01-25.

[2] Boeing Scandal. https://www.theguardian.com/commentisfree/2019/apr/07/boeing-737-max-regulation-corporate-america, 2019. Accessed: 2019-05-20.

[3] CrossBrowserTesting. https://crossbrowsertesting.com/blog/development/software-bug-cost/, 2019. Accessed: 2019-05-20.

[4] Tiago L Alves, Pedro Silva, and Miguel Sales Dias. Applying ISO/IEC 25010 Standard to prioritize and solve quality issues of automatic ETL processes. *IEEE International Conference on Software Maintenance (ICSM 2014)*, pages 573–576, 2014.

[5] Tiago L. Alves, Christiaan Ypma, and Joost Visser. Deriving Metric Thresholds from Benchmark Data. In *Proceedings of the 26th IEEE International Conference on Software Maintenance (ICSM 2010)*, 2010.

[6] Erik Arisholm and Lionel C Briand. Predicting fault-prone components in a java legacy system. In *Proceedings of the 2006 ACM/IEEE international symposium on Empirical software engineering*, pages 8–17. ACM, 2006.

[7] Robert Baggen, Katrin Schill, and Joost Visser. Standardized Code Quality Benchmarking for Improving Software Maintainability. In *Proceedings of the Fourth International Workshop on Software Quality and Maintainability (SQM 2010)*, 2010.

[8] Ger Bakker and Fred Hirdes. Recent industrial experiences with software product metrics. In *Objective Software Quality*, pages 179–191. Springer, 1995.

[9] 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.

[10] 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)*, Riva del Garda, Italy, 2012. IEEE Computer Society.

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

[12] P. Bangcharoensap, A. Ihara, Y. Kamei, and K. Matsumoto. Locating source code to be fixed based on initial bug reports - a case study on the eclipse project. In *Empirical Software Engineering in Practice (IWESEP), 2012 Fourth International Workshop on*, pages 10–15, Oct 2012.

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

[14] Vera Barstad, Morten Goodwin, and Terje Gjøsæter. Predicting source code quality with static analysis and machine learning. In *NIK*, 2014.

[15] Victor R Basili, Lionel C. Briand, and Walcélio L Melo. A validation of object-oriented design metrics as quality indicators. *IEEE Transactions on software engineering*, 22(10):751–761, 1996.

[16] V.R. Basili, L.C. Briand, and W.L. Melo. A validation of object-oriented design metrics as quality indicators. *Software Engineering, IEEE Transactions on*, 22(10):751 –761, oct 1996.

[17] Dilek Baski and Sanjay Misra. Metrics suite for maintainability of extensible markup language web services. *IET software*, 5(3):320–341, 2011.

[18] Ira D Baxter, Andrew Yahin, Leonardo Moura, Marcelo Sant'Anna, and Lorraine Bier. Clone detection using abstract syntax trees. In *Proceedings. International Conference on Software Maintenance (Cat. No. 98CB36272)*, pages 368–377. IEEE, 1998.

[19] Brett A Becker and Catherine Mooney. Categorizing compiler error messages with principal component analysis. In *12th China-Europe International Symposium on Software Engineering Education (CEISEE 2016), Shenyang, China, 28-29 May 2016*, 2016.

[20] Christian Bird, Peter C Rigby, Earl T Barr, David J Hamilton, Daniel M German, and Prem Devanbu. The promises and perils of mining git. In *2009 6th IEEE International Working Conference on Mining Software Repositories*, pages 1–10. IEEE, 2009.

[21] Cathal Boogerd and Leon Moonen. Assessing the Value of Coding Standards: An Empirical Study. In *Proceedings of the 24th IEEE International Conference on Software Maintenance (ICSM 2008)*, pages 277–286. IEEE, 2008.

[22] David Bowes, Tracy Hall, Mark Harman, Yue Jia, Federica Sarro, and Fan Wu. Mutation-aware fault prediction. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, pages 330–341. ACM, 2016.

[23] Lionel C. Briand, John W. Daly, and Jurgen K Wust. A unified framework for coupling measurement in object-oriented systems. *IEEE Transactions on software Engineering*, 25(1):91–121, 1999.

[24] Frederick P Brooks Jr. The mythical man-month (anniversary ed.). 1995.

[25] William H Brown, Raphael C Malveau, Hays W McCormick, and Thomas J Mowbray. *AntiPatterns: refactoring software, architectures, and projects in crisis*. John Wiley & Sons, Inc., 1998.

[26] Magiel Bruntink and Arie Van Deursen. Predicting class testability using object-oriented metrics. In *Source Code Analysis and Manipulation, 2004. Fourth IEEE International Workshop on*, pages 136–145. IEEE, 2004.

[27] Gerardo Canfora, Andrea De Lucia, and Giuseppe A Di Lucca. An incremental object-oriented migration strategy for rpg legacy systems. *International Journal of Software Engineering and Knowledge Engineering*, 9(01):5–25, 1999.

[28] Peggy Cellier, Mireille Ducassé, Sébastien Ferré, and Olivier Ridoux. Multiple fault localization with data mining. In *SEKE*, pages 238–243, 2011.

[29] S.R. Chidamber and C.F. Kemerer. A metrics suite for object oriented design. *Software Engineering, IEEE Transactions on*, 20(6):476 –493, jun 1994.

[30] Don Coleman, Dan Ash, Bruce Lowther, and Paul Oman. Using metrics to evaluate software system maintainability. *Computer*, 27(8):44–49, 1994.

[31] H. Coles. PITest Mutation Framework. `http://pitest.org/`, 2018. Accessed: 2018-01-25.

[32] José Pedro Correia and Joost Visser. Benchmarking Technical Quality of Software Products. In *Proceedings of the 15th Working Conference on Reverse Engineering (WCRE 2008)*, pages 297–300, Washington, DC, USA, 2008. IEEE Computer Society.

[33] C. Couto, C. Silva, M.T. Valente, R. Bigonha, and N. Anquetil. Uncovering causal relationships between software metrics and bugs. In *Software Maintenance and Reengineering (CSMR), 2012 16th European Conference on*, pages 223–232, March 2012.

[34] Valentin Dallmeier and Thomas Zimmermann. Automatic extraction of bug localization benchmarks from history. In *Proc. Int'l Conf. on Automated Software Eng*, pages 433–436. Citeseer, 2007.

[35] Valentin Dallmeier and Thomas Zimmermann. Extraction of bug localization benchmarks from history. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pages 433–436. ACM, 2007.

[36] Marco D'Ambros, Michele Lanza, and Romain Robbes. An extensive comparison of bug prediction approaches. In *2010 7th IEEE Working Conference on Mining Software Repositories (MSR 2010)*, pages 31–41. IEEE, 2010.

[37] Marco D'Ambros, Michele Lanza, and Romain Robbes. Evaluating defect prediction approaches: a benchmark and an extensive comparison. *Empirical Software Engineering*, 17(4-5):531–577, 2012.

[38] Andrea De Lucia, Rita Francese, Giuseppe Scanniello, and Genoveffa Tortora. Developing legacy system migration methods and tools for technology transfer. *Softw. Pract. Exper*, 38:1333–1364, 2008.

[39] R. Ferenc, I. Siket, and T. Gyimothy. Extracting facts from open source software. In *Software Maintenance, 2004. Proceedings. 20th IEEE International Conference on*, pages 60 – 69, sept. 2004.

[40] David Gray, David Bowes, Neil Davey, Y Sun, and Bruce Christianson. Reflections on the nasa mdp data sets. *IET software*, 6(6):549–558, 2012.

[41] David Gray, David Bowes, Neil Davey, Yi Sun, and Bruce Christianson. The misuse of the nasa metrics data program data sets for automated software defect prediction. In *Evaluation & Assessment in Software Engineering (EASE 2011), 15th Annual Conference on*, pages 96–103. IET, 2011.

[42] Tibor Gyimothy, Rudolf Ferenc, and Istvan Siket. Empirical validation of object-oriented metrics on open source software for fault prediction. *Software Engineering, IEEE Transactions on*, 31(10):897–910, 2005.

[43] Mark Hall, Eibe Frank, Geoffrey Holmes, Bernhard Pfahringer, Peter Reutemann, and Ian H Witten. The weka data mining software: an update. *ACM SIGKDD explorations newsletter*, 11(1):10–18, 2009.

[44] Tracy Hall, Sarah Beecham, David Bowes, David Gray, and Steve Counsell. A systematic literature review on fault prediction performance in software engineering. *IEEE Transactions on Software Engineering*, 38(6):1276–1304, 2012.

[45] Tracy Hall, Min Zhang, David Bowes, and Yi Sun. Some code smells have a significant but small effect on faults. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 23(4):33, 2014.

[46] Maurice Howard Halstead. *Elements of software science*, volume 7. Elsevier New York, 1977.

[47] Mark Harman, Syed Islam, Yue Jia, Leandro L Minku, Federica Sarro, and Komsan Srivisut. Less is more: Temporal fault predictive performance over multiple hadoop releases. In *International Symposium on Search Based Software Engineering*, pages 240–246. Springer, 2014.

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

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

[50] Péter Hegedűs. A probabilistic quality model for c#-an industrial case study. *Acta Cybern.*, 21(1):135–147, 2013.

[51] Ilja Heitlager, Tobias Kuipers, and Joost Visser. A practical model for measuring maintainability. In *Quality of Information and Communications Technology, 2007. QUATIC 2007. 6th International Conference on the*, pages 30–39. IEEE, 2007.

[52] J Horning, H Lauer, P Melliar-Smith, and Brian Randell. A program structure for error detection and recovery. *Operating Systems*, pages 171–187, 1974.

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

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

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

[56] Ho-Won Jung, Seung-Gweon Kim, and Chang-Shin Chung. Measuring Software Product Quality: A Survey of ISO/IEC 9126. *IEEE Software*, pages 88–92, 2004.

[57] René Just, Darioush Jalali, and Michael D Ernst. Defects4j: A database of existing faults to enable controlled testing studies for java programs. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, pages 437–440. ACM, 2014.

[58] Eirini Kalliamvakou, Georgios Gousios, Kelly Blincoe, Leif Singer, Daniel M German, and Daniela Damian. The promises and perils of mining github. In *Proceedings of the 11th working conference on mining software repositories*, pages 92–101. ACM, 2014.

[59] Y. Kamei and E. Shihab. Defect prediction: Accomplishments and future challenges. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, volume 5, pages 33–45, March 2016.

[60] Stephen H Kan, SD Dull, DN Amundson, Richard J. Lindner, and RJ Hedger. AS/400 software quality management. *IBM Systems Journal*, 33(1):62–88, 1994.

[61] Matinee Kiewkanya, Nongyao Jindasawat, and Pornsiri Muenchaisri. A methodology for constructing maintainability model of object-oriented design. In *Quality Software, 2004. QSIC 2004. Proceedings. Fourth International Conference on*, pages 206–213. IEEE, 2004.

[62] Sunghun Kim, Thomas Zimmermann, E James Whitehead Jr, and Andreas Zeller. Predicting faults from cached history. In *Proceedings of the 29th international conference on Software Engineering*, pages 489–498. IEEE Computer Society, 2007.

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

[64] Anuradha Lakshminarayana and Timothy S Newman. Principal component analysis of lack of cohesion in methods (lcom) metrics. *Technical Report TRUAH-CS-1999-01*, 1999.

[65] Michele Lanza and Radu Marinescu. *Object-oriented metrics in practice: using software metrics to characterize, evaluate, and improve the design of object-oriented systems*. Springer Science & Business Media, 2007.

[66] Ruchika Malhotra, Shivani Shukla, and Geet Sawhney. Assessment of defect prediction models using machine learning techniques for object-oriented systems. In *Reliability, Infocom Technologies and Optimization (Trends and Future Directions)(ICRITO), 2016 5th International Conference on*, pages 577–583. IEEE, 2016.

[67] Thomas J McCabe. A complexity measure. *IEEE Transactions on software Engineering*, (4):308–320, 1976.

[68] Tim Menzies, Bora Caglayan, Zhimin He, Ekrem Kocaguneli, Joe Krall, Fayola Peters, and Burak Turhan. The promise repository of empirical software engineering data, June 2012.

[69] Tim Menzies, Jeremy Greenwald, and Art Frank. Data mining static code attributes to learn defect predictors. *IEEE transactions on software engineering*, 33(1), 2007.

[70] Ayse Tosun Misirli, Ayse Bener, and Resat Kale. Ai-based software defect predictors: Applications and benefits in a case study. *AI Magazine*, 32(2):57–68, 2011.

[71] Sanjay Misra, Ibrahim Akman, and Ricardo Colomo-Palacios. Framework for evaluation and validation of software complexity measures. *IET software*, 6(4):323–334, 2012.

[72] Sanjay Misra, Murat Koyuncu, Marco Crasso, Cristian Mateos, and Alejandro Zunino. A suite of cognitive complexity metrics. *Computational Science and Its Applications–ICCSA 2012*, pages 234–247, 2012.

[73] Raimund Moser, Witold Pedrycz, and Giancarlo Succi. A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction. In *Proceedings of the 30th international conference on Software engineering*, pages 181–190. ACM, 2008.

[74] S Muthanna, Kostas Kontogiannis, Kumaraswamy Ponnambalam, and B Stacey. A maintainability model for industrial software systems using design level metrics. In *Reverse Engineering, 2000. Proceedings. Seventh Working Conference on*, pages 248–256. IEEE, 2000.

[75] Nachiappan Nagappan and Thomas Ball. Use of relative code churn measures to predict system defect density. In *Software Engineering, 2005. ICSE 2005. Proceedings. 27th International Conference on*, pages 284–292. IEEE, 2005.

[76] Nachiappan Nagappan, Thomas Ball, and Andreas Zeller. Mining metrics to predict component failures. In *Proceedings of the 28th international conference on Software engineering*, pages 452–461. ACM, 2006.

[77] Csaba Nagy, László Vidács, Rudolf Ferenc, Tibor Gyimóthy, Ferenc Kocsis, and István Kovács. Magister: Quality assurance of magic applications for software developers and end users. In *IEEE International Conference on Software Maintenance (ICSM 2010)*, pages 1–6. IEEE, Sep 2010.

[78] Csaba Nagy, László Vidács, Rudolf Ferenc, Tibor Gyimóthy, Ferenc Kocsis, and István Kovács. Complexity measures in 4gl environment. In *Computational Science and Its Applications - ICCSA 2011, Lecture Notes in Computer Science*, volume 6786 of *Lecture Notes in Computer Science*, pages 293–309. Springer, 2011.

[79] Farid A Naib. An application of software science to the quantitative measurement of code quality. In *ACM SIGMETRICS Performance Evaluation Review*, volume 11, pages 101–128. ACM, 1982.

[80] Paul Oman and Jack Hagemeister. Construction and testing of polynomials predicting software maintainability. *Journal of Systems and Software*, 24(3):251–266, 1994.

[81] Maryoly Ortega, María Pérez, and Teresita Rojas. Construction of a systemic quality model for evaluating a software product. *Software Quality Journal*, 11(3):219–242, 2003.

[82] Thomas J Ostrand, Elaine J Weyuker, and Robert M Bell. Predicting the location and number of faults in large software systems. *Software Engineering, IEEE Transactions on*, 31(4):340–355, 2005.

[83] Girish Parikh and Nicholas Zvegintzov. The world of software maintenance. *Tutorial on Software Maintenance*, pages 1–3, 1983.

[84] Jean Petrić, David Bowes, Tracy Hall, Bruce Christianson, and Nathan Baddoo. The jinx on the nasa software defect data sets. In *Proceedings of the 20th International Conference on Evaluation and Assessment in Software Engineering*, page 13. ACM, 2016.

[85] Mario Piattini, Coral Calero, and Marcela Genero. Table oriented metrics for relational databases. *Software Quality Control*, 9(2):79–97, 2001.

[86] Shruthi Puranik, Pranav Deshpande, and K Chandrasekaran. A novel machine learning approach for bug prediction. *Procedia Computer Science*, 93:924–930, 2016.

[87] QualityGate quality management platform. `http://www.quality-gate.com`, 2015.

[88] Brian Randell. System structure for software fault tolerance. *IEEE Transactions on Software Engineering*, 10(2):220–232, 1975.

[89] Adnan Rawashdeh and Bassem Matalkah. A new software quality model for evaluating cots components. *Journal of Computer Science*, 2(4):373–381, 2006.

[90] Gregorio Robles. Replicating msr: A study of the potential replicability of papers published in the mining software repositories proceedings. In *Mining Software Repositories (MSR), 2010 7th IEEE Working Conference on*, pages 171–180. IEEE, 2010.

[91] Martin Shepperd and Darrel C Ince. A critique of three metrics. *Journal of Systems and Software*, 26(3):197–210, 1994.

[92] Martin Shepperd, Qinbao Song, Zhongbin Sun, and Carolyn Mair. Data quality: Some comments on the nasa software defect datasets. *IEEE Transactions on Software Engineering*, 39(9):1208–1215, 2013.

[93] Thomas Shippey, Tracy Hall, Steve Counsell, and David Bowes. So you need more method level datasets for your software defect prediction?: Voila! In *Proceedings of the 10th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, ESEM '16, pages 12:1–12:6, New York, NY, USA, 2016. ACM.

[94] SonarQube quality management platform. `https://www.sonarqube.org/`, 2019.

[95] SourceMeter. `http://www.sourcemeter.com`, 2019.

[96] Ramanath Subramanyam and Mayuram S. Krishnan. Empirical analysis of ck metrics for object-oriented design complexity: Implications for software defects. *Software Engineering, IEEE Transactions on*, 29(4):297–310, 2003.

[97] K. Suntiparakoo and Y. Limpiyakorn. Flowchart knowledge extraction on rpg legacy code. *Advanced Science and Technology Letters*, 29:258–263, 2013.

[98] Georg Von Krogh and Eric Von Hippel. The promise of research on open source software. *Management science*, 52(7):975–983, 2006.

[99] Shuo Wang and Xin Yao. Using class imbalance learning for software defect prediction. *Reliability, IEEE Transactions on*, 62(2):434–443, 2013.

[100] Elaine J Weyuker, Robert M Bell, and Thomas J Ostrand. Replicate, replicate, replicate. In *Replication in Empirical Software Engineering Research (RESER), 2011 Second International Workshop on*, pages 71–77. IEEE, 2011.

[101] Elaine J Weyuker, Thomas J Ostrand, and Robert M Bell. Comparing the effectiveness of several modeling methods for fault prediction. *Empirical Software Engineering*, 15(3):277–295, 2010.

[102] 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.

[103] Chadd C Williams and Jeffrey K Hollingsworth. Automatic mining of source code repositories to improve bug finding techniques. *Software Engineering, IEEE Transactions on*, 31(6):466–480, 2005.

[104] G.E. Witting and G.R Finnie. Using artificial neural networks and function points to estimate 4GL software development effort. *Australasian Journal of Information Systems*, 1(2), 1994.

[105] W Eric Wong, Vidroha Debroy, and Dianxiang Xu. Towards better fault localization: A crosstab-based statistical approach. *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, 42(3):378–396, 2012.

[106] W Eric Wong, Ruizhi Gao, Yihao Li, Rui Abreu, and Franz Wotawa. A survey on software fault localization. *IEEE Transactions on Software Engineering*, 42(8):707–740, 2016.

[107] Rongxin Wu, Hongyu Zhang, Sunghun Kim, and Shing-Chi Cheung. Relink: recovering links between bugs and changes. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, pages 15–25. ACM, 2011.

[108] Michalis Xenos, D Stavrinoudis, K Zikouli, and D Christodoulakis. Object-oriented metrics-a survey. In *Proceedings of the FESMA*, pages 1–10, 2000.

[109] Zhiwei Xu, Taghi M Khoshgoftaar, and Edward B Allen. Prediction of software faults using fuzzy nonlinear regression modeling. In *High Assurance Systems Engineering, 2000, Fifth IEEE International Symposim on. HASE 2000*, pages 281–290. IEEE, 2000.

[110] Zhe Yu, Nicholas A Kraft, and Tim Menzies. How to read less: Better machine assisted reading methods for systematic literature reviews. *arXiv preprint arXiv:1612.03224*, 2016.

[111] M. Zhivich and R. K. Cunningham. The real cost of software errors. *IEEE Security Privacy*, 7(2):87–90, March 2009.

[112] Thomas Zimmermann, Nachiappan Nagappan, Harald Gall, Emanuel Giger, and Brendan Murphy. Cross-project defect prediction: a large scale experiment on data vs. domain vs. process. In *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 91–100. ACM, 2009.

[113] Thomas Zimmermann, Rahul Premraj, and Andreas Zeller. Predicting defects for eclipse. In *Predictor Models in Software Engineering, 2007. PROMISE'07: ICSE Workshops 2007. International Workshop on*, pages 9–9. IEEE, 2007.

Corresponding Publications of the Author

[114] 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.

[115] 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.

[116] Z. Tóth, G. Novák, R. Ferenc, and I. Siket. Using version control history to follow the changes of source code elements. In *2013 17th European Conference on Software Maintenance and Reengineering*, pages 319–322, March 2013.

[117] 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).

[118] 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).

[119] 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).