Politecnico di Torino
Dottorato in Ingegneria Informatica e Dei Sistemi

EMPIRICAL ASSESSMENT OF THE IMPACT OF USING AUTOMATIC STATIC ANALYSIS ON SOFTWARE QUALITY

Ph.D. Dissertation of:

Antonio Vetro'

Advisor:

Prof. Maurizio Morisio

XXV Cycle

Politecnico di Torino
Dipartimento di Automatica e Informatica
Corso Duca degli Abruzzi, 24, 10139 Torino

ACKNOWLEDGEMENTS

This thesis closes an intensive chapter of my life.

When I began the PhD three years ago, I would have imagined having at this point clear and definitive answers not only to the questions of my research, but on my life as well. Three years later, I have much more open questions than I had at the beginning. For example, in my first year of PhD, when I finished my first experiment and I realized that I came out with a few answers and many new doubts, I was astonished and I was wondering whether a single hour of my work was valid. However, step by step, I understood that learning implies uncertainty by definition: when you learn, you widen your vision of things. Hence, after three years of research, I can get a bigger picture and have a wider landscape in front of me: that is why I can ask myself more and more questions. This is science, this is exploration.

I am grateful to my supervisor Maurizio Morisio to have wisely taught me how to deal with uncertainty: I learnt that a carefree spirit of adventure is the best way to face the questions. Not only in research, but in life as well. Moreso, I was so lucky to have even a second, "unofficial", supervisor: I am really grateful to Marco Torchiano. Observing the passion that he puts in understanding the reality with facts and numbers, I learnt that creativity and scientific rigour can coexist and you can do great things with both of them.

I spent my second year of the PhD at the Fraunhofer CESE, in Maryland, USA. That was a cornerstone experience of my life: very tough at the beginning, but when I had to leave I recognized that it was extremely generous and I had a lot of fun. The Wednesday meetings at 3 pm with Forrest Shull, Nico Zazworka and Carolyn Seaman were the best formative moments I ever had. I sincerely thank them for the time they spent in working together with me. I would also like to thank Mikael Lindvall and Michele Shaw for their precious suggestions and all the interns that were at Fraunhofer with me: I learnt a lot from their innovative ideas. My American adventure would have not been so cool without the

wonderful flatmates at 435 11th Street NE and my first Couchsurfing host: Annick, Henning, Natalie, James, you are awesome. Thanks for support and fun made in USA.

I was back in Turin for my third and final year of the PhD. In that period episodes in my life put me in difficulties, but I found at Politecnico nice people to have fun while working: that helped me in overcoming the storm. Giuseppe, Federico, Giuseppe, Luca and Andrea: thanks to you and to all previous and current people in Lab 1 to have made this place a bit more personal and unique for each of us.

I also would like to thank all technicians and clerks at DAUIN for their supporting work, all co-authors and the anonymous reviewers to have helped me improving my work, and all the companies that collaborated to my research.

As I wrote at the beginning of these acknowledgements, this work closes a chapter of my life. However, I'm sure that some protagonists appeared in this chapter will still be present in the next ones: I am talking about my closest friends. There's no need to mention them by name: you all know how much is important what we are building up, and how much is fun.

Finally, every building needs strong foundation. This PhD would have not been possible without my family: my dead and mum, my brothers, my sister in law and my wonderful, wonderful, nieces. I could never pay off what you have done and you do every day for me. However, I am strongly convinced that the best way I can thank you is to continue living all the next chapters of my life with passion and authenticity.

That’s all.

ABSTRACT

Automatic static analysis (ASA) tools analyze the source or compiled code looking for violations of recommended programming practices (called issues) that might cause faults or might degrade some dimensions of software quality.

Antonio Vetro' has focused his PhD in studying how applying ASA impacts software quality, taking as reference point the different quality dimensions specified by the standard ISO/IEC 25010. The epistemological approach he used is that one of empirical software engineering. During his three years PhD, he's been conducting experiments and case studies on three main areas: Functionality/Reliability, Performance and Maintainability. He empirically proved that specific ASA issues had impact on these quality characteristics in the contexts under study: thus, removing them from the code resulted in a quality improvement.

Vetro' has also investigated and proposed new research directions for this field: using ASA to improve software energy efficiency and to detect the problems deriving from the interaction of multiple languages.

The contribution is enriched with the final recommendation of a generalized process for researchers and practitioners with a twofold goal: improve software quality through ASA and create a body of knowledge on the impact of using ASA on specific software quality dimensions, based on empirical evidence.

This thesis represents a first step towards this goal.

Contents

List of Tables

List of Figures

1. INTRODUCTION

1.1. AUTOMATIC STATIC ANALYSIS

Source code analysis is a specific technique of reverse engineering [1] that consists in extracting information about a program from its source or artifacts (e.g., from Java byte code or execution traces) generated from the source code using automatic tools [2].

Automatic static analysis (ASA) tools analyze source or compiled code looking for violations of recommended programming practices ("issues") that might cause faults or might degrade some dimensions of software quality (e.g., maintainability, efficiency). The purpose of these tools is to extract some information from the source code or make judgments about it. The most common use of static analysis is in optimizing compilers. In fact, most of the high-level optimizations performed by a modern compiler depend on the results of static analyses such as control-flow and data-flow analysis. Outside of the compiler realm, static analysis techniques are used primarily for computing software metrics, in quality assurance, program understanding and refactoring. The area of our interest is quality assurance and refactoring.

The first static analysis tool, Lint, was developed more than 30 years ago by Stephen Johnson of Bell Labs [3]: it was able to examine C source programs that had compiled without errors and to find bugs that had escaped detection. Thirty years later, there are dozens of static analysis tools: comprehensive lists can be found both in the literature [4] [5] and on the Internet[1].

According to Li and Cui [4], currently-used static analysis techniques are: lexical analysis, type inference, theorem proving, data flow analysis, model checking and symbolic execution.

[1] http://en.wikipedia.org/wiki/List_of_tools_for_static_code_analysis

1.2. SOFTWARE QUALITY AND ASA

The international standard ISO/IEC 25010 [44] defines two models for software quality:

- a quality in use model;
- a product quality model.

The *quality in use model* refers to the final usage of software and its interaction with human beings. It can be applied to both computer systems in use and software in use. The *product quality model* refers to the static and dynamic properties of the computer systems: it can be applied both to computer systems and software products.

Both models are hierarchically composed of characteristics and sub-characteristics, which provide terminology for specifying, measuring and evaluating the system and software product quality. Such decomposition permits to represent the stated and implied

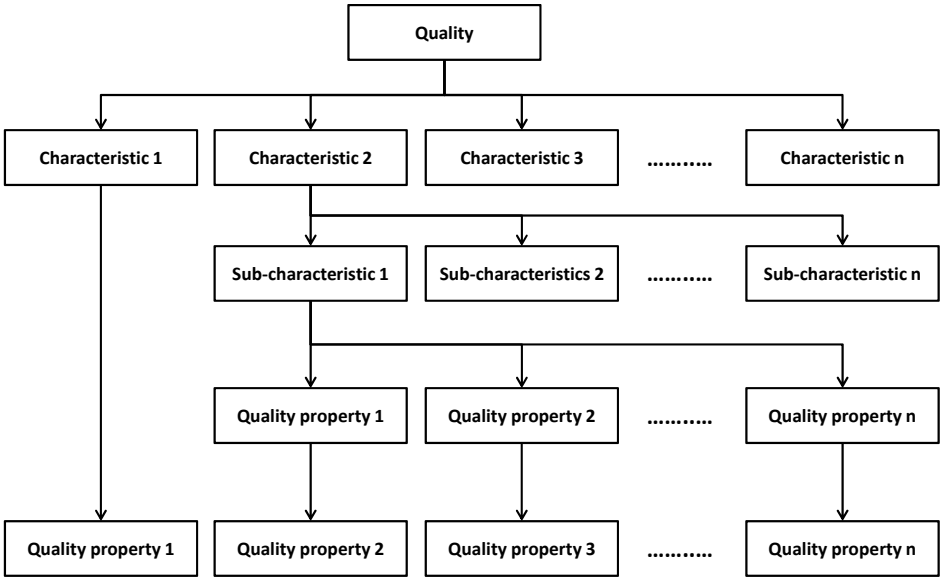

Figure 1. Hierarchical structure of the quality model from ISO/IEC 25010

needs of the various stakeholders. The leaves of the tree structure are the measurable quality properties, as depicted in Figure 1. Quality properties are measured by applying a measurement method, i.e. a logical sequence of operations used to quantify the properties with respect to a specific scale: the result of applying a measurement method is called a quality measure element. Quality measure elements can be composed in a measurement function in order to obtain a quality measure, which in turn quantifies the quality characteristics and sub-characteristics. This quality measurement model is represented in Figure 2 .

The focus of this work is in the software quality product model, which includes eight characteristics: Functional suitability, Performance efficiency, Compatibility, Usability, Reliability, Security, Maintainability, and Portability. Each of the characteristics is further decomposed in sub-characteristics. Figure 3 represents the whole structure.

Software quality assurance is a critical activity [6]. It is possible to adopt several techniques to improve quality: testing, code inspections and formal verification are the

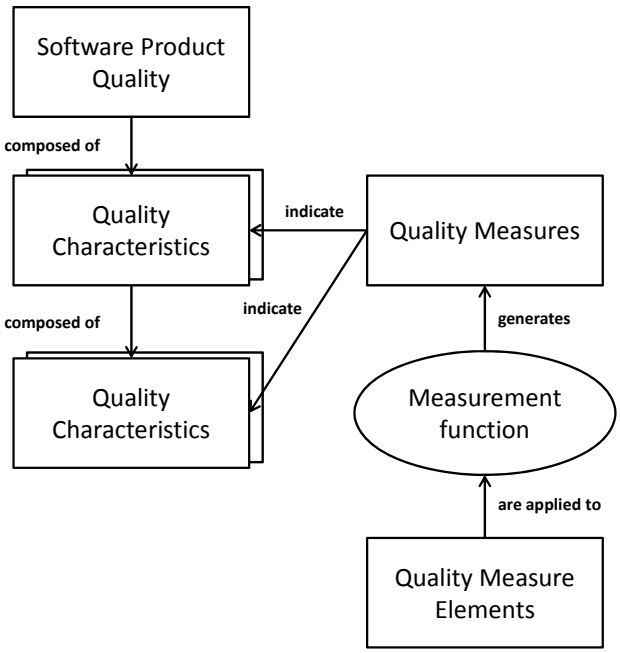

Figure 2. Software product quality measurement model from ISO/IEC 25010

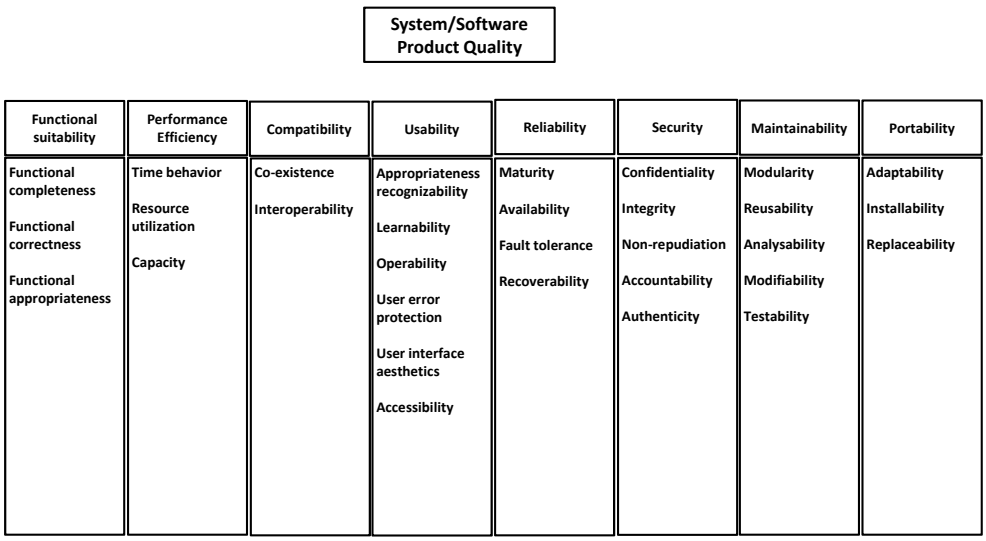

Figure 3. Software product quality model from ISO/IEC 25010

most widely used in industry. Although effectiveness and importance of these activities and methodologies is historically proved, all these techniques share a common limitation if compared to static analysis, i.e. the applicability during software development. In fact these techniques can be applied only after the system or parts of it are built and working. This necessity introduces a delay, which is quickly translated into money lost when modifications to the code are necessary to fix a defect or improve certain aspects. On the contrary, static analysis tools instead promise to speed up the software quality verification process because they can be applied during development.

However, ASA tools need to be precise and effective, i.e. to signal real problems (and not false positive) and to address the most urgent quality concerns of stakeholders. These two properties cannot be achieved without a proper customization of the tools or a triage of the produced warnings with respect to the contexts and the projects they are going to be applied.

Here comes the underlying idea of this PhD work: customize ASA tools with respect to different quality dimensions of interest.

1.3. APPROACH

We assess the impact of using automatic static analysis on software quality through the empirical approach.

According to Wohlin et al [7], this approach is fundamental to conduct the software process quality improvement in the most objective and safe way, both in terms of process assessment and evaluation of an improvement proposal. In fact, without a proper methodology it is difficult to measure an improvement. Empirical studies have been traditionally used in social sciences and psychology, and less frequently in technical fields. Problems like the lack of applicability to the Software Engineering (SE) domain, the cost of experimentation or the excessive level of noise in data collected, have been largely discussed and finally rebutted in the literature (e.g, [8], [9]) and nowadays the need of a more scientific approach and theory building in SE is impelling from several SE research communities (e.g., [10], [11], [12] and [13]). Moreover, software production is a human-intensive process and experimentation provides "a systematic, disciplined, quantifiable and controlled way" [7] to evaluate the software production activities and their quality.

The Empirical Software Engineering (EMSE) research can be applied using two paradigms: qualitative and quantitative research. Qualitative research studies objects in their natural settings, and phenomena are explained by eliciting explanations from people. This approach is for construction of perspective-based results and reflects the question "*Why…?*" . Quantitative research is focused on the quantification of a relationship or a comparison of two or more groups. This type of research answers the question "*How…?*".

The two approaches are complementary and are often mixed or alternated in a research. Both approaches provide a set of *methodological tools* defined by Shull et al. [14] as "a set of organizing principles around which empirical data is collected and analyzed." These tools are called "empirical strategies" by Wohlin et al. [7], "kinds of empirical studies" by Juristo et al. [15] and "research methods" by Shull et al. [14]. The first two authors identify three methodological tools (experiments, surveys, case studies), the latter

one adds two more tools (ethnographies and action research). Below we present the tools as defined by Shull et al [14].

- *Survey*: it is used to identify the characteristics of a broad population of individuals. It is conducted through the use of questionnaires, structured interviews, or data logging techniques. One important step in survey researches is the identification of a representative subset of the population, since usually it is not possible to poll every member.
- *Case study*: it investigates a phenomenon within its real-life context, offering in-depth understanding of how and why certain phenomena occur, thus investigating cause–effect relationships (qualitative research). *Exploratory case studies* are used as initial investigations of some phenomena to derive new hypotheses and build theories, while *confirmatory case studies* are used to test existing theories.
- *Controlled experiment*: it is an investigation of a testable hypothesis where one or more independent variables are manipulated to measure their effect on one or more dependent variables. Each combination of values of the independent variables is a treatment. Not always true experiments are possible in SE (e.g., full randomization is often not achievable in real contexts): in that case quasi-experiments can be conducted (for instance, the subjects are not assigned randomly to the treatments).
- *Ethnography*: this method is focused on the sociological aspects. Ethnographies study a community of people to understand how the members of that community make sense of their social interactions. For software engineering, ethnography can help to understand how technical communities build a culture of practices and communication strategies that enables them to perform technical work collaboratively. A special form of ethnography is *participant observation*, where the researcher becomes a member of the community being studied for a period of time.

- *Action Research*: in Action Research, the researchers attempt to solve a real-world problem while simultaneously studying the experience of solving the problem, intervening inside the real contexts to improve the situation.

The presented methodological tools can be mixed to form a more complex research strategy where the weaknesses of one method can be compensated by the strengths of other methods: this is the strategy that we followed to empirically evaluate the impact of ASA issues on the quality characteristics of the ISO-IEC 25010 (former 9126) standard quality model. The methodological tools used are experiments and case studies: for each quality characteristic analyzed, we selected the more appropriate methodological tool with respect to the research questions and the context of the assessment. Thus, given a software quality characteristic and a quality measure related to it, we empirically assessed whether applying ASA has a relevant impact on the quality measure and, as a consequence, on the corresponding quality characteristic under study. In practice, this process is applied to the software quality measurement model of the ISO/IEC 25010, as depicted in Figure 4.

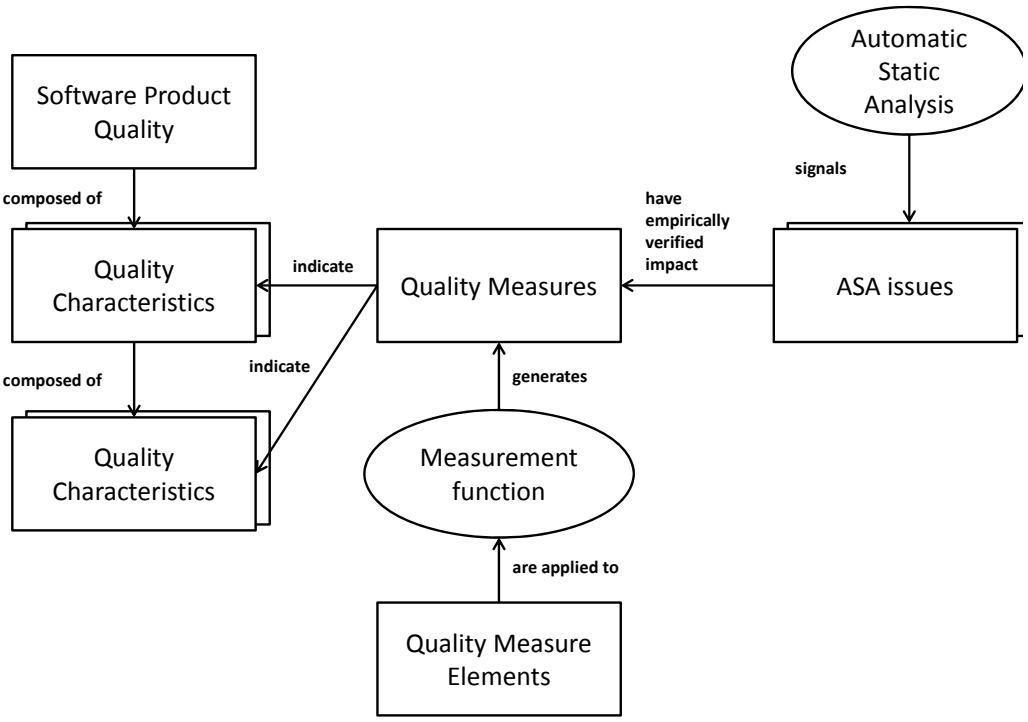

Figure 4. Empirical assessment of the impact of ASA on software quality

1.4. STRUCTURE OF THE THESIS

The structure of the thesis is quality characteristic-oriented. Chapters 2, 3 and 4 are related to a different software quality characteristic analyzed: Chapter 2 is about Functional suitability and Reliability, Chapter 3 focuses on Maintainability in terms of Technical Debt, Chapter 4 on Performance Efficiency. Chapters 2, 3 and 4 share the same logical structure: their first section provides the reader with definitions and terminology, the following sections contain the empirical studies conducted, and the last section draws the conclusion. Chapter 2 adds specific related work sections before (2.2) and after (2.5) the empirical study in order to compare our results with results of the state of the art. This was not possible with Maintainability and Performance Efficiency because we did not find similar approaches to compare with. However, each empirical study contains its own background/related work sub section.

Chapter 5 prepares the path for future directions of the research on ASA and software quality: we present an experiment on the impact of ASA on Energy Efficiency and a case study on the problem of ASA for multi-language software projects.

Finally, Chapter 6 summarizes the work done and the most important contribution to the state of the art.

2 FUNCTIONAL SUITABILITY AND RELIABILITY

2.1. DEFINITIONS

Functional suitability is defined in ISO-IEC 25010 as “the degree to which a product provides functions that meet stated and implied needs when used under specified conditions”. It is based on three sub-characteristics:

- Functional completeness: “degree to which the set of functions covers all the specified tasks and user objectives”
- Functional correctness: “degree to which a product or system provides the correct results with the needed degree of precision”
- Functional appropriateness: “degree to which the functions facilitate the accomplishment of specified tasks and objectives”

Reliability is defined in ISO-IEC 25010 as “the degree to which a system, product or component performs specified functions under specified conditions for a specific period of time”. It is based on the following sub-characteristics:

- Maturity: “degree to which a system, product or component meets needs for reliability under normal operation”
- Availability: “degree to which a system, product or component degree to which is operational and accessible when required for use
- Fault tolerance: “operates as intended despite the presence of hardware or software faults”
- Recoverability: “the degree to which, in the event of an interruption or a failure, a product or system can recover the data affected and re-establish the desired state of the system”

Both functional suitability and reliability are affected when defects in the software cause faults or failures, and unexpected behaviors of the software occur.

2.2. AUTOMATIC STATIC ANALYSIS AND DEFECTS

One of the most promising advantages of using Automatic Static Analysis is the possibility to find defects in the source code during development, without waiting test or inspection (assuming the traditional software lifecycle). In fact ASA tool evaluate the software in the abstract, without running it or considering a specific input. Yet another advantage in terms of time is that ASA tools do not need a working code base, contrary to the other usual VV activities like testing and code inspections that have hence a consistent delay injection. Given that the longer the delay of a fault insert-remove is, the higher the cost of removing that defect is [16], the introduction of ASA tools in software production could lead to important benefits. However these benefits might be hindered by the rate of false positive issues, i.e. issues that are not related to any defect. Despite the fact that the definition of defect is dependent on the context and requirements of the software (e.g., a function return with a delay of 0.5 second may be acceptable in a web application, but not in real time systems), the problem has been widely reported in the literature. For this reason, we started the empirical validation of ASA issues from defects, hence from functional suitability and reliability.

The effort of the research community has focused on evaluating the capability of using ASA tools to find defects in two main streams: I) looking at single ASA issues to identify defects in single lines of code or II) looking at large sets of issues as early indicators of the more defect-prone modules (e.g. classes, files, software components).

We now introduce the main results found in each of the two research streams. Our contributions will follow, again using the logical framework of the two research streams.

FIRST RESEARCH STREAM: LOOKING AT SINGLE ASA ISSUES TO FIND DEFECTS

Several studies in the literature have reported on the percentage of false positive ASA issues of different tools and in different contexts. For instance, Wagner et al. [17] analyzed and classified with experienced developers issues from three ASA tools

(FindBugs, QJPro and PMD) on four industrial projects and one university project, and they reported that the percentage of false positive was 47% for FindBugs, 31% for PMD and 96% for QJ Pro.

Weydan et al. [18] reported that more than 96% of FindBugs and IntelliJ issues did not relate to any fault or refactoring in two open source systems (jEdit and iText).

Lower percentages of false positives are reported by Ayewah et al. [19] running FindBugs on the JDK 1.6.0-b105; the authors report that almost 50% of medium/high priority issues related to correctness had impact on the functionality, and 10% had a serious impact. On the flip side, 160 issues out of 379 were trivial (i.e., no impact), while 5 issues were due to faulty analysis of FindBugs. A similar experiment with the same category of issues was performed at Google, with similar percentages of false positive issues, and a further validation conducted on Glassfish v2 showed an even better result: 50 defects out of 58 disappeared due to changes made to specifically address the issues raised by FindBugs.

Switching to the C languages, Boogerd and Moonen [20] [21] analyzed four industrial projects in C and C++ with an ASA tool for the MISRA standard [22] , and they discovered that a small set of rule violations (12 out of 72 in [20], and 10 out of 88 in [21]) were related to defects in source code. Finally, Nagappan and Ball [23] reported that only 12.5% of defects fixed in Windows Server 2003 pre-release were found with two ASA tools (PREfix and PREfast). Precision was not reported in this study, so this figure is not directly comparable to the previously reported results.

Overall, except for one study [19], we conclude that the precision of the ASA tools is rather low, because high ratios of false positives (i.e. low precision) were reported in many studies.

SECOND RESEARCH STREAM: USING ASA ISSUES TO PREDICT MODULES WITH MORE DEFECTS

The second approach is to investigate whether static analysis issues can be used as early predictors for the most defect prone modules in software systems, rather than identify the single issues that point to specific defects.

Nagappan and Ball [23] discovered positive correlations (0.37 and 0.58) between issue densities from two ASA tools, PREfix and PREfast, and the pre-release defect density. Moreover, they successfully used the ASA issue densities to discriminate between components of high and low quality.

A similar approach was used by the same author in a study carried out at Nortel Networks [24], where automatic inspection defects found by ASA tools had a positive correlation with failures (0.40 and 0.49). Moreover, together with code churn, ASA issues were good discriminators in identifying fault-prone modules. The study at Nortel continued and one year later Zheng et al. [25] reported even higher correlations between the number of ASA issues in files and three different indicators of external quality, i.e. number of tests failures, number of customer reported failures and number of total failures (respectively 0.71, 0.60 and 0.73).

Other authors used a similar approach. For instance, Plosch et al. [26] studied the correlation between the number of FindBugs and PMD issues, and defects in Eclipse SDK 2.0, 2.1 and 3.0. They found positive correlations for both tools (0.34, 0.25 and 0.30 for PMD, and 0.20, 0.08, 0.20 for FindBugs). Excluding the LOC related metrics, PMD issues correlated better with defects than other static metrics (e.g., number of methods, number of fields, etc.)

Finally, Marchenko and Abrahamsson [27] used two tools, namely CodeScanner and PC-LINT, to analyze five projects in the Symbian C++ environment. They computed the correlation between issues and critical defects in two snapshots of the project (i.e. within 90 days after the release and within 180 days after the release) and they observed contradictory results: CodeScanner obtained very high positive correlations (0.70 and 0.90), while PC-LINT issues strongly correlated negatively (-0.90 and -0.70) with defects.

Overall, all the current results available in the literature but one [27] show that using ASA issues to find the most defect-prone files or modules is more effective than using individual ASA issues to discover individual defects.

CONTRIBUTION TO THE STATE OF THE ART

We performed two experiments in the first research stream and a case study in the second research stream. In the first stream, i.e. looking at single ASA issues to identify defects, we analyzed two pools of small Java programs developed by students. Results were compared later to the same analysis applied to an open source tool.

Regarding the second research stream, we conducted a case study with an industrial application.

2.3. ASSESSING THE PRECISION OF FINDBUGS BY MINING JAVA PROJECTS DEVELOPED AT A UNIVERSITY: FIRST CASE STUDY

In a first experiment we studied the precision of issues revealed by bug findings tools. Our goal is to answer the following research question: which issues are actual predictors of bugs, and which are not? This knowledge is very important to provide the developers with accurate information that can be used effectively in developing and maintaining the software.

We conducted an empirical validation of the issues of a widely used tool: FindBugs v1.3.8 [28]. In particular we analyzed the issues produced by FindBugs on a large pool of similar programs. The main contributions of the work are:

- It provides empirical evidence about the validity of issues categories as bug predictors;
- As a consequence identifies a first step to make bug-finding tool usage more effective;
- Using a large pool of developers, it eliminates the effect of developer style on the results

CONTEXT

The program pool was developed in the context of the Object Oriented Programming (OOP) course at the authors' university, where students develop Java programs for the exam.

The program pool was developed in the context of the Object Oriented Programming (OOP) course at the authors' university, where students develop Java programs for the exam. The exam procedure is carried out on six steps.

- Teachers define the project and provide the students with a textual description and a set of wrapper classes. The students develop a first version of the program in the laboratory (the “lab” version) and submit it to a central server by means of an Eclipse Plugin.
- A tool on the server, PoliGrader [29], manages the delivery process and runs a suite of black box acceptance tests (JUnit classes). Acceptance tests are written by teachers of the course to check all functionalities required and the highest possible code coverage is obtained running tests on a correct solution program.
- Results of test execution and test source code are sent back to the students.
- Students improve the lab version at home, creating a new version of the program (the “home” version), that must pass all acceptance tests. This new version is submitted back to the server.
- The PoliGrader tool checks that home versions pass all tests and compute marks taking in considerations the numbers of tests passed in the lab version and the diff between lab and home version.
- All information (marks, source code, tests, and changes) is available to teachers in order to finally evaluate the students.

The code base used in the experiment consists of 85 Java assignments from the 2009 OOP course: requirements are the same for all the assignments; and they are publicy available at the following URL: http://softeng.polito.it/vetro/confs/msr2010/Requirements.htm. Each assignment contains both lab and home versions syntactically correct, and home version passes 100% of the acceptance tests. Acceptance tests are written by teachers of the course in such a way all functionalities are checked. Teachers develop also a

correct “solution program”, and they check test coverage on it. The average size of projects is 166.4 NCSS (Non Commenting Source Statements) for lab versions and 183.81 NCSS for home versions. The estimated number of function points for the project is 66.30.

An issue produced by FindBugs is characterized by an ID, a textual explanation, and a location in the source code. The issues are categorized by FindBugs according to two dimensions: category (Bad Practice, Correctness, Style, Performance, and Malicious Code are the categories with at least one issue signaled in our code base) and priority (Low, Medium, High). Both classifications have been decided by the tool's authors and are based on their personal experience.

EXPERIMENT DESIGN

To address the research question we consider a main dependent measure: precision of the issues that can be defined as the proportion of the signaled issues that correspond to actual defects.

Precision is a derived measure that can be computed on the basis of the following primitive measures: NI, the number of issues signaled by FindBugs and NA, the number of issues corresponding to actual defects. We do not compute recall (commonly coupled with precision), because it would require the knowledge of the complete set of defects. This can be computed only by hand: given the large number of projects to be checked this is a long and error prone process.

To determine NA we adopted the concepts of temporal and spatial coincidence, previously presented in literature in [20] [21] [30].

We have temporal coincidence when one or more issues disappear in the evolution from the lab to the home version, and in the same time one or more defects are fixed: probably those issues were related to the fixed defects. In this context defects fixed are revealed when a test that in lab version fails instead in home version succeeds. Figure 5 and Figure 6 show a real example of temporal coincidence, extracted from the programs

examined with FindBugs in the experiment. We observe in that an issue (self-assignment of a field) is signaled on line 9: the field forum is assigned to itself. In the evolution from lab to home version (Figure 6) the student discovers the error and adds a parameter to the constructor's method, in such a way it is assigned to the field forum. The issue effectively disappears in the home version. However, the real cause of the fault isn't on line 9, but on the list of parameters on lines 1-2-3: in fact the student modified only line 3 (underlined in Figure 6). Hence, there is a possibility that a disappearing issue is not related to the disappearing defect: this is the noise of temporal coincidence metric that can be filtered out by adding the spatial coincidence. We observe spatial coincidence when an issue's location corresponds to lines in the source code that have been modified in the evolution from the lab to the home versions. Figure 7 and Figure 8 show an example of temporal + spatial coincidence. In the lab version (Figure 7), an issue is signaled on line 6: it is an infinite recourse loop, because the function calls itself without any stopping criterion. In the new version (Figure 8), the student detects the error and fixes it changing line 6 (underlined): in the home version the issue is no longer signaled and it was located in the same line changed during the fix, therefore we observe temporal + spatial coincidence. In practice the combination of temporal and spatial coincidence is interpreted as a change intended to remove the issue, which is linked to a defect. After the computation of precision with temporal + spatial coincidence method, we establish 2 precision thresholds and we perform a statistical test against null hypotheses to determine whether an issue is a good or bad defect predictor.

The procedure followed to conduct the study is very simple: we ran the FindBugs tool on both versions of each assignment in the repository, then we collected the information about the change performed to evolve the lab version into the home version. The changes were identified using the DiffJ tool, which operates on two versions of a Java program and is able to compute for each pair of corresponding Java classes which lines changed.

Afterwards, we computed precision of issues, first without considering categories and priorities, then analyzing results observing each issue group (combination of category and priority) separately.

To determine whether an issue group is a good or bad defect predictor, we established 2 precision thresholds and we performed statistical test against null hypotheses. Thresholds were established after observing the distribution of issues precision for each assignment (

	Bad Pr.	**Corr.**	**Mal.C.**	**Perf.**	**Style**
Low	5 / 70	1 / 3	0 / 0	0 / 7	5 / 11
Medium	2 / 145	12 / 45	4 / 15	31 / 144	6 / 16
High	13 / 28	12 / 19	0 / 0	0 / 0	3 / 5

Table 2 and Figure 9), without distinction of categories and priorities.

The mean of precisions is quite low (0.15) and the variability is high. We decided to consider the issue group (group G in the following) as a defect predictor if it has a precision greater than 30%. Such a low value is justified by the exploratory nature of this work and it compensates for the large variability we expect to find in each group. Furthermore this value is far enough from the average precisions of the issues: in 50% of assignments precision is 0; in 75% (3rd quartile) of the assignments precision is at most 0.25, less than the threshold; finally, the 30% precision threshold is the double of the mean of precisions, that is a quite wide ratio.

To identify the issue groups that can be considered as defect predictors, we define the first null hypothesis:

HA$_0$: precision of the issues belonging to group G is less than 30%.

The next step is to find false positives, the bad defects predictors. We consider as false positives the ones with precision <5%, a very low threshold. So we formulate the following parametric null hypothesis:

HB_0: precision of the issues belonging to group G is greater than 5%.

Read together, the two hypotheses mean that a group of issues G is a good predictor (GP) if precision of the issues that it contains is >30% and is a bad predictor (BP) (i.e. a generator of false positives) if precision of the issues that it contains is <5%. The goal of the data analysis is to reject the above null hypothesis by means of statistical tests. For this purpose we selected the single-tailed proportion test with binomial distribution [31]. Given a sample proportion and sample size, such a test computes the probability that the general population (from which the sample is extracted) has a proportion greater (or lower) than a reference proportion. To reject the null hypothesis we adopt the standard significance level at 5%, that is the probability of rejecting a null hypothesis when it is true (type I error) we consider acceptable.

RESULTS

Overall FindBugs revealed a total of 508 issues (NI) in the 85 lab versions of the assignments, among them 94 (NA) were removed in changed lines (temporal and spatial coincidence). Table II shows NA / NI at issue group level. Table III contains precisions and hypothesis tests computed for each different issue group (p-values are shown below precision). Columns of Table II and Table III contain abbreviations of the full names of categories, that are: Bad Practice, Correctness, Malicious Code, Performance, Style.

The full tables with number of detections (NI) and number of issues removed in changed lines (NA) for each project and each issue group are available at the following URL: http://softeng.polito.it/vetro/confs/msr2010/ .

HA: The null hypothesis is rejected only for categories Bad Practice and Correctness both at High priority: this is the set of true positives for spatial + temporal coincidence. All the other groups have non significant p-values and exhibit low estimate precisions except for Style at High priority which has a relatively high precision, though not significant.

HB: Bad Practice and Performance at Low priority, and Bad Practice Medium priority, are the groups whose precision is lower than 5%: however, only Bad Practice at Medium priority has a significant p-value, and we can reject HB0 for this group.

The results from the hypothesis testing presented above let us identify the sets of good and bad defect predictor issue groups.

Table 3. Precision: Spatial+Temporal coincidence

	Bad Pr.	Corr.	Mal.C.	Perf.	Style
Low	7%	33%	NA	0%	45%
HA	1	0.50	NA	0.91	0.21
HB	0.71	0.82	NA	0.50	1
Medium	**1%**	27%	27%	22%	38%
HA	1.00	0.63	0.50	0.98	0.35
HB	**0.04**	1	1	1	1
High	46%	63%	NA	NA	60%
HA	**0.05**	**<0.01**	NA	NA	0.16
HB	1	1	NA	NA	1

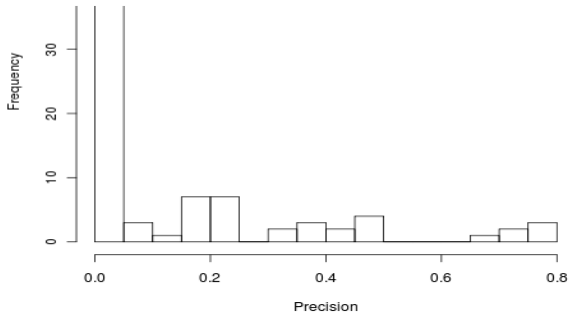

Figure 9. Histogram of precisions

DISCUSSION

On the basis of these results, we built a partial ordering of the issue groups dividing them into three sets: good, bad and ambiguous. We devised the ordering by putting in the set of good issues the issues marked as defect predictors, in the set of bad issues those issues marked as false positives, and in the set of ambiguous issues all the others that

haven’t been classified . The set of good predictor issues is GP={Bad Practice High, Correctness High}, the set of bad predictors is BP={Bad Practice Medium}, and the remaining issue groups are ambiguous. Counting the single issues belonging to those groups, they are just 8 out of 359 (2.23 %).

The rationale of this ranking is a new prioritization of warnings based on groups, that takes into account the probability of signaling a defect. An important practical application of this finding is a filtering strategy that can avoid to developers the information overload constituted by a very large number of issues: in our datasets bad predictor issues are the 28.5 % of the total detections in lab versions. Fixing issues with a low probability of being related to a defect is dangerous since we know from Adam’s law [32] that the probability of introducing a new error during a fault correction is always different from zero.

THREATS TO VALIDITY

We can identify 2 threats: an external and a construct threat. The external threat is: we have studied small student projects, hence the application of findings in industrial context is debatable. Construct threats is concerning the identification of defects. In this study, no bug database was available: we made the assumption that all changes were done to fix a defect: actually, it is possible that some changes were not related to real defects, but to other motivations (cleaner code, more readable code, and so on). Nevertheless, we don't expect that this kind of noise could change results and ranking, because usually students correct the lab versions in a quick and dirty way, doing as few changes as possible, for two reasons: 1) the home version is the last version of the project, actually no maintenance has to be done subsequently; 2) students are discouraged in doing many changes, because the mark suggested by PoliGrader decreases with the quantity of changes made (see details in [29]).

CONCLUSIONS

The analysis of precisions demonstrated that only 2 out of 15 groups of issues can be considered as reliable predictors of actual defects, and one group of issues has a precision that is practically negligible. These findings and the adoption of the technique used may have a practical impact in filtering issue notifications for developers to reduce information overload.

The experiment presented in the next section expands the current one in the following way: we apply the temporal and spatial analysis with higher level of detail, specifying the single issues, besides categories and priorities, and we enlarge the repository of projects.

2.4. ASSESSING THE PRECISION OF FINDBUGS BY MINING JAVA PROJECTS DEVELOPED AT A UNIVERSITY: SECOND CASE STUDY

In the previous work [33], described in Section 2.3, we analyzed the issues produced by FindBugs v1.3.8 on a pool of 85 similar small programs, each of them developed by a different student in our university. The goal of our experiment was to verify which FindBugs issues were related to real defects on source code and which not.

In the work we present hereby we reproduce the same experiment, with the following improvements:

- we enlarge the code base (301 projects)
- we consider the single FindBugs issues instead of considering only the categories
- we use functional tests failures to validate the relationship FindBugs between issues and defects in the code.

The knowledge of the issues related to defects is very important to provide the developers with accurate information that can be used effectively in developing and maintaining the software.

EXPERIMENT DESIGN

Adhering to the Goal-Question-Metric approach [34] we first define the goal of the research at conceptual level, which is formally presented in Table 4. The goal aims at identifying the issues revealed by FindBugs and their relationship with the defects. Corresponding to the goal we formulate the research question (RQ1) and identified the relative metric (M1).

Table 4. Goal of the experiment

	Goal
Purpose	Identify and characterize
Issue	issues linked to real defects and generated
Object (Process)	by FindBugs 1.3.8 analysis on 301 University Java Projects
Viewpoint	from the view point of a student Java programmer

- RQ1: Which FindBugs issues are related to defects (good defect predictors) and which not (bad defect predictors)?
- M1: Issue precision (spatial + temporal coincidence)

To address research question RQ1 we consider the same main dependent measure used in the former experiment: the precision of the issues (M1) that can be defined as the proportion of the signaled issues that correspond to actual defects.

We establish 2 precision thresholds and we perform a statistical test against null hypotheses to determine whether an issue is a good or bad defect predictor.

The 2 thresholds are:

- a minimum precision threshold that issue must exceed to be considered as good defect predictor,
- a maximum precision threshold that issues must not exceed to be eligible to the role of bad defects predictors.

Given the exploratory nature of this work, we decide to consider an issue as a good defect predictor if it has a precision greater or equal to 50%. Such threshold is also a compromise between the different true positive ratios of FindBugs issues found in literature, and it is higher than the threshold used in [33] because we want to achieve stronger results. Therefore, we can formulate the first null hypothesis as follows:

HA$_0$: the precision of issue I is not greater than 50%.

The next step is to find false positives, i.e. bad defects predictors. We consider as bad defects predictors those issues with precision <=5%, a very low threshold, that we consider a strict inclusion criterion. So we formulate the following null hypothesis:

HB_0: the precision of issues I is not lower than 5%.

Read together, the two hypotheses mean that an issue I is a good predictor (GP) if hypothesis HA0 can be rejected, i.e. its precision is >=50%, conversely it is a bad predictor (BP) (or source of false positives) if hypothesis HB0 can be rejected, i.e. its precision is <=5%. The goal of the data analysis is to reject the above null hypothesis by means of statistical tests. Since data is not normally distributed, for these tests we select the Mann-Whitney test [35] that estimates the median. To reject the null hypotheses we adopt the standard significance level at 5%, that is the probability of rejecting a null hypothesis when it is true (type I error).

Furthermore, to increase results reliability, we perform a sensitivity analysis and a validation of results. The sensitivity analysis is carried out by computing threshold ranges in which the composition of good/bad predictor sets remains the same: in this way we understand the impact of the thresholds choice on results, and we also exanimate border values. The validation is based on the idea that the good predictors effectively identify real bugs in the programs, therefore affecting their external quality, whereas the bad predictors are not related to defects and do not have impact on external quality. Hence quality of projects that contain good predictor issues detections should be lower than the mean quality of all the other projects, whilst quality of projects that contain bad predictor issues detections should be not different from the mean quality of the remaining projects. The proxy for projects' external quality is the percentage of passed tests in lab versions, positively correlated to the quality. Therefore we carry out the validation by comparing the proportion of acceptance tests passed by projects containing at least one occurrence of the issues in the set to be validated vs. the same proportion in the remaining programs.

DATA COLLECTION

An issue produced by FindBugs is characterized by an ID, a textual explanation, and a location in the source code. The issues are grouped by FindBugs in category (Bad Practice, Correctness, Style, Performance, and Malicious Code have at least one occurrence in the code base) and priority (Low, Medium, and High): hence the single issue is uniquely identified by the combination of ID, category, and priority. We store also their locations in the source code (file name, class, method, line number) and in the project (course ID, student ID, lab/home version). Afterwards, we use the DiffJ tool to collect the changes done to evolve the lab version into the home version: DiffJ operates on two versions of a Java program and is able to compute for each pair of corresponding Java classes which lines changed. Finally, results of functional tests are obtained through the PoliGrader tool.

The data collection process is represented in Figure 10.

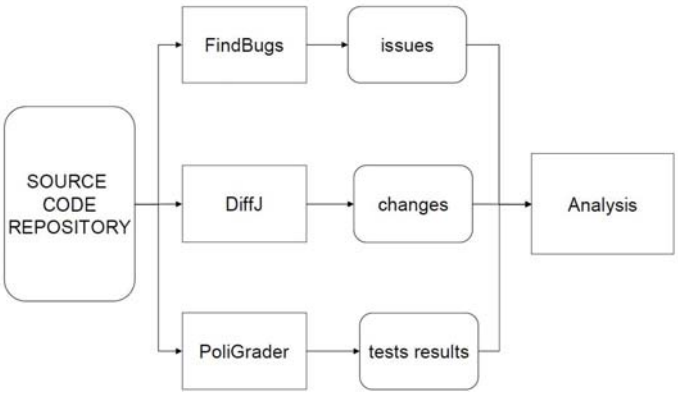

Figure 10. Data Collection Process

THREATS TO VALIDITY

We can identify three main threats: two external and one construct threat. The first external threat is: we study small student projects, hence the application of findings in industrial context is debatable. However, this weakness is balanced by the fact that this study eliminates the effect of developer style on the results, because a large pool of developers is used for the same projects. In addition, we recall the study of P. Runeson [36], whose conclusions could neither reject nor accept the hypothesis on differences between freshmen, graduate students and industry people. We also draw in section 5.2 similar conclusions. The construct threat is concerning the identification of defects. We do not have a bug database but only tests failures: we make the assumption that all changes are done to fix a defect. It could be possible that some changes are not related to real defects, but to other motivations (cleaner code, more readable code, and so on).

RESULTS

The automatic application of FindBugs on all the 301 projects (both versions, lab and home) produced a large collection of detections: 1692 in lab versions, belonging to 77 issues, whilst home versions detections are 1662, belonging to 73 issues (this does not mean that 30 issues were removed across all projects, since the number of issues in home version is given by: issues in lab version – issues fixed + new issues introduced). We answer to RQ1 computing Metric M1, that is the precision of the issues, with respect to temporal + spatial coincidence. Table 2 indicates minimum, maximum, 1st and 3rd quartile, median and mean of precisions (NA/NI) in projects.

The mean of precisions in projects is low (0.126) and the variability is high (standard deviation is 0.22, almost the double of the mean). More than 2/3 of projects have a precision lower than the selected minimum threshold 0.50 (only 6 projects out of 301 have a higher precision), and in half of the projects precision is about 1/5 of this threshold.

These observations show that the threshold selected is very strict, despite of the initial considerations. and show the issues for which we could reject either of the two null hypotheses. We do not provide the precision of issues for which we can not reject either of the two null hypotheses because of their large number (77), however the full list is available on line . The columns in the tables show: the issues ID, the average precision (sum of NA/sum of NI), the estimated median of precision, and finally the p-value of the Mann-Whitney single-tailed test.

The set of good defects predictors is composed of 4 elements: 3 out of 4 have an estimated median precision of 1, the double that of the threshold. The median of the last issue, UUF_UNUSED_FIELD (Performance, 2), is exactly the threshold value: this is a border value and it will be examined in depth in Section 5. The 4 issues are:

- GC_UNRELATED_TYPES: a call to a generic collection method that contains an argument with an incompatible class from the collection's parameter.
- SA_FIELD_SELF_ASSIGNMENT: a self-assignment of a field, like int y = y.
- UR_UNINIT_READ: the constructor reads a field which has not yet been assigned a value.
- UUF_UNUSED_FIELD: a field is never used.

In contrast there are many more issues among the defect predictors set i.e. 16. All of them have median = 0. Since they are many, for their descriptions please refer to FindBugs website .

We perform a sensitivity analysis of results to check their stability with respect to the inclusion criteria: we compute the threshold ranges in which the composition of groups remains the same. The good predictors set is stable in the range 0.21–0.50. For threshold values greater than 0.5 the issues GC_UNRELATED_TYPES (Correctness,1) and UUF_UNUSED_FIELD (Performance,2) are excluded, and above 0.51 the set becomes empty. Analyzing instead lower bound, a new issue could be included in the set of good predictors only putting a very low threshold: at 0.20 issue NP_UNWRITTEN_FIELD (Correctness,2) could enter the group, and 2 more issues can enter with even lower

thresholds : 0.12 and 0.11. Since the upper bound is already very strict and lower bound must be relaxed from 0.50 to 0.20 to change the set, we can affirm that results about good predictors are reliable.

The sensitivity analysis of bad predictors have the following result: the set is stable in the threshold range 0 – 0.15, so again a wide range. In fact we should use a high threshold, 0.16 (3 times bigger than the 5% of the original one) to change the set and include a new issue, NM_FIELD_NAMING_CONVENTION(BadPractice,3). A further issue, REC_CATCH_EXCEPTION (Style,3), enters only with threshold = 0.25. We conclude that also bad predictors set is robust.

Table 5. Distribution of issues precisions

Min	1st q	Median	Mean	3rd q	Max
0	0	0	0.126	0.20	1

Table 6. Precision of good defect predictor issues

Issue ID	NA/NI	Prec. Est.	p-val
GC_UNRELATED_TYPES (Correctness,1)	12/15	1	0.048
SA_FIELD_SELF_ASSIGNMENT(Correctness,1)	7/10	1	0.012
UR_UNINIT_READ (Correctness,1)	6/7	1	0.012
UUF_UNUSED_FIELD (Performance,2)	26/55	0.5	0.045

Table 7. Precision of bad defect predictor issues

Issue ID	NA/NI	Prec. Est.	p-val
DM_NUMBER_CTOR (Performance,2)	0/6	0	0.018
DM_STRING_CTOR (Performance,2)	0/29	0	<0.01
DM_STRING_TOSTRING(Performance,3)	0/5	0	0.018
EQ_COMPARETO_USE_OBJECT_EQUALS (Bad_Practice,2)	5/275	0	<0.01
ES_COMPARING_STRINGS_WITH_EQ (Bad_Practice,2)	0/10	0	<0.01
IL_INFINITE_LOOP (Correctness,1)	0/5	0	0.036
NM_CLASS_NAMING_CONVENTION (Bad_Practice,2)	0/17	0	<0.01
NM_CONFUSING (Bad_Practice,3)	0/6	0	0.01
NM_METHOD_NAMING_CONVENTION (Bad_Practice,2)	2/44	0	<0.01
NP_NULL_ON_SOME_PATH (Correctness,2)	0/4	0	0.036
OS_OPEN_STREAM (Bad_Practice,2)	0/71	0	<0.01
OS_OPEN_STREAM_EXCEPTION_PATH (Bad_Practice,3)	0/5	0	0.018
SE_BAD_FIELD (Bad_Practice,3)	0/11	0	<0.01
SE_COMPARATOR_SHOULD_BE_SERIALIZABLE(Bad_Practice,2)	0/49	0	<0.01
SIC_INNER_SHOULD_BE_STATIC_ANON (Performance,3)	0/92	0	<0.01
URF_UNREAD_FIELD (Performance,2)	33/259	0	<0.01

VALIDATION OF GOOD DEFECT PREDICTOR ISSUES

Figure 11 shows the boxplots of passed tests percentages in lab versions: NO_GP is the set of projects that do not contain any detection of a good predictor issue (259 projects), while GP is the set of projects containing at least one good predictor issue detection (the remaining 42 projects). The box plots clearly show that external quality of GP set is lower than external quality of NO_GP set: medians are respectively 63.64% and 40.91% . According to Mann-Whitney tests, we observe significant (p=0.001) differences between the two groups of projects. The 95% confidence interval for the difference between the medians is [6.29,∞]. There is strong statistical evidence that average external quality of projects with at least one good defect predictor issue detected is lower than the average externally quality of all projects. It is very likely that defects in projects with lower quality are correctly identified by the good predictor issues.

We continue the validation and we inspect all the good predictors issues signaled on source code, manually determining whether they were correctly detected by the tool and whether the problem signaled actually caused a wrong behavior of the program (failure of functional test). The detections of issues identified as good defects predictors are 87. We present the results of the manual code inspection in Table 8: for each issue, we indicate the ID, the total number of detections, the number of correct detections and the number of detections that impacted the functionality. Three issues of four have all detections correct and are the cause of an incorrect behavior of the program.

UUF_UNUSED_FIELD (Performance,2) is the exception: we discuss it later.

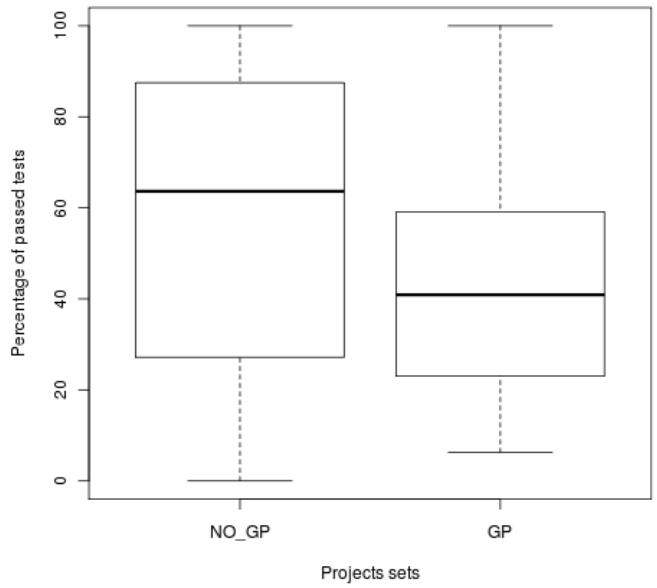

Figure 11. Box plots of passed tests percentages: good defect predictors

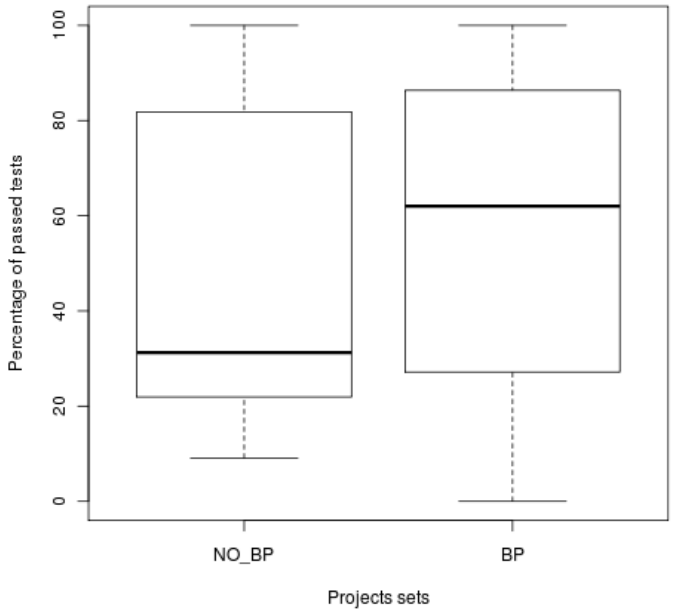

Figure 12. Box plots of passed tests percentages: bad defect predictors

Table 8. Manual inspection of good defects predictors issues

Issue	Nr of dete-ctions	Correct detections	Impact on functiona-lity
GC_UNRELATED_TYPES (Correctness,1)	15	15	15
SA_FIELD_SELF_ASSIGNMENT(Correctness,1)	10	10	10
UR_UNINIT_READ (Correctness,1)	7	7	7
UUF_UNUSED_FIELD (Performance,2)	55	46	0

VALIDATION OF BAD DEFECTS PREDICTOR ISSUES

Figure 12 contains boxplots of passed tests percentages: NO_BP is the set of projects that do not contain any detection of a bad predictor issue (they are just 9), on the right BP is the set of projects with at least one detection of a bad predictor issue (292 projects). We observe that projects BP have higher percentages of passed tests then NO_BP projects. The medians are respectively 62.02% and 31.25%. However, the number of projects in NO_BP is so small that they cannot be a representative sample. In fact, although medians are so different, the null hypothesis that the two medians are equals cannot be rejected with α=0.05 and p-value is 0.1041 (according to Mann-Whitney test). The 95% confidence interval for the difference is $[-\infty,+3.75]$. We can therefore assume that no difference exists among the two sets. We do not perform a manual validation because of the high number of detections related to bad predictors issues (888 in lab versions): we consider the manual check of a representative sample of this bigger population an error prone task.

EXTERNAL VALIDATION: LUCENE PROJECT

We addressed the external validity threats identified in Section 3.2 through a validation of good and bad defect predictor issues on a real project. We selected the open-source system Lucene2, Apache's free information retrieval software library. We made this choice because it provides a very good infrastructure to access its data and because it was peviously studied in the FindBugs literature [37].We selected for the validation the release of the project with the highest number of fixed and closed bugs among those that compile under JDK 5, i.e. version 3.1.0, and the following one 3.2.0. We applied to Lucene the same methodology applied to our student's case study, computing the spatial + temporal coincidence to assess those FindBugs issues that are related to defects.

The Apache Software Foundation has a unified bug tracking system (JIRA3) for all its projects: we downloaded from the Lucene JIRA database (webref3)) the list of fixed and closed bugs that affected version 3.1.0, obtaining 27 defects at the time this paper was written.4 Then we collected from Lucene Subversion repository the relative bug fixes commits and corresponding modified java files, thus obtaining a list of 'buggy files', i.e. the files where the bug had impact. After that, we ran FindBugs on Lucene's source code, analyzing all classes under package org.apache.lucene, in version 3.1.0 and in the following released version, 3.2.0. We enabled all FindBugs issues except the "noisy issues", i.e. those issues randomly inserted for data mining experimentation, and we set the analysis effort to Maximal to be consistent with our experiment on students' projects. We obtained 332 issues in version 3.1.0. Some issues occurrences as CD_CIRCULAR_DEPENDENCY were pointing to more than a class, thus we reported each of them for each class impacted. We also computed through the FindBugs command "computeBugHistory" which issues were deleted in the evolution from version 3.1.0 to 3.2.0: they were 30. Finally, to achieve

[2] http://lucene.apache.org/java/docs/index.html , last access 27/12/2012

[3] http://www.atlassian.com/software/jira/overview , last access 27/12/2012

4 The permanent link to reproduce the search is : https://issues.apache.org/jira/secure/IssueNavigator.jspa?reset=true&jqlQuery=project+%3D+LUCENE+AND+issuetype+%3D+Bug+AND+resolution+%3D+Fixed+AND+affectedVersion+%3D+12314822+AND+status+%3D+Closed+ORDER+BY+priority+DESC

the spatial+temporal coincidence, we computed with DiffJ which lines were modified in the buggy files in their evolution from release 3.1.0 and release 3.2.0, and compared those lines with the location of the issues. We recall that a FindBugs issue occurrence satisfies the temporal+spatial coincidence if it disappeared from a version of the code and the following one and if it was located on lines that changed in a bug fix. For those issues whose location is the whole class instead of specific lines of code, the criterion is to be located in a buggy file. Having only one project analyzed, the precision of a FindBugs issue is defined as the ratio NA/NI, where NI is the total number of detections of an issue and NA is the number of occurrences in bug fixes. This is a minor difference from the students' project analysis.

We adopt again the same two thresholds, considering an issue as a good defect predictor if it has a precision greater or equal to 50%. On the contrary, we consider as bad defects predictors those issues with precision <=5%. Therefore, we can formulate same pair of null hypotheses:

HA_0: the precision of issue I is not greater than 50%.
HB_0: the precision of issues I is not lower than 5%.

We use the Mann-Whitney test to reject the above null hypothesis with the standard significance level 5%.

Table 9 shows the FindBugs issues satisfying the spatial+ temporal coincidence: we found 9 occurrences, that determined an overall precision of 2.71% (9/332). We applied Mann-Whitney test and we obtained that none of the issues rejected HA0 : the set of good predictors is empty. Table 10 reports the issues that are potential bad predictors, together with the result of the test for hypothesis HB_0.

Table 9. Issues satisfying temporal+spatial coincidence

Issue ID	Priority	Category	Classname
CD_CIRCULAR_DEPENDENCY	2	STYLE	org.apache.lucene.index.DocumentsWriter
CD_CIRCULAR_DEPENDENCY	2	STYLE	org.apache.lucene.index.IndexWriter
CD_CIRCULAR_DEPENDENCY	2	STYLE	org.apache.lucene.index.IndexReader
CD_CIRCULAR_DEPENDENCY	2	STYLE	org.apache.lucene.index.DirectoryReader
CD_CIRCULAR_DEPENDENCY	2	STYLE	org.apache.lucene.index.IndexWriter
CD_CIRCULAR_DEPENDENCY	2	STYLE	org.apache.lucene.index.IndexReader
CD_CIRCULAR_DEPENDENCY	2	STYLE	org.apache.lucene.index.IndexWriter
CD_CIRCULAR_DEPENDENCY	2	STYLE	org.apache.lucene.index.SegmentMerger
EQ_COMPARETO_USE_OBJECT_EQUALS	2	BAD_PRACTICE	org.apache.lucene.search.PhraseQuery

The goal of the external validation is to understand if results hold in a very different context. We highlight the differences of the two contexts analyzed reporting the distribution of issues categories in Table 11: Lucene had more Style, Malicious Code and Multi Thread (MT) Correctness issues, whilst the issues on students code are more related to Performance, Bad Practice and Correctness. Also [37] found many race-condition-related issues running FindBugs on Lucene.

The temporal + spatial coincidence criterion produced no Good Predictors issues and no comparisons are possible with the students' projects results. A motivation beyond the difference of contexts is that we probably underestimated the Good Predictor issues, since it is possible that not all defects that affected release 3.1.0 have been fixed in release 3.2.0, therefore some FindBugs issues related to those defects could have been fixed later. However, the overall precision 2.71% is similar to the precision we found in the students' pool of projects (median 0%, average 12.6 %).

On the contrary, the set of Bad Defects Predictor contained 17 issues that were responsible of almost 90% of all detections (292 in 332). We found in students projects 16 issues, and 4 of them are in common with Lucene:

Table 10. Bad defect predictor issues

Issue ID	Category	Precision	NA	NI	P value
CD_CIRCULAR_DEPENDENCY	STYLE	2	8	59	0.0001
CN_IDIOM_NO_SUPER_CALL	BAD_PRACTICE	1	0	6	0.0098
CN_IDIOM_NO_SUPER_CALL	BAD_PRACTICE	2	0	8	0.003
DLS_DEAD_LOCAL_STORE	STYLE	2	0	12	0.0003
EI_EXPOSE_REP	MALICIOUS_CODE	2	0	16	0
EI_EXPOSE_REP2	MALICIOUS_CODE	2	0	17	0
EQ_DOESNT_OVERRIDE_EQUALS	STYLE	2	0	37	0
ES_COMPARING_STRINGS_WITH_EQ	BAD_PRACTICE	2	0	17	0
FL_MATH_USING_FLOAT_PRECISION	CORRECTNESS	2	0	17	0
ICAST_INTEGER_MULTIPLY_CAST_TO_LONG	STYLE	2	0	4	0.0359
ICAST_QUESTIONABLE_UNSIGNED_RIGHT_SHIFT	STYLE	2	0	7	0.0054
IS_INCONSISTENT_SYNC	MT_CORRECTNESS	2	0	16	0
IS2_INCONSISTENT_SYNC	MT_CORRECTNESS	2	0	20	0
MS_PKGPROTECT	MALICIOUS_CODE	2	0	12	0.0003
NM_METHOD_NAMING_CONVENTION	BAD_PRACTICE	2	0	29	0
SE_BAD_FIELD	BAD_PRACTICE	2	0	4	0.0359
URF_UNREAD_FIELD	PERFORMANCE	2	0	16	0

- ES_COMPARING_STRINGS_WITH_EQ (Bad_Practice,2)

  FindBugs Description: “Comparison of String objects using == or !=. This code compares java.lang.String objects for reference equality using

the == or != operators. Unless both strings are either constants in a source file, or have been interned using the String.intern() method, the same string value may be represented by two different String objects. Consider using the equals(Object) method instead."

- NM_METHOD_NAMING_CONVENTION (Bad_Practice,2)
  FindBugs Description: "Method names should start with a lower case letter . Methods should be verbs, in mixed case with the first letter lowercase, with the first letter of each internal word capitalized."

- SE_BAD_FIELD (Bad_Practice,3) –Priority is 2 in Lucene validation
  FindBugs Description: "Non-transient non-serializable instance field in serializable class. This Serializable class defines a non-primitive instance field which is neither transient, Serializable, or java.lang.Object, and does not appear to implement the Externalizable interface or the readObject() and writeObject() methods. Objects of this class will not be deserialized correctly if a non-Serializable object is stored in this field."

- URF_UNREAD_FIELD (Performance,2)
  FindBugs Description: "Unread field. This field is never read. Consider removing it from the class"

Table 11. Issues distribution by category

Category	Occurrences - Lucene	Occurrences - Students	% Occurrences - Lucene	% Occurrences - Students
BAD_PRACTICE	80	701	24.10 %	41.43 %
CORRECTNESS	21	230	6.33 %	13.59 %
MALICIOUS_CODE	49	19	14.76 %	1.12 %
MT_CORRECTNESS	38	0	11.45 %	0.00 %
PERFORMANCE	20	453	6.02 %	26.77 %
STYLE	124	248	37.35 %	14.66 %
INTERNAZIONALIZATION	0	41	0.00 %	2.42 %
TOT	332	1692	100.00 %	100.00 %

Because of the differences in the nature of the projects, our external validation produced two categories of Bad Predictors issues: Generally Bad Defect Predictors, containing the 4 issues in the intersection, and Context-Specific Bad Defect Predictors, containing the remaining 10 issues. We share the dataset online to enable repetitions and validations of the study[5].

DISCUSSION ON RESULTS: ANSWER TO RQ1.

On the basis of the temporal + spatial coincidence criterion, issues related to defects are: call to unrelated types, field self-assignment, uninitialized read of field in constructor (Correctness, 1), and unused field (Performance,2).

The manual validation (see Table 8) of detections showed that Correctness detections are all valid and have an impact on functionality. However, the Performance issue is the only one that has no correct detections that cause an incorrect behavior of the

[5] http://softeng.polito.it/vetro/confs/iet2011/data.zip

program. This fact is reasonable because the issue, as the name of the category suggests, is just signaling waste of memory (variable never used), and it is not a real error (because in this kind of little Java projects, performance of the program is neither mission nor safety critical). However, since their detections are about the 63% of all detections in the set, their contribution to the external quality prediction is important. In fact, there is a reason why projects with detections of unused fields have lower quality: their presence in a program means that the student encountered difficulties in the design of the program, because he planned to use more/different variables that in fact were not necessary. In contrast students who developed applications with higher external quality did not have this kind of problem. This is the reason why we decided to leave this issue in the set of good defect predictors issues, despite the category it belongs is Performance. Furthermore, the double internal validations confirmed that all the 4 issues have a clear impact on external code quality and they can be considered as good defects predictors, with a very high confidence that limit the impact of the internal threats to validity. The external validation was not possible for the GDP, therefore we couldn't assess the impact of external threats on this set.

We also identify 16 issues that are bad defects predictors, and the statistical validation confirms that their detection has no correlation with the external quality of the projects. The internal threat (assumption that all changes are done to fix a defect) could affect results on Bad Defects Predictors, because students must make as few changes as possible, otherwise their mark will decrease: for this reason, they just correct errors and do not perform any change related to performance, maintainability or even errors that are in impossible paths. It is possible that in other projects some of these issues could be fixed by developers. Observing the type of issues in the set, we could assert that the majority of them could be related to this fact. However, this threat is controlled by the comparison with related work in Section 2.5.

For instance, 3 issues are naming convention violations, whose importance for code comprehension is well known, and 4 of the 5 issues belonging to the category Performance are memory leaks (useless constructor of String and Number, unread field and field that should be static). The fifth issue of Performance, i.e. useless toString() applied on a String, could indicate that students have not fully understood the nature of the objects in

Table 12. Bad defect predictors

Issue ID	NA/NI	Prec. Est.	p-val
DM_NUMBER_CTOR (Performance,2)	0/6	0	0.018
DM_STRING_CTOR (Performance,2)	0/29	0	<0.01
DM_STRING_TOSTRING(Performance,3)	0/5	0	0.018
EQ_COMPARETO_USE_OBJECT_EQUALS (Bad_Practice,2)	5/275	0	<0.01
ES_COMPARING_STRINGS_WITH_EQ (Bad_Practice,2)	0/10	0	<0.01
NM_CLASS_NAMING_CONVENTION (Bad_Practice,2)	0/17	0	<0.01
NM_CONFUSING (Bad_Practice,3)	0/6	0	0.01
NM_METHOD_NAMING_CONVENTION (Bad_Practice,2)	2/44	0	<0.01
OS_OPEN_STREAM (Bad_Practice,2)	0/71	0	<0.01
OS_OPEN_STREAM_EXCEPTION_PATH (Bad_Practice,3)	0/5	0	0.018
SE_BAD_FIELD (Bad_Practice,3)	0/11	0	<0.01
SE_COMPARATOR_SHOULD_BE_SERIALIZABLE(Bad_Practice,2)	0/49	0	<0.01
SIC_INNER_SHOULD_BE_STATIC_ANON (Performance,3)	0/92	0	<0.01
URF_UNREAD_FIELD (Performance,2)	33/259	0	<0.01

Java, as the GC_UNRELATED_TYPES "good" issue demonstrates. Also the issues on the comparison of Strings or Objects with == (Bad Practice) could be related to this problem. The remaining issues of category Bad Practice do not signal bugs but do indicate code that could lead to a waste of resources or to difficulties in maintenance. Finally, there are 2 issues in the category Correctness in the list: the infinite loop (IL_INFINITE_LOOP) and the null pointer dereference in some path (NP_NULL_ON_SOME_PATH). We checked them manually and we discovered that they are actually errors: however all the 9 detections are on unfeasible paths, and this is probably the reason that students did not notice these errors with tests execution. Thus, we decided to remove the two issues of category Correctness from the list of bad predictors. The definitive set of bad predictors is reported on Table 12.

A further control strategy against external threat is applied in this paper with the external validation towards Apache Lucene. We observed that some issues remain Bad Defect Predictors even when we switch context and we analyze real projects. The Generally Bad Defect Predictors, that are the 1% of the total number of issues in FindBugs (4 in 359), were responsible of 62 detections in Lucene (18.7% of the total) and 324 in the students' projects repository (19.1% of the total): these digits are the basis of an important practical application of our findings, i.e. the adoption of a filtering strategy that can avoid information overload on developers caused by a very large number of detections with a very low probability to be related to defects. In particular fixing issues with a low probability of being related to a defect is dangerous because we know from Adam's law [32] that the probability of introducing a new error during a fault correction is always greater than zero. The ranking of Bad Defects Predictors could be adopted by developers that want to enable only those issues with the highest precision. For instance, in this experiment, the Good Defects Predictors issues are just 4 out of 359 in FindBugs 1.3.8 database (about 1%) and they are responsible for only the 4.4% of all detections in lab versions. Finally, the Context-Specific Bad Predictor issues (about 3% of the complete set) produced a further 15% of detections in lab versions. Therefore, summing up the Context Specific Bad Defects Predictors and the Generally Bad Defects Predictors, we obtain that more than a third of the detections in the students projects (34%) were not related to defects and could be ignored. In Lucene they were the 88%. These comments are summarized in Figure 13 and Figure 14 . Finally, from an educational perspective, although the occurrences of good predictors are few, we consider them important topics to be stressed more in future iterations of the OOP course.

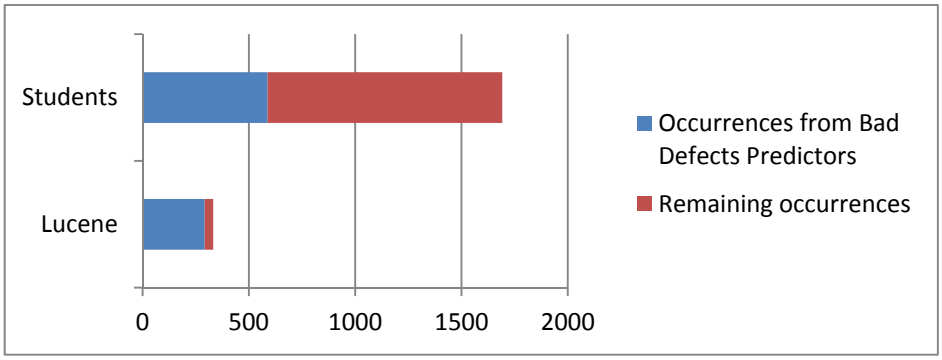

Figure 13. Bad defect predictors proprtions (Students projects vs Lucene)

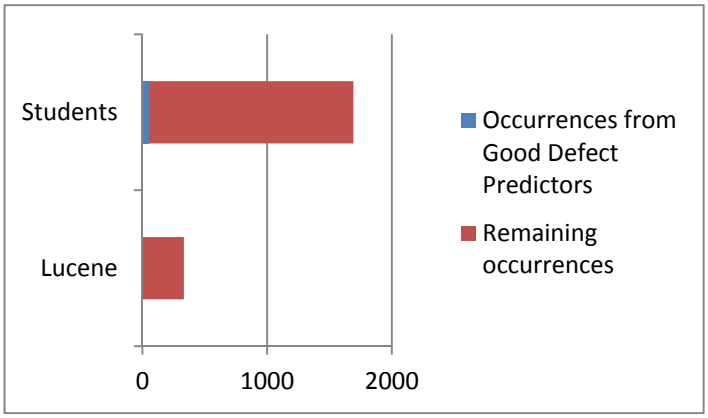

Figure 14. Good defect predictors proprtions (Students projects vs Lucene)

2.5. COMPARISON WITH PREVIOUS AND RELATED WORK

The main threat to validity of the experiments with students is the generalizability of the findings because we analyzed small projects developed at our University. We faced such an external validity threat by conducting an external validation, repeating the analysis with the same methodology on a real project, Apache Lucene.

The cross validation wasn't applicable to the set of Good Defect Predictors since none was found, but to the Bad Defects Predictors set only. Four issues among the latter were in common with the students project analysis (they produced the 19% of total detections in LAB version and 18% in Lucene 3.1.0).

In the first experiment [33], we analyzed a small repository of Java projects (85 projects) and we studied the precision of issues at group level (combination of category and priority): the analysis showed that only 2 groups (Bad Practice High, Correctness High) out of 15 groups of issues could be considered as reliable predictors of actual defects, and one group of issues (Bad Practice Medium) had a precision that was practically negligible.

We group the good and bad defects predictors issues found in this study by the same criteria of the previous work to compare the findings. Since the repository in the replicated study is larger, we find more kinds of issues and more groups, however the group Correctness High is still in the good defects predictors set as well as Bad Practice Medium is still in the Bad Defect Predictors set. While the group Bad Practice High is not present in either of the two sets. Therefore, 2 out of 3 groups are confirmed in the respective sets and, most important, there are no conflicts in the group compositions of the two studies: we conclude that the finding of our previous work are confirmed and improved in this study.

Before us, Boogerd and Moonen [20] [21] and Kim and Ernst [30] also used temporal and spatial coincidence in their research. Our research confirms the findings of Boogerd and Moonen: a reduced set of rule violations are strongly related to defects in source code, and many violations are not related to real defects at all. Kim and Ernst also applied FindBugs on Lucene in another paper [37], and they reached a similar finding:

only 9% of the warnings issued by FindBugs were removed by a fix-change in the period 10/19/2001~11/9/2006.

Furthermore, in our analysis there are 3 high priorities issues and 1 medium priority in the good defects predictors, whilst the majority of issues in the bad predictors set are medium and low priority (respectively 10 and 5 issues, 1 high priority): thus the tool's default prioritization of issues seems to be effective, in contrast with what is found by Kim and Ernst [30] in two open source projects (Columba and jEdit).

Differently from our second experiment [38], the new comparison includes the validation on Lucene and it is performed with more objective criteria, specific to each study. We couldn't apply the same criteria because every study had its own methodology.

The first comparison we present is that with Kim and Ernst [30]. The authors list the FindBugs issues with shortest and longest "life" in multiple versions of two open source projects: the underlying idea is that if some issues are resolved quickly by developers, those issues are important and likely related to real defects. The issue SA_FIELD_SELF_ASSIGNMENT, which we identify as good defect indicator, is also among the issues with shortest life in one of the two projects analyzed by Kim and Ernst (Columba). The bad defect predictors issues in common are instead two: ES_COMPARING_STRINGS_WITH_EQ and OS_OPEN_STREAM, both of Category BadPractice and Priority 2. Furthermore, we do not observe any conflict with this study, i.e. none of our "good" issues have long life in the experiment of Kim and Ernst and none of the "bad" have short life.

We are also able to compare our findings with the findings of Ayewah and Pugh [39], who analyzed thousands of FindBugs warnings fixed by engineers during the May 2009 "Google FixIt". The authors used a lightly modified temporal coincidence to find which FindBugs issues were fixed with higher frequency in the Google code base in a period of 9 months. In their paper they show 12 issues with high removal rate and 3 with low removal rate, distinguishing issues only by bug pattern and category. We found in the first set 3 out of 4 of our "good" issues (only the self-assignment is not present). Moreover, "our" bad issue DM_NUMBER_CTOR (Performance) has low fix rate. We didn't find any conflict also with this study.

In a previous work of the same authors [19], they tried to understand the efficiency of FindBugs by manually checking medium/high priority Correctness issues signalled on 3 projects: Glassfish v2, JDK 1.6.0-b105 and the Google code base. The authors classified issues in different groups, based on their impact on code. Overall, they observe that in JDK 1.6.0-b105 almost 50% of them had an impact (misbehaviour of the program), a further 10% had a serious impact, 160/379 were trivial, whilst 5 issues were due to bad analysis by FindBugs. The comparison with this work is more difficult because the authors reported the issues descriptions but not the issues Ids. We applied the following strategy to make the comparison possible:

- We looked at the issue description and try to find the respective issue ID on FindBugs website (section Bug Bug descriptions. We considered the issue only if the description corresponded univokely to an ID.
- We applied our precisions threshold to their results with the following schema:
  - NI = Number of reviews “Impact” + Number of reviews “Serious”
  - NA = Total number of reviews
  - Precision = NI/NA
  - If precision >= 0.50, then the issue is a good predictor, otherwise it is a bad predictor

We were able to unambiguously identify 2 issues: the uninitialized read of field in constructor (1 detection had impact and 7 tagged as trivial, resulting in a Bad Predictor and then a conflict) and the self-assignment of field (1 “impact” and 2 “trivial”), that is confirmed as good predictor.

In the same paper the authors provide the results of a similar review that was performed at Google, where they classified issues reviewed in impossible (i.e. wrong detection), trivial, open, fixed. We applied the same strategy but we defined NI = fixed. Looking at the results, 2 out of 13 uninitialized reads were fixed (but 7 were wrong), whilst all the 7 detections of the issue GC_UNRELATED_TYPES (Correctness) were still open.

Among the 12 detections of the self-assignment issue, 5 were fixed, 1 classified as trivial and the remaining left open.

We present a synopsis of the above comparisons in Table 13, according to the following binding id-study on columns:

1. Ayewah et al 2007 [19]
2. Kim and Ernst 2007 [30]
3. Ayewah and Pugh 2010 [39]
4. Validation of Vetro' et al. 2011 [38] on Lucene

X marks in the table indicate that the corresponding issue was identified as related/not related to defects in that study. The areas in gray represent the agreements. The only study that presented conflicting results is study 1 [19], that in any case resulted in conflict with their later study conducted at Google. Overall, we obtained a proportion of agreement within studies of 70.59 % (100% without considering study number 1).

Table 14 reports the comparison of our findings with related work on a project basis instead of paper base: we observe a 50% of agreement in industry projects and 100 % in open source projects. Finally, stratifying data by Good/Bad Predictors, we observe that agreement has the following proportions: 55.56% Good Predictors, 87.50% Bad Predictors. In summary, the highest agreement of results in the comparison between our study and related work in reached in Open Source projects and in the set of the Bad Defects Predictor issues. We include in the Generally Bad Predictors issues set also the issues coming from the meta-analysis conducted, i.e. with at least an agreement with another study and no disagreement. The new elements are the following ones:

- ES_COMPARING_STRINGS_WITH_EQ (Bad_Practice,2)
- DM_NUMBER_CTOR (Performance,2)
- NM_METHOD_NAMING_CONVENTION (Bad_Practice,2)
- SE_BAD_FIELD (Bad_Practice,3)
- URF_UNREAD_FIELD (Performance,2)

Given the large variety of contexts, we assert that these issues have a very low probability to detect a defect. Although the double internal empirical validation on the Good Defects Predictors confirmed them, the comparison with the related work showed that they are context specific, differently from the Bad Predictors.

Table 13. Summary of comparisons with related work (paper-based)

	Issue ID	Related to defects				Not related to defects			
		1	2	3	4	1	2	3	4
Good	GC_UNRELATED_TYPES (Correctness,1)			X		X			
	SA_FIELD_SELF_ASSIGNMENT(Correctness,1)	X	X			X			
	UR_UNINIT_READ (Correctness,1)			X		X X			
	UUF_UNUSED_FIELD (Performance,2)			X					
Bad	DM_NUMBER_CTOR (Performance,2)							X	
	DM_STRING_CTOR (Performance,2)								
	DM_STRING_TOSTRING(Performance,3)								
	EQ_COMPARETO_USE_OBJECT_EQUALS (Bad_Practice,2)								
	ES_COMPARING_STRINGS_WITH_EQ (Bad_Practice,2)						X		X
	NM_CLASS_NAMING_CONVENTION (Bad_Practice,2)								
	NM_CONFUSING (Bad_Practice,3)								
	NM_METHOD_NAMING_CONVENTION (Bad_Practice,2)								X
	OS_OPEN_STREAM (Bad_Practice,2)	X					X		
	OS_OPEN_STREAM_EXCEPTION_PATH (Bad_Practice,3)								
	SE_BAD_FIELD (Bad_Practice,3)								X
	SE_COMPARATOR_SHOULD_BE_SERIALIZABLE(Bad_Practice,2)								
	SIC_INNER_SHOULD_BE_STATIC_ANON (Performance,3)								
	URF_UNREAD_FIELD (Performance,2)								X

Table 14. Summary of comparisons with related work (project-based)

Projects	Agreements			Disagreements			Proportion of Agreement		
	Good	**Bad**	**Tot**	**Good**	**Bad**	**Tot**	**Good**	**Bad**	**Tot**
Industry									
Google (Ayewah et al 2007)	0	0	0	3	1	4	0.00%	0.00%	0.00%
JDK 1.6.0-b105 (Ayewah et al 2007)	1	0	1	1	0	1	50.00%	NA	50.00%
Google (Ayewah et Pugh 2010)	3	1	4	0	0	0	100.00%	100.00%	100.00%
Tot Industry	*4*	*1*	*5*	*4*	*1*	*5*	*50.00%*	*50.00%*	*50.00%*
Open Source									
Columba (Kim and Ernst 2007)	1	1	2	0	0	0	100.00 %	100.00%	100.00%
jEdit (Kim and Ernst 2007)	0	1	1	0	0	0	NA	100.00%	100.00%
Lucene (current paper)	0	4	4	0	0	0	NA	100.00%	100.00%
Tot Open Source	*1*	*6*	*8*	*0*	*0*	*0*	*100.00%*	*100.00%*	*100.00%*
TOT	***5***	***7***	***12***	***4***	***1***	***5***	***55.56%***	***87.50%***	***70.59%***

2.6. AN INDUCTIVE STUDY AS A CONTRIBUTION TO THE SECOND RESEARCH STREAM

We helped an industrial partner in understanding the usefulness and effectiveness of the Resharper ASA tool in their development projects and we decided to adopt the second approach because it is more promising than the first one, as summarized in Section 2.2. Our long term aim is to provide our partner with models that use ASA issues to point to more specific quality problems. These models should be able to make recommendations for code inspections based on a set of quality characteristics of interest. For example, a security inspection should be able to use a prediction model pointing to software files and components with potential security flaws. Or, the user experience review should be able to use a model that selects parts of the software with potential usability problems.

The main novelties that we introduce with respect to the previously conducted related work are:

- We contribute to the body of evidence of the second research stream by adding a new tool/language/application combination (Resharper/ C#/ Web application). The Resharper tool has, to our knowledge, not yet been evaluated in past work.
- We perform the analysis at two granularity levels, i.e. software components and source code files. Components are high-level functional units encapsulating one or more main functionalities of the software system, such as: “User Login”, “Database Access”, or “Admin Backend”. Source code files are low-level artifacts, usually containing classes that are the building blocks for components. Since past studies were done at only one of the two levels this study will give some insight into the comparison between the two levels.

We investigate whether specific types of ASA issues can be linked to specific quality dimensions. This is helpful to understand if an increased importance of one quality dimension, such as usability, can help to pre-select the set of ASA issue types that will

predict usability defects with the highest precision. Or more generally, the approach can be used to prioritize the set of ASA issues a reviewer would have to inspect, based on a prioritization of desired quality characteristics.

To our knowledge, no past work has yet studied the correlation between ASA issue types and quality characteristics. The most similar works we found were two studies that investigated instead the typology of defects found by ASA tools. The first one is a study conducted by Nagappan et al. [24], who classified defects found by the FlexeLint tool using the ODC classification schema [40] and found that defects associated with ASA fell into three ODC defect types: checking, assignment/initialization, and interface. Wagner et al. [17] also classified ASA issues, but they focused on their effect on code rather than their causes. The authors used a 5-point scale of severity to classify the true positive issues signaled by FindBugs, PMD and QJPro on five industrial projects. The highest category level was "Defects that lead to a crash of the application", while the lowest was "Defects that reduce the maintainability of the code". The authors found that most of the true positives were related to maintainability of the code (e.g., readability and changeability). They also compared ASA issues with defects found using code reviews and unit tests, and they discovered that all defects found by ASA tools were also found by the review, while testing activities found different categories of defects.

We adopt a different perspective from these two studies, and we focus on whether any ASA issues can predict defects relating to a very general set of software quality attributes, using the well-known ISO/IEC 9126 quality model [41] as a basis for defect classification. The ISO/IEC 9126 Software engineering Product Quality Model is an international standard for the evaluation of software quality. It defines a quality model with six main characteristics namely, functionality (F), reliability (R), usability (U), efficiency (E), maintainability (M), and portability (P), which are further broken down into 22 sub-characteristics. The standard was revised in March 2011 by the ISO/IEC 25010 standard committee [42]. Our defect classification based on the standard was created two months after the new standard was released, but we decided to keep the old standard because of its widespread use and because of the large overlap between the two.

We proposed our defect classification in a previous work [43]; it is complementary to already existing defect classifications because it helps in understanding the impact of the software on different quality attributes. Such classification might help programmers and managers with practical tasks, such as the prioritization of defects according to the different stakeholders' interests, the ease of process improvement measurement on specific quality dimensions or tuning verification activities according to specific quality dimensions. This work is specific to the latter point and it is a first step towards understanding whether different ASA issues could be related to specific quality dimensions.

The first goal of this study is to understand whether some predefined subsets of ASA issues (a.k.a. ASA issue categories[6]) are eligible as indicators of defect-proneness. The second goal is to understand whether and which categories of ASA issues are related to specific software quality dimensions. Both questions are analyzed at two levels of granularity: firstly with respect to components, and secondly source code files. The rationale behind the decision to perform analysis on different levels is to better comprehend if results would differ, be the same, or even contradict each other.

STUDY CONTEXT

The study was carried out at a software company that develops web-based applications in C# (using .NET and Visual Studio). The company uses the JIRA tracking system[7] to record defects.

Of the current projects at this company, we selected one for in-depth analysis based on data quality. Preliminary analysis showed that data quality varied considerably between available projects, reflecting the level of process conformance [44] with which developers recorded defects in JIRA. We chose the project with the best data quality

[6] Issue categories vary depending on the ASA tool used. Typical categories for the Resharper tool used in this study are: Redundancies in Code, Common Practices and Code Improvements, Compiler Warnings, etc.

[7] http://www.atlassian.com/software/jira

(according to the three criteria below) to reduce the influence of incomplete or noisy data on the results:

A. Number of empty fields in defect reports (e.g. missing data).

B. Number of defect report fields that were filled with the default value (which may indicate the default value was accepted rather than that the true value was investigated).

C. Percentage of components that could be bound to files (our approach for this is described below).

The selected application has about 35 KLocs and has been active in production since November 2009, with 4 developers working on it in parallel. At the time of the analysis, the JIRA system contained 78 fixed and closed defects for the selected project (which we will call J).

MAPPING BETWEEN ASA ISSUES, DEFECTS, FILES, AND COMPONENTS

Our methodology for performing the mapping between components, files, and ASA issues, as illustrated in Figure 15, is based upon the fact that JIRA systems can track not only defects but any other element that can be associated with software artifacts. Those elements are called "JIRA issues", and each project has its own set of issues. Example of JIRA issues are change requests, system incident reports, implementation tasks, etc. Moreover, developers establish links between files in the SVN code repository to JIRA issues by including ticket ids in their SVN commit comments. Finally, each JIRA issue is linked by the software developers to one or more software components.

With this information one can build a frequency table (see Figure 16) of files (rows) and components (cells) indicating how often files were changed (i.e. added, modified, or deleted) when working on a component. If a JIRA issue is related to one or more logical components, then the set of modified files belong to the respective components. Using this method a mapping is built based on evidence of how the system changed and evolved over time.

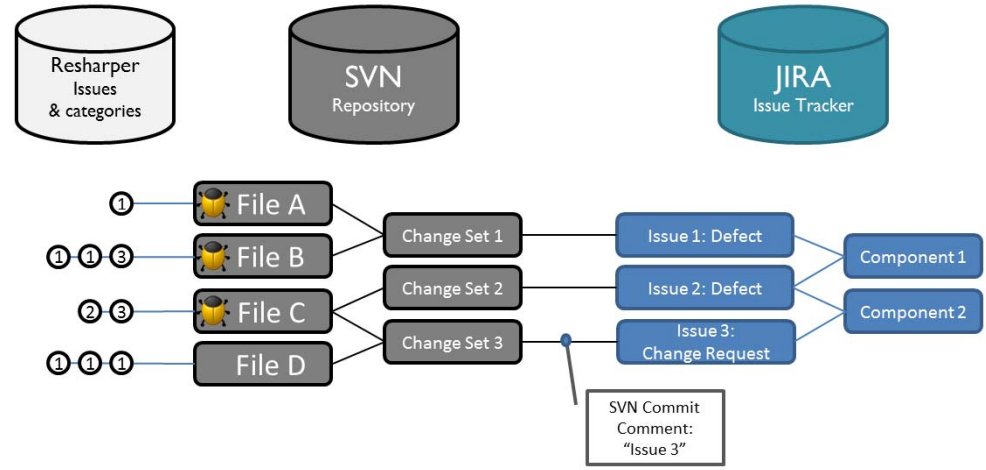

Figure 15. Linkage between Resharper issues, source code files, issue and defect fixes, and components. Yellow defects indicate that a file is linked to at least one defect issue in JIRA.

	Component 1	Component 2
File A	1	0
File B	1	0
File C	1	2
File D	0	1

	Component 1	Component 2
File A	X	
File B	X	
File C	X	X
File D		X

Figure 16. Evidence-based binding of files to logical components

Since a file can belong to many logical components, we accept multiple classifications. Further we reduce some possible noise by mapping a file only to a component if it was linked to this component in at least 20% of all the files' changes. This percentage was set after an analysis of frequency distributions.

STUDY EXECUTION

We derive from our first goal two research questions on component (C) and file (F) level:

- RQ C1: Which ASA issue categories can identify defect-prone components?
- RQ F1: Which ASA issue categories can identify defect-prone files?

Additional research questions are derived from our second goal:

- RQ C2: Which ASA issue categories can point to defect-prone components that impact various system quality characteristics?
- RQ F2: Which ASA issue categories can point to defect-prone files that impact various system quality characteristics?

We address these questions inductively, investigating whether the detection of defect-proneness was possible and if so, which types of ASA issues were useful for doing so. We discuss the metrics and the methodology separately for each research question below.

RQ C1: Which ASA issue categories can identify defect-prone components?

To answer RQ1-C1, we first performed the mapping as described earlier to link Resharper issues to components. Secondly, we checked to see if the number of Resharper issues is correlated with software size. This step was necessary to investigate a possible bias from code size. If such a correlation exists, it is necessary to normalize the data (e.g. by using issue density instead of number of issues). The same analysis is done for defects.

In a third step we test for correlations between numbers of defects and numbers of Resharper issues in each Resharper category, per component. We use the Spearman coefficient correlation (a non-parametric statistic), since we observe a wide range of issues

and defects that do not appear to follow any defined distribution (see Table 15 and Table 16).

RQ F1: Which ASA issue categories can identify defect-prone files?

To answer this research question we used again the mapping procedure from sub-section 3.2. We also checked for possible bias as described in RQ-C1. Lastly, we tested for correlation between Resharper issue categories and defects by using a two sample Mann-Whitney test [23] after running an unsuccessful Shapiro test for normality. This type of test was more appropriate than the Spearman correlation due the sparseness of the data; it has also been used in previous studies [21] [33]. As the results will show, only a small number of files (about 10%) were associated with defects. Therefore, we partitioned the sample into non-defect-prone files and defect-prone files in order to perform the Mann-Whitney test. This decision implies that the analysis will investigate if files with at least one defect can be identified by the Resharper issues residing in the same file.

RQ C2: Which ASA issue categories can point to defect-prone components that impact various system quality characteristics?

RQ F2: Which ASA issue categories can point to defect-prone files that impact various system quality characteristics?

For both of these research questions, we used the ISO/IEC 9126 quality model as a basis for classifying the defects according to different quality characteristics. The method for classifying defects in this way was developed and validated in a prior experiment [43], which also used the same project as the subject project. In that study, six different subjects, divided into two groups with respect to their expertise, classified the 78 defects using the ISO/IEC 9126 quality main characteristics and sub-characteristics. Subjects read the defect reports and assigned each defect to one or more quality characteristics and sub-characteristics (the classification is not orthogonal). The underlying idea is that each defect reduces a software capability and impacts the corresponding characteristic and sub-characteristic.

Table 15. Resharper issues detections

Resharper category	Number of issues
ASP.NET	2
Common Practices and Code Improvements	521
Compiler Warnings	36
Constraints Violations	445
Language Usage Opportunities	591
Potential Code Quality Issues	14
Redundancies in Code	645
Redundancies in Symbol Declarations	82
Unused Symbols	7
Sum of issues	2343

Table 16. Resharper issues on components

Component	Sum of ReSharper issues	Defects	NCSS
Cmp 1	1407	43	3192
Cmp 2	324	13	961
Cmp 3	232	6	711
Cmp 4	29	5	97
Cmp 5	7	4	9
Cmp 6	29	4	97
Cmp 7	0	3	0
Cmp 8	119	2	246
Cmp 9	93	1	208
Cmp 10	0	0	0
Cmp 11	428	0	1392
Cmp 12	0	0	0
Cmp 13	0	0	147
Cmp 14	0	0	0

We observed that more experienced software engineers produced classifications with less variability, and that the classification at characteristic level was more reliable than those at sub-characteristics level. As a consequence, we adopted as the final classification the one created by experts at the characteristics level.

Using that classification we were then able to check, in the work described in this paper, whether various types of Resharper issues are correlated to the defects related to specific quality characteristics.

RESULTS

We collected metrics on the revision of the target project preceding the first defect fix commit to include as many defects as possible. Resharper reported 2343 issues on the source code of the web application: Table I reports the issues per each Resharper category and Table II reports, for each logical component, total number of Resharper issues, number of defects and non-commented source statements. Some components have 0 NCSS for two reasons: a component was built after the version of the software analyzed, or the files-component mapping produced zero files for a component, or in some cases both. Resharper reported issues on files with extension .aspx, .xaml, .csproj, .cs (including .xaml.cs, .ascx.cs, .aspx.cs, .ashx.cs, .Master.cs). .

Among the 78 fixed and closed defects, 65 had commits linked to them. According to the experts' classification [43] (Figure 17), the majority of defects (58%) impacted only functionality, followed by usability (26%) and reliability (6%). Mixed classifications (FR and FU) accounted for 5% each, while no defects had impact in the remaining three categories.

The total number of files with at least one defect fix is 58. However, excluding those files that were out of scope of the Resharper analysis (e.g., .sql files, .css files) and those files that were added after the revision we analyzed, only 11 of the 58 remained. These files are listed in Table 17. As with components, the data indicates that there is not a clear relationship between number of defects and Resharper issues: the most defect prone

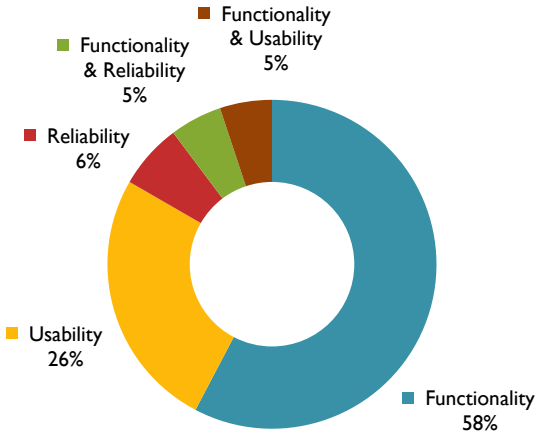

Figure 17. Defect classification

file (C) has 35 issues whereas some of the less defect prone files (G,I,J) have up to twice the issue count.

As this is an exploratory study, when analyzing statistical significance we ran our tests at a 90% confidence level. As we are intending to discover relationships that can be later more rigorously examined, we would prefer to err on the side of finding false positives, rather than missing any relationship.

We now answer separately each research question.

RQ C1-C2: Which ASA issue categories can identify defect-prone components?

Table 18, first column, reports Spearman correlations between Resharper issues densities of specific issue categories and defects. Statistically significant values (i.e., p-value ≤ 0.10) are shown in bold.

We used issue densities (issues/NCSS) in the following computations because a positive Spearman correlation (rho=0.93, pval < 0.01) was found between NCSS and

number of issues. We did not normalize the number of defects because the correlation between defects and size was not significant (rho= 0.42, pval= 0.15).

The total number of Resharper issues has an insignificant but positive correlation with defect-proneness (0.19, p = 0.29) with all defects. Looking at Table 18, column “All RQ C1”, we observe positive correlations for all but one category (Common Practices and Code Improvements), and one (Language Usage Opportunities, rho= 0.57) is significant at the 90% confidence level (in bold). Hence, the answer to RQ C1 is:

Only a few issue categories, such as Language Usage Opportunities in this example, are positively correlated with defects at the component level. Issues in the category Language Usage Opportunities identify optimizations at code level based on specific characteristics of C#. The most frequent detections were:

- Convert 'if' statement to 'switch' statement
- Invert 'if' statement to reduce nesting
- Loop can be converted into LINQ-expression
- Use 'var' keyword when initializer explicitly declares type
- Use 'var' keyword when possible

Possible root causes for this correlation are that the usage of more advances language features leads to less defect (i.e. the more language usage opportunities, the less code features are used in the code). Or, it might be that junior developers use less advanced language features than their more experience peers, and also produce more defect prone code.

Table 17. Defects per file

File ID	Component(s)	Resharper issues	Defects
A	C1,	29	1
B	C1,C2,	15	4
C		35	6
D	C1,	84	3
E		7	1
F	C1,C2,	73	4
G	C3,C1,C2,	73	2
H		1	2
I	C1,	45	1
J	C1,	65	2
K	C5,C9,	7	2

Table 18. Correlation between density of Resharper issue types and defect densities

Defect types:	All RQ1C1	F	FR	FU	R	U
ASP.NET						
Common Practicesand Code Improvements	-0.14	-0.13	-0.34	0.07	0	-0.2
Compiler Warnings	0.3	0.31	0.48	0.28	0.04	0.25
Constraints Violations	0.11	0.1	0.03	0.09	0.23	0.18
Language Usage Opportunities	**0.57**	0.53	0.55	0.5	0.2	0.43
Potential Code Quality Issues	0.54	0.5	0.51	0.44	0.22	0.44
Redundancies in Code	0.52	0.49	0.47	0.33	0.39	0.53
Redundancies in Symbol Declarations	0.42	0.45	0.01	0.28	0.17	0.14
Unused symbols	0.53	0.53	**0.75**	**0.57**	0.33	**0.56**
Sum of Resharper issues	0.19	0.18	0.1	0.09	0.23	0.23

Table 18, columns 2-6, reports on the correlations between Resharper issue densities and defects, divided into the ISO\IEC 9126 quality characteristics. The only category with significant positive correlations (in bold) is Unused Symbols: 0.75 with FR defects, 0.57 with FU defects, 0.56 with U defects. All Unused symbols issues were type members never used. We answer the research question the following way: Only very few indicators can be mapped to defects on the component level, and these indicators point to a wider range of quality characteristics rather than on a single one.

We performed a follow-up analysis to see whether the two categories Language Usage Opportunities and Unused Symbols could be used as defect locators. We tested their

Table 19. Research Question F2 (only statistically significant results)

Quality characteristic – Resharper issue category	Defect prone files			Non defect prone files			Pval
	Mean Resharper issues/NCSS	Sd Resharper issues/NCSS	Nr of files	Mean Resharper issues/NCSS	Sd Resharper issues/NCSS	Nr of files	
F – Constraints Violations	0.14	0.06	6	0.08	0.05	94	0.013
F – Redundancies in Code	0.23	0.14	6	0.09	0.13	94	0.002
FR – Compiler Warnings	0.02	NA	1	0	0.01	99	0.001
FU – Constraints Violations	0.18	0.04	3	0.08	0.05	97	0.002
FU – Redundancies in Code	0.35	0.05	3	0.09	0.13	97	0.004
FU - Sum	0.74	0.09	3	0.53	0.24	97	0.062
R – Redundancies in Code	0.39	0.39	2	0.09	0.12	98	0.033
R - Sum	0.93	0.21	2	0.53	0.23	98	0.029
U – Constraints Violations	0.13	0.07	4	0.08	0.05	96	0.085
U – Language Usage Opportunities	0.15	0.07	4	0.09	0.08	96	0.042
U – Potential Code Quality Issues	0.01	0.01	4	0	0	96	<0.001
U – Redundancies in Code	0.16	0.12	4	0.09	0.13	96	0.033

capability to detect defects earlier than metrics of size and complexity, widely used in the defect prediction literature (e.g., [45] , [46], [47], [48], [49]). Figure 18 shows the cumulative distribution of defects found ranking logical components with respect to the following indicators:

- An ideal indicator that perfectly rank logical components from the faultiest one to the ones with no defect.
- The density of issues of each of the following Resharper issues categories:
  - Unused Symbols
  - Language usage opportunities
  - The density of all Resharper issues.
  - The number of statements (NCSS).
  - The average McCabe complexity.

In other words, the curves in Figure 18 represent how quickly defects would be found if components were tested in different orders, sorted by the criteria listed above. A horizontal line on the graph indicates the point at which 80% of defects have been found.

We observe in Figure 18 that the first 3 components contain 80% of the defects using the ideal locator. Language Usage Opportunities issue density and the total Resharper issue density find 80% of defects at the 5th component, and all the other indicators at the 6th (Unused Symbols, Complexity and Size). The figure also shows that the two selected Resharper categories are overall close to the "all issues" data line which does not consider the category of Resharper issues. This indicates that, at the component level, the distinction between issue categories might lead to small but not vast improvement compared to using all issues.

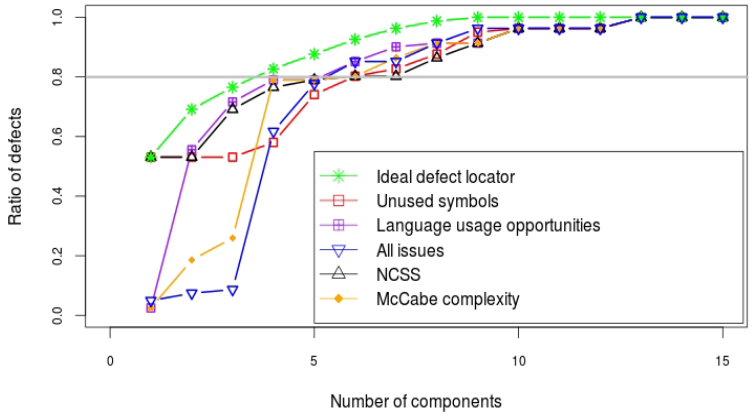

Figure 18. Cumulative distribution of defects in components and indicators

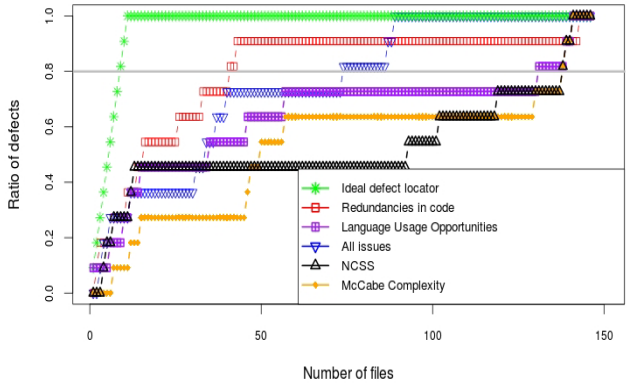

Figure 19. Cumulative distribution of defects in files and indicators

RQ F1-F2: Which ASA issue categories can identify defect-prone files?

Table 20 and Table 19 show, both for defect prone files and non-defect prone files and for each Resharper issue category, mean and standard deviation of Resharper issues densities, the number of files for each set and the p-value of the Mann-Whitney test on the difference between the two sets. Bold percentages indicate p-values that are significant at our chosen confidence level of 90%. Table 19 presents only combinations of Resharper categories and ISO\IEC 9126 defect classifications for which the null hypothesis was rejected.

The categories with highest differences on Resharper issues densities in defect prone/non defect prone files are Redundiancies in Code and Language usage opportunities. Redundancies in Code are related to Functionality and Usability defects, both separately and together. Constraints violations are related to Functionality and Functionality-Usability, while Language usage opportunities only with Usability.

We already presented examples of the issues of the category Language Usage. Examples of Redundancies in Code are:

- Assignment is not used
- Explicit delegate creation expression is redundant
- Expression is always 'true' or always 'false'
- Redundant boolean comparison
- Redundant cast
- Redundant 'else' keyword
- Redundant explicit type in array creation
- Redundant 'this.' qualifier

We performed the same follow up analysis that we did for components and we report in Figure 19 the cumulative distribution of defects found ranking files with respect to the following indicators:

- an ideal indicator that perfectly rank logical components from the faultiest one to the ones with no defect;

- the density of issues of each of the following Resharper issues categories:
- Language Usage Opportunities
    - Redundancies in code
    - the density of all Resharper issues;
    - the average McCabe complexity ;
    - the number of statements (NCSS).
- A horizontal line in the graphs indicates the point at which 80% of defects are found.
- Results at file level are more diverse than at component level: Selecting files based on the density of Redundancies in code issues outperforms all the other indicators, reaching 80% of defects at the 41st file (compared to the 9th file of the ideal locator). The second best indicator is the sum of Resharper issues: however, it reaches the threshold at the 74th position. NCSS and McCabe complexity are less precise indicators at file level: they are able to identify the 80% of defects only very late: a user will have to examine at 90% of all files before capturing 80% of all defect prone ones.

Overall we answer the research questions on file level the following way:

1. Multiple Resharper categories are good candidates for building predictive models for defect prone modules.
2. There is a set of promising candidates of Resharper categories that is able to predict the quality impact of defect more precisely.

In a follow up analysis we picked two quality characteristics of interest, Functionality (F) and Usability (U), and plotted the same graphs as before (see Figure 20

and Figure 21) for the respective significant issue categories from Table 19 . In both cases Redundancies in Code is a more efficient predictor than the sum of all issues.

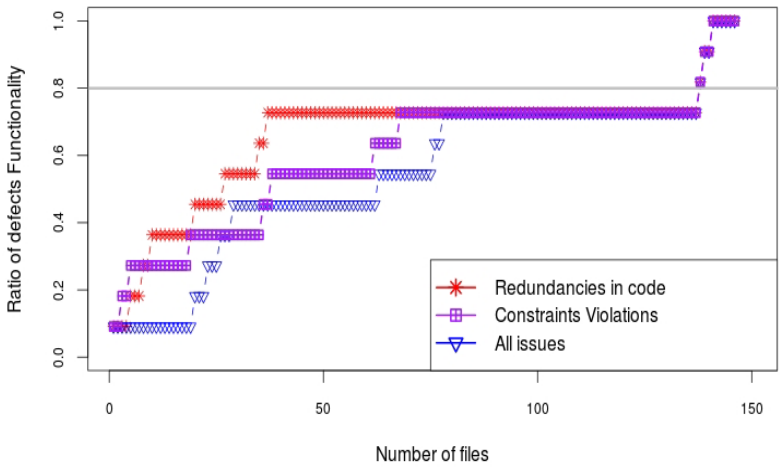

Figure 20. Predictor Performance for Functionality

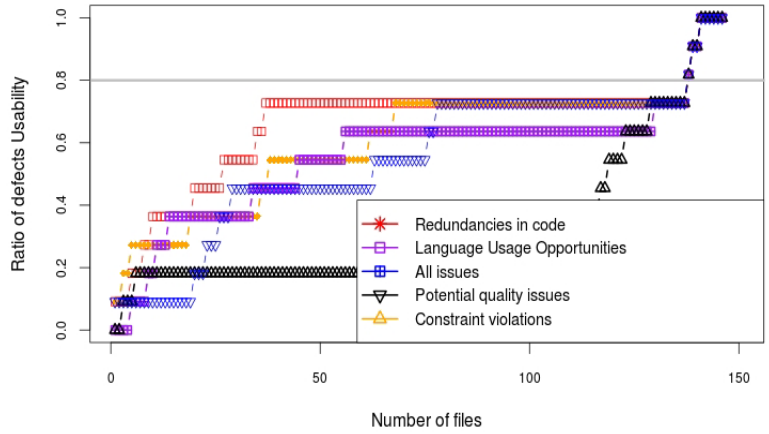

Figure 21. Predictor Performance for Usability

Table 20. Research Question F1: results

	Defect prone files (11)		Non defect prone files (101)		Pval
Resharper issues	**Mean Resharper issue/NCSS**	**Sd Resharper issues/NCSS**	**Mean Resharper issues/NCSS**	**Sd Resharper issues/NCSS**	
ASP.NET	0	0	0	0	NA
Common Practices and Code Improvements	0.13	0.19	0.21	0.18	0.983
Compiler Warnings	0	0.01	0	0.01	0.333
Constraints Violations	0.13	0.05	0.08	0.05	**0.014**
Language Usage Opportunities	0.14	0.07	0.08	0.08	**0.026**
Potential Code Quality Issues	0	0.01	0	0	**0.021**
Redundancies in Code	0.27	0.20	0.08	0.11	**<0.001**
Redundancies in Symbol Declarations	0	0	0.06	0.1	0.969
Unused.Symbols	0	0	0	0	NA
Sum	0.67	0.24	0.52	0.23	0.133

DISCUSSION

The presented data indicates that the answer to the research questions is not straight forward in all cases. Most statistics on component level were rather inconclusive and showed only small correlations or a small set of useful issue categories. We believe that this indicates the high-level component view is perhaps not the right perspective for future research direction. The more promising results showed on file level, even if we had to deal with a sparse data set. The results indicated that number of promising indicators is larger, and this also holds for the number of categories pointing to specific quality problems.

On both analysis levels we could improve the defect prediction quality by using selected single predictors, e.g. as Figures 4-7 show. Results also indicate that ASA issues are more promising to be good defect predictors than traditional software metrics, such as complexity or size.

Some of the inspected issue categories, such as redundancies in code and unused symbols (both components and file level) indicate problems regarding memory waste. In the previous experiment with students data [38] we also found a correlation between a similar category of FindBugs issues (unused variables) and defects in students' projects. We commented that this correlation could be the consequence of the programmers' difficulties in the design of the class, because they planned to use more/different variables that indeed were not necessary. A similar explanation could be extended for these categories of Resharper.

Further, some of the issues of category Language Usage Opportunities can also be an indicator of the level of programmers' knowledge on the language.

THREATS TO VALIDITY

We identify a first construct threat in the mapping files- components. Even though this heuristic eliminates the subjectivity of the manual mapping, 18% of the files were not assigned to any component.

Another threat is subjectivity in the ISO 9126 defect classification. We controlled this threat selecting the most reliable classification made by the experts. A more comprehensive discussion of this threat is found in the original study [43].

The small number of components and of files with defects (11) makes statistical significance and a definitive answer to our research questions hard to obtain. We were aware of this threat and also for this reason we performed an explorative study and findings will be evaluated and better investigated in future work.

As in any inductive study, the generalization of these findings is debatable because they are tied to the specific context of the analysis. Our research design reflects this

concern: in this study we were focused on identifying whether there was any evidence that Resharper issues could be used as early indicators of defect-prone parts of the system, and especially whether estimates could be made regarding the type of quality impacted by those defects. Having obtained an initial indication that this is in fact a feasible approach, further study is necessary to determine whether the specific correlations found in this study can be replicated elsewhere.

2.7. CONCLUSIONS

We analyzed the state of the art and we found two main research streams: I) looking at single ASA issues to identify defects in single lines of code or II) looking at large sets of issues as early indicators of the more defect-prone modules (e.g. classes, files, software components).

Regarding the first research stream, we conducted two case studies: we analyzed the relationship between FindBugs issues and defects on two different pools of University Java projects [33] [38], using information on changes in source code and tests failures. We obtained that only 4 issues could be considered as reliable predictors of real defects and 14 issues had a negligible precision. We conducted both internal and external validations. Internal validations included correlation observations with projects quality and a manual inspection of issues. The external validation included a repetition of the study on an open source project (Lucene) and a comparison with similar studies in literature. The validation of results against internal threats confirmed all issues except two (both in the Bad Predictors sets), whilst the external validation showed some different results. We obtained high agreement in open source projects and in the Bad Predictors set. From these observations, we consider the Good Predictors and 9 Bad issues as Context Specific, whilst the remaining 5 issues are confirmed as Generally Bad Defects Predictors.

We can summarize our experience in the form of advices to practitioners that aim at using FindBugs on their code with the goal of predicting defects (not taking into account other quality aspects such as performance or maintainance):

- disable the Generally Bad Defects Predictors (they are unlikely to predict any defect nevertheless they represented the 19% of total detections in LAB versions and 18% in Lucene 3.1.0).
- analyze the history of your software and apply the temporal + spatial coincidence (with information on test failures if present) and identify Context Specific Good/Bad Defects Predictors
- disable the Context Specific Defects Predictors

Regarding the second research stream, the study presented added the following contributions:

- We evaluated a combination tool-language (Resharper,C#) not yet evaluated in past works, up to our knowledge.
- We performed and compared the analysis at two granularity levels, i.e. logical components and files.
- We investigate whether ASA issues are able to identify specific categories of defects belonging to specific quality dimension.

We found that few Resharper categories had positive correlations with defects at component level, while several categories were more efficient at file level. The issues with higher correlations identify problems regarding code readability, performance, and more in general related to maintainability problems. Moreover, classifying the defects according to the ISO 9126 quality characteristics, different ASA issues categories were positively correlated to different quality characteristics.

We compared the capability of Resharper issues to detect the faultiest modules, both at components and files levels with the result that specific ASA issues were more efficient than the sum of them or traditional indicators (i.e. software metrics).

Based on the experience of this study, we provide future researchers with the following set of recommendations:

- Analysis on file level might lead to more promising results than on component level.
- The size of the project should be at least, but preferably larger than our medium sized project, to avoid data sparseness problems as we found in our study.

Considering future research directions, we suggest to better understand if results for specific categories are useful in other environments, or if this approach will always require a process of exploration, data analysis, and tailoring towards a specific software environment. In latter case, the contribution of future research should focus on building practitioner-oriented methods to build such prediction models rather than building new models.

3 MAINTAINABILITY

3.1. DEFINITIONS

Maintainability is defined in ISO/IEC 25010 [42] in the following way: “degree of effectiveness and efficiency with which a product or system can be modified by the intended maintainers”. The standard also specifies what a modification is: “modifications can include corrections, improvements or adaptation of the software to changes in environment, and in requirements and functional specifications. Modifications include those carried out by specialized support staff, and those carried out by business or operational staff, or end users. Maintainability “includes installation of updates and upgrades”, and it is built atop five sub-characteristics:

- Modularity: “degree to which a system or computer program is composed of discrete component such that a change to one component has minimal impact on other components”
- Reusability: “degree to which an asset can be used in more than one system, or in building other assets”
- Analysability: “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”.

In practice, Maintainability can be interpreted as "either an inherent capability of the product or system to facilitate maintenance activities, or the quality in use experienced by the maintainers for the goal of maintaining the product".

We studied Maintainability in relation to the concept of Technical Debt.

Technical debt is a metaphor that describes the trade-off between the short-term payoffs (such as a timely software release) of delaying some maintenance activities and the long-term consequences of those delays [50].

As Cunningham described [51], in a software development project, rushing implementation to meet pending deadlines "is like going into debt. A little debt speeds development so long as it is paid back promptly with a rewrite. Objects make the cost of this transaction tolerable. The danger occurs when the debt is not repaid. Every minute spent on not-quite-right code counts as interest on that debt. . Entire engineering organizations can be brought to a stand-still under the debt load of an unconsolidated implementation…"

The Technical Debt (TD) problem is depicted in Figure 22[8]: any time we degrade the quality of our work we introduce new issues (e.g. defects, hard-to-understand code, tangled design etc.) that will induce extra maintenance effort, i.e. additional costs. The additional costs are in practice the interests that must be repaid to reduce the TD and restore the health of the system, avoiding that future changes in the system become too costly and hard to perform. Following the schema in Figure 22, the interest we pay on debt corresponds to the difference between the present value of future cost of extra maintenance minus the cost we save by choosing a less than optimal practice.

[8] From Marco Torchiano, http://mtorchiano.wordpress.com/2012/02/24/software-technical-debt/

TD has become a well-known metaphor indicating the possibly significant economic consequences of such quick-and-dirty implementations. It has facilitated discussion among practitioners and researchers by providing a familiar framework and vocabulary from the financial domain. Since the metaphor appears to be sound and intuitively understandable by all stakeholders involved in the software development process, it has potential to become a truly universal language for communicating technical tradeoffs.

Identifying TD [52], quantifying the value of debt and make proper decision making [53] are some of the open issues in the current research on TD [50].

The research done for this PhD work is focused on the problem of TD identification at code level. We performed two studies: a correlational analysis with an open source platform (Hadoop) and an industrial case study with a Brazilian company.

3.2. COMPARING FOUR APPROACHES FOR TECHNICAL DEBT IDENTIFICATION: ANALYSIS ON HADOOP PROJECT

One important class of TD is manifested by problematic implementations in code. Many types of such code-based TD can be potentially detected automatically using static program analysis tools that find anomalies of various kinds in the source code.

There are a number of tools designed for this purpose. Some tools are designed to detect design problems such as code smells [54] or modularity violations [55], some are designed to discover design pattern degradations [56], and some are intended to spot potential defects. From a tool user's point of view, the relevant questions are: which tool(s) should be used to inform the existence of TD under what circumstances?, and, Is it sufficient to use one of the tools, or can the usage of multiple tools lead to benefits in finding more TD?

Plus, not all problematic code detected by these tools is worth being fixed. Some detected source code problems are not likely to cause future maintenance problems or affect the overall quality of the system. In terms of the TD metaphor, the TD principal (i.e. the cost of fixing the debt) may be higher than the TD interest being paid on the debt (i.e. the probable future cost of not fixing it). Thus, another question is: Which tools reveal TD that is likely to incur interest?

TD interest is inherently difficult to estimate or measure. Given the data that we had available in this study, we chose to use two proxies for expected interest (hereafter referred to as "interest indicators"): defect- and change-proneness. These proxies are concrete manifestations of problematic code and are related to future maintenance cost, and therefore are useful, independent indicators of likely interest payments. However, they are not the only, and possibly not the best, indicators, since they do not capture other forms of TD interest, such as increasing effort to make changes. However, defect- and change-proneness are flavours of interest that are of concern to practitioners, and thus it is relevant to determine if they correlate with the TD indicators generated by the four approaches studied here. Thus, a lack of relationship between our selected interest indicators and a TD

indicator cannot clearly be interpreted to mean that the associated TD identification approach is ineffective, but that instead it may identify TD that exhibits other forms of interest.

It should be noted that our aim is not to predict either defects or change-proneness in future instances of the code base, as has been done by many other researchers (e.g., [23] [46] [57] [58] [59]). Rather, we are calculating, for a given version of the system, which classes are already exhibiting change- or defect-proneness, and using these indicators as proxies of a construct (TD interest) that is more difficult to measure.

So as a first attempt to compare and contrast different TD identification techniques, we conducted an empirical study to answer the following research questions:

- (RQ1) Considering a set of four representative TD detection techniques (resulting in a set of 25 "TD indicators"), do they report the same set of modules as problematic? If not, how much overlap is there?
- (RQ2) To what extent do any of the techniques for detecting TD in code happen to point to classes with high interest indicators (defect-proneness or change-proneness)?

We first compare the results of applying the different TD identification techniques to 13 versions of the Apache Hadoop open source software project.

Secondly, we investigate whether and how likely the problems detected by these different techniques are related to the two interest indicators we've chosen. This study is a first attempt to map out a "TD landscape" that illustrates the overlaps, gaps, and synergies between a variety of code analysis techniques with respect to their use in identifying TD.

Both research questions are answered through the computation and combination of the pairwise relationship between TD indicators (RQ1) and TD indicators vs interest indicators (RQ2). The aim is to understand which TD and interest indicators point to the same locations (i.e. Java classes).

We use four different statistical association measures: Pearson correlation, conditional probability, chance agreement and Cohen's Kappa. Each of the selected four measures assesses association from different perspectives that complement each other .

Measures are computed for each class and each of the 13 selected versions of the Hadoop software: hence a further aggregation is needed to combine measures from all versions and have a unique association measure for every possible combination of TD indicators (RQ1) or TD indicator vs interest indicator (RQ2).

The results of this study have the potential to improve our understanding about how existing source code analysis approaches can be tailored towards identifying TD. The long-term vision of this work is to create practical approaches that software developers can use to make TD visible, to help assess the principal (i.e. value) and interest (i.e. long-term cost) of the debt, and to assist managers in making educated decisions on strategies for debt retirement.

RELATED WORK

Past research efforts into TD have focused on building techniques that independently identify TD through source code analysis. For instance, Gat and Heintz [60] identified TD in a customer system using both dynamic (i.e., unit testing and code coverage) and static (computing rule conformance, code complexity, duplication of code, design properties) program analysis techniques.

Nugroho et al. [61]also performed static analysis to identify TD. They first calculated lines of code, code duplication, McCabe's cyclomatic complexity, parameter counts, and dependency counts. After that, they assigned these metrics to risk categories to quantify the amount of interest owed in the form of estimated maintainability cost.

A CAST report [9] also presented the usage of static analysis as a way to identify technical debt. The proposed approach examines the density of static analysis issues on security, performance, robustness, and changeability of the code. The authors built a pricing model assuming that only a percentage of the issues are actually being fixed.

[9] Available at http://docs.media.bitpipe.com/io_10x/io_102267/item_465972/whitepaper_77813451881.pdf

The Sonar tool[10] is an open source application that has gained in popularity. It also uses static measurements against various source code metrics and attributes to assess the level of TD in a code base.

The approaches discussed thus far calculate TD holistically, i.e. they yield an overall assessment of the total TD in a system, but do not point to specific problematic parts of the code base, or to specific remedies applicable to those parts. Another approach to TD identification, that attempts to yield more actionable information, is to use source code analysis to identify potentially problematic parts of the code, and to use the results of that analysis to suggest specific changes to be made to that code. Examples of such approaches that have been partly evaluated to be valid TD indicators are code smells [62], grime build up [56] [63] and modularity violations [55]. We discuss these techniques in more detail in Section 4.

This works further evolves the study on these analysis techniques by asking the question of the amount of similarity between them. If it turned out that many, or even all, of the TD indicators point to the same code, one could propose to choose only one of the tools when searching for TD. Alternatively, if each of the techniques selects a unique subset of problems, the usage of multiple tools can be recommended. The relationships among different TD identification approaches have not previously been addressed in the literature.

GOALS AND RESEARCH QUESTIONS

The objective of our research is twofold: the first goal is to compare the similarities and differences between four code analysis techniques in terms of TD identification. We are interested in understanding the degree of convergence and divergence of these techniques and their associations. The second goal is to understand how the problematic code identified by these four techniques relates to our chosen proxies for TD interest, defect and change-proneness, as explained in Section 1. We define the goals of our research according to the Goal Question Metric framework [16].

[10] http://www.sonarsource.org/

- Goal 1: Characterizing the similarities and differences in the problematic classes reported by these four different TD detection approaches, in the context of an open source software project.
- Goal 2: Comparing these four TD detection approaches in terms of their correlation to one subset of interest indicators, namely change-proneness and defect-proneness, in the context of an open source project.

We deduced from the above goals the following research questions:

- R1: Which of these techniques tend to report problems in the same sets of classes?
- R2: Which TD indicators (derived from the four TD detection techniques) correlate with the interest indicators defect- and change-proneness?

CASE STUDY

The application studied is Apache Hadoop[11]. Hadoop is a software library for the distributed processing of data across numerous computer nodes, based on the map-reduce processing model. It provides two key services: reliable data storage using the Hadoop Distributed File System (HDFS) and high-performance parallel data processing using a technique called MapReduce. Data are spread and replicated differently among all the nodes of the cluster, while operations are split so that each node works on its own piece of data and then sends results into a unified whole.

We selected Hadoop because it is a mature project (it has been released 59 times starting from 2 April 2006). We focused our analysis on the Java core packages of the system (java/org.apache.hadoop.*), which includes the common utilities that support the other Hadoop subprojects and provides access to the file systems supported by Hadoop. We focused the analysis from release 0.2.0 to release 0.14.0 (the latest release, at the time this paper was written, is 1.0.3). The system initially had 10.5k NCSS (non- commented source

[11] http://hadoop.apache.org

statements) and 126 Java classes, and grew to 37k NCSS and 373 Java classes by release 0.14.0.

TD IDENTIFICATION TECHNIQUES SELECTED

We selected four main techniques for identifying technical debt in source code: modularity violations, grime buildup, code smells, and automatic static analysis (ASA). We introduce their basic concepts and report on our and other related past work.

Modularity Violations (tool: CLIO). In large software systems, modules represent subsystems that are typically designed to evolve independently. During software evolution, components that evolve together though belonging to distinct modules represent a discrepancy. This discrepancy may be caused by side effects of a quick and dirty implementation, or requirements may have changed such that the original designed architecture could not easily adapt. When such discrepancies exist, the software can deviate from its designed modular structure, which is called a modularity violation. Wong et al. [55] have demonstrated the feasibility and utility of this approach. In their experiment using Hadoop, they identified 231 modularity violations from 490 modification requests, of which 152 (65%) violations were conservatively confirmed by the fact that they were either indeed addressed in later versions, or were recognized as problems in the developers' subsequent comments.

Design Patterns and Grime Buildup. Design patterns are popular for a number of reasons, including but not limited to claims of easier maintainability and flexibility of designs, reduced number of defects and faults [64], and improved architectural designs. Software designs decay as systems, uses, and operational environments evolve, and decay can involve design patterns. Classes that participate in design pattern realizations accumulate grime – non-pattern-related code. Design pattern realizations can also rot, when changes break the structural or functional integrity of a design pattern. Both grime and rot represent forms of TD. Izurieta and Bieman [56] introduced the notion of design pattern grime and performed a pilot study of the effects of decay on one small part of an open-

source system, JRefactory. They studied a small number of pattern realizations and found that coupling increased and namespace organization became more complex due to design pattern grime, but they did not find changes that "break" the pattern (design pattern rot). Izurieta and Bieman [65] also examined the effects of design pattern grime on the testability of JRefactory, a handful of patterns were examined, and they found that there are at least two potential mechanisms that can impact testability: 1) the appearance of design anti-patterns [66] and 2) the increases in relationships (associations, realizations, and dependencies) that in turn increase test requirements. They also found that the majority of grime buildup is attributable to increases in coupling.

Code Smells (tool: CodeVizard). The concept of code smells (aka bad smells) was first introduced by Fowler [62]and describes choices in object-oriented systems that do not comply with widely accepted principles of good object oriented design (e.g., information hiding, encapsulation, use of inheritance). Code smells can be roughly classified into identity, collaboration, and classification disharmonies [67]. Automatic approaches (detection strategies [68]) have been developed to identify code smells. Schumacher et al.'s research [54] focused on evaluating these automatic approaches with respect to their precision and recall, and their other work [69] [70] evaluated the relationship between code smells (e.g., god classes) and the defect and change proneness of software components. This work showed that automatic classifiers for god classes yield high recall and precision when studied in industrial environments. Further, in these environments, god classes were up to 13 times more likely to be affected by defects and up to seven times more change-prone than their non-smelly counterparts.

ASA issues (tool: FindBugs). Automatic static analysis (ASA) tools analyze source or compiled code looking for violations of recommended programming practices ("issues") that might cause faults or might degrade some dimensions of software quality (e.g., maintainability, efficiency). Some issues can be removed through refactoring to avoid future problems. We have analyzed in sections 2.3 and 0 the issues detected by FindBugs on two pools of similar small programs (85 and 301 programs respectively), each of them developed by a different student. Their purpose was to examine which issues detected by FindBugs were related to real defects in the source code. By analyzing the changes and test

failures in both studies they observed that a small percentage of detected issues were related to known defects in the code. Some of the issues identified as good/bad defect detectors by the authors in these studies were also found in similar studies with FindBugs, both in industry [39] and open source software [30][23]. Similar studies have also been conducted with other tools [21] [37] and the overall finding is: a small set of ASA issues is related to defects in the software, but the set depends on the context and type of the software.

DATA COLLECTION

For our analysis we considered 13 Hadoop releases. We ignored the very first one (0.1.0) since CLIO's modularity violation computation is based on the current and previous versions. Across the 13 Hadoop releases, from 0.2.0 to 0.14.0, and all 30 indicators over every class, the total size of our data set was 96,720 data points. Due to limitations in the tools used for TD identification we excluded nested classes from our analysis. To understand the threat to validity, we inspected all versions of Hadoop and found that (depending on the version) 39-45% of all classes were nested classes. We will discuss this threat in Section 6.

It should be noted that the range (i.e., possible values) of each indicator varies. As shown in Table 21, TD indicators that solely express the presence of TD (e.g., the presence of a modularity violation or a code smell on class level) map to 0 (meaning no presence) or 1 (meaning the indicator is present). This is expressed by [0,1] in Table 21. For indicators that can be identified multiple times in a Java class (e.g. code smells on the method level and ASA issues that can be repeatedly detected) the measure indicates how many times the indicator was identified. Table 21 shows this as [0..N]. We measured the presence of grime as well as the absence of design patterns. Even if we cannot be sure that the absence of design patterns is harmful, we included this information to investigate if we can find interesting relationships. Therefore classes not following design patterns received a value of "1" for the indicator. We collected issues reported by FindBugs (version 1.3.9) from the

source code of each Hadoop version, considering all issues of any FindBugs category (Table 21: 17-25) and priority (Table 21: 14-16).

Defect proneness measurement. To link classes with defects, for a bug that was fixed and closed in a version v, we computed which classes were modified during the fix change (identifiable through Subversion repository by using bug links provided in commit comments, e.g. HADOOP-123). The linkage between source code anomalies and their resulting defects is potentially stretched temporally over time. For example, as illustrated in Figure 23. Three ways of computing defect proneness, a bug can be found and reported in version 0.3.0, but may not be fixed until version 0.5.0. We thus measure the defect proneness of a class *c* in version *v* using the following three different ways respectively:

- The number of times class *c* is involved in fixing bugs that were injected in version *v*, that is, the version where the bugs were found and reported.
- The number of times class *c* is involved in fixing bugs that were resolved in version *v*.
- The number of times class *c* is involved in fixing bugs that were alive in version *v*, that is, the bugs were reported before or in version v, and were resolved after or in version *v*.

Change proneness measurement. Following the work of Schumacher et al. [54], we measure the change proneness of class *c* in version *v* as the number of repository changes affecting class *c* divided by the total number of changes in the repository during the class' lifetime (e.g., creation to deletion date).

Size measurement. We chose the Number of Methods in each class as a measure of

Table 21. Indicators used in the analysis

Technical Debt Indicators	Modularity Violations	[1]Presence of Modularity Violation *CLIO*		[0,1]
	Grime	[2]Presence of Grime		[0,1]
		[3]Absence of Design Pattern		[0,1]
	Code Smells *CodeVizard*	Class Level Code Smells	[4]God Class	[0,1]
			[5]Brain Class	[0,1]
			[6]Refused Parent Bequest	[0,1]
			[7]Tradition Breaker	[0,1]
			[8]Feature Envy	[0,1]
			[9]Data Class	[0,1]
		Method Level Code Smells	[10]Brain Method	[0..N]
			[11]Intensive Coupling	[0..N]
			[12]Dispersed Coupling	[0..N]
			[13]Shotgun Surgery	[0..N]
	ASA Issues *FindBugs*	By Priority	[14]High	[0..N]
			[15]Medium	[0..N]
			[16]Low	[0..N]
		By Category	[17]Bad Practice	[0..N]
			[18]Correctness	[0..N]
			[19]Experimental	[0..N]
			[20]I18N (internationalization)	[0..N]
			[21]Malicious Code	[0..N]
			[22]Multi Thread (MT) Correctness	[0..N]
			[23]Performance	[0..N]
			[24]Security	[0..N]
			[25]Style	[0..N]
Other Metrics	Size	[26]Number of Methods *Eclipse Metrics Plugin*		[0..N]
Interest Indicators	Defect Proneness	[27]Number of bug fixes affecting this version		[0..N]
		[28]Number of bug fixes fixed in this version		[0..N]
		[29]Number of bug fixes counting between affected and fixed in this version		[0..N]
	Change Proneness	[30]Change Likelihood		[0.0…1.0]

size.

ANALYSIS METHODOLOGY

In order to investigate the two research questions proposed in Section 3, that is, the overlap between the results generated by these TD detection techniques (TD indicators), and their correlation with defect- and change- proneness (interest indicators), we designed a 5-step methodology that reduces the complex set of indicator values on the large, multidimensional dataset into a graph. The methodology is illustrated in Figure 24:

- Step 1: Compute a set of association measures to examine how each TD indicator and each interest indicator are related to each other.
- Step 2: Apply statistical and significance functions to filter only highly associated pairs of indicators.
- Step 3: Combine the set of three significant association measures into one measure.
- Step 4: Combine measures from each of the 13 Hadoop versions to one set of aggregate measures.
- Step 5: Build visualization and data tables to provide insight into the most strongly associated indicators.

To answer the first research question regarding to the overlap between the results reported by these techniques, we examine the association measures between their respective TD indicators. To answer the second research question regarding interest indicators, we order the TD indicators (and the one size measure) by their level of association with the four interest indicators.

Step1: Compute Statistical Association Measures

We identified different statistical techniques to quantify the relationship between pairs of TD indicators in each version of Hadoop. Because there are many choices for association measurement (e.g. Pearson correlation, conditional probability), we performed a sensitivity analysis to investigate how the results generated from these statistic models differ from each other. This analysis showed that different statistical analysis techniques result in different answers: from the top 50 list of the most highly associated pairs of indicators and metrics generated by each of the statistical analysis techniques, only three pairs were common. Therefore we concluded that using only one single measure of association would be inadequate because different statistic models assess association from different perspectives that complement each other. We thus apply 4 different association models to assess the relation between the 30 X 30 pairs of indicators. The different association measures will be combined in one unique measure in Step 3.

In order to illustrate our methodology, we use a pair of TD indicators, Dispersed Coupling (one type of code smell) and Performance issues (reported by FindBugs), as a running example. The four association techniques are described below:

1. The Pearson correlation between the number of occurrences of Dispersed Coupling and the number of occurrences of Performance issues across all Java classes of one Hadoop version (Figure 24: Step 1, M1). Pearson correlation is widely used in the literature on defect prediction models [46], including the usage of ASA issues as an early indicator of defects [23], and in maintainability prediction [57].
2. The conditional probability of a Java class having at least one occurrence of Dispersed Coupling given that the same class has at least one

occurrence of Performance issue, and vice versa: P(Dispersed Coupling | Performance), P(Performance | Dispersed Coupling) (Figure 24: Step 1, M2). The conditional Probability is used for software defects and maintainability predictions [58] [59].

3. The chance agreement, which is the probability that a Java class holds an occurrence of Dispersed Coupling and Performance issue at the same time by chance. This probability is computed as: P(Performance)*P(Dispersed Coupling)+P(No Performance)*P(No Dispersed Coupling) (Figure 24: Step 1, M3). Chance agreement is at the basis of Cohen's Kappa computation [71].
4. Cohen's Kappa, which is an inter-rater agreement between Dispersed Coupling and Performance issues, and indicates the strength of agreement and disagreement of two raters (e.g., TD indicators). Cohen's Kappa is appropriate for testing whether agreement exceeds chance levels for binary and nominal ratings. Therefore, Dispersed Coupling and Performance issues are in agreement if both occur at least once in the same Java class, or if both of them are not present. In any other case, they are in disagreement (Figure 24: Step 1, M4). Cohen's Kappa was used for the assessment of defect classification schemes [72] and in software process assessment [73]. Used in conjunction with other statistical measures, it prevents the misinterpretation of results possibly affected by prevalence and bias.

The computation of chance agreement, conditional probability and Cohen's Kappa required data transformation for some measures: all metrics ranges [0..N] were reduced to [0,1], where "1" indicates that at least one occurrence of the TD indicator was found in the class and "0" otherwise. For example, for the Number of bug fixes (fixed) in version v, we assign "1" if the Java class was part of at least one defect fix during the analyzed version. The partial loss of measurement resolution is further discussed in Threats to validity subsection.

For change likelihood, where the metric ranges from 0.0 to 1.0 (floating point numbers), we investigated its empirical distribution on each version and selected a threshold value to be used in the data transformation. This threshold was computed so as to guarantee that, on average, only the top 25% of classes with high change likelihood obtained a value of "1" in the data transformation, "0" otherwise. The computed threshold was 0.01, indicating that a Java class that is changed more often than in 1 out of 100 cases is considered change prone.

The same was done for the size metric, Number of Methods. We use a threshold of 11 to guarantee that only the top 25% of classes were considered large. The computation of the four statistical analysis techniques produced four respective matrices shown in Figure 24: M1, M2, M3 and M4. Each of the statistical measures offers a different perspective on the association between pairs of metrics.

The Pearson correlation indicates whether two indicators increase/decrease together following a linear pattern. The conditional probability is useful in understanding the direction of the association, since it indicates whether an indicator is present in a class given that another indicator is present. The chance agreement, instead, is the probability that two indicators are either indicating or not indicating a problem in the same class randomly; in the following section we will see how we use this measure with the conditional probability. Finally, the Cohen's Kappa is useful because the agreement takes into account not only when two indicators are present in the same class, but also when they are not present simultaneously.

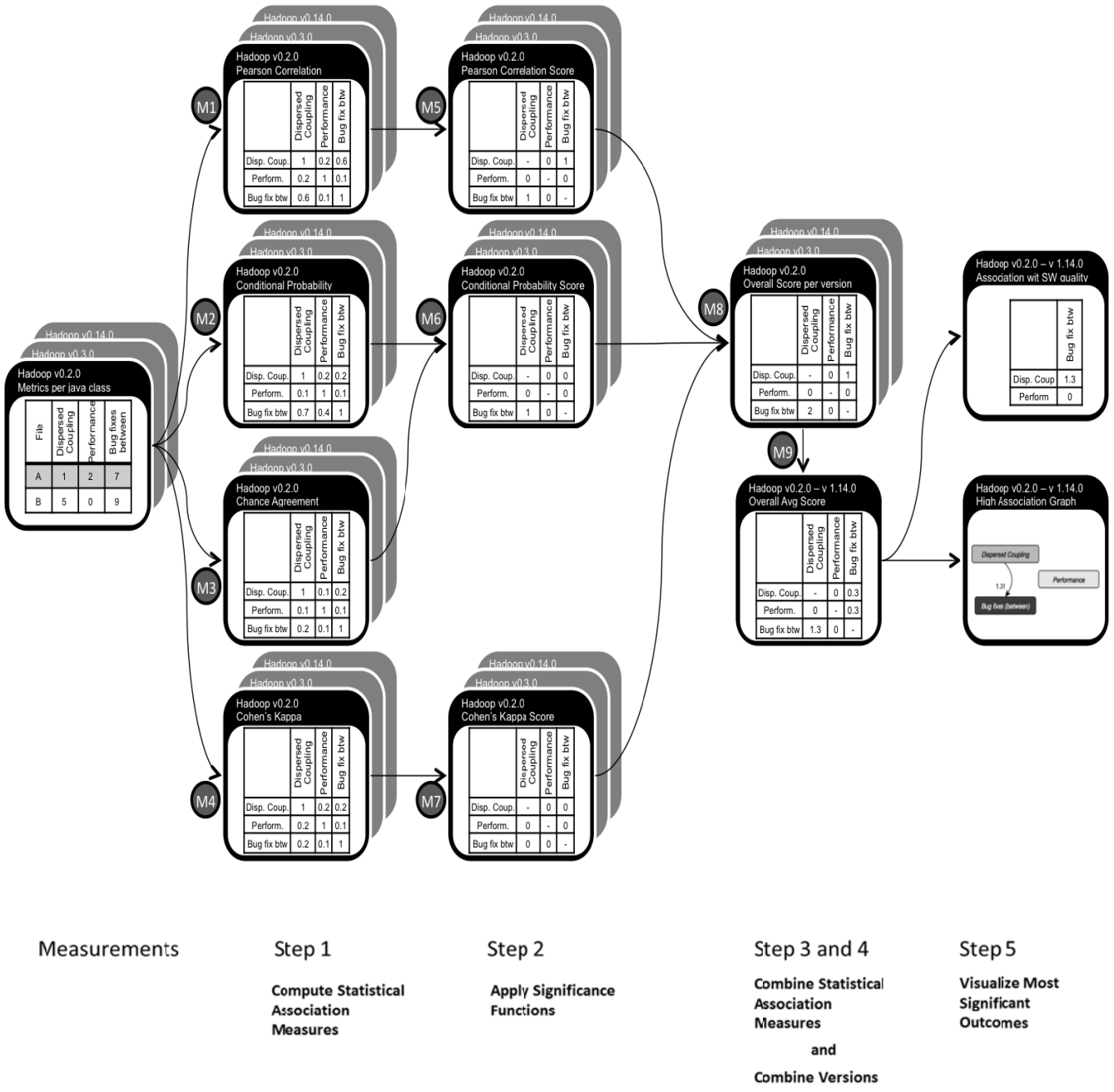

Figure 24. Five-step analysis methodology

Step 2: Apply Significance Functions

After applying the different techniques in Step 1, the goal was to filter the more significant associations from the less significant ones. Therefore we applied different significance functions Ω. The significance functions map the associations between each pair of indicators to [0,1], expressing that the pairs are strongly associated ("1") or not associated strongly enough ("0"). The following three formulae define the functions applied to produce matrices M5, M6 and M7 for each of the 13 versions in Figure 24:

- Significant Pearson Correlations:

$$\Omega(M1_{i,j})=\begin{cases}1, & \text{if}(M1_{i,j})\geq T1 \wedge \text{p-val} \leq 0.05 \wedge (i\neq j)\\ 0, & \&\text{otherwise}\end{cases}$$

- Significant Conditional Probability: using M2 (Cond. Probability) and M3 (Chance agreement):

$$\Omega(M2_{i,j},M3_{i,j})=\begin{cases}1, & \text{if}(M2_{i,j})>(M3_{i,j}) \wedge (M2_{i,j})\geq T2 \wedge (i\neq j)\\ 0, & \&\text{otherwise}\end{cases}$$

- Significant Cohen's Kappa:

$$\Omega(M4i,j)=\begin{cases}1, & \text{if}(M4i,j)>T3 \wedge (i\neq j)\\ 0, & \&\text{otherwise}\end{cases}$$

The goal of the significance function Ω is to discern significant relations from insignificant ones. Parameters T1, T2 and T3 are three specific thresholds of the respective significance functions that we chose based on the association strength levels found in the literature:

T1 (Correlation) = 0.60. We found two main correlation strength classifications, Cohen [74] and Evan [75]. We adopt Evan's strong definition because it is stricter than Cohen's definition.

T2 (Conditional probability) = 0.60. To our knowledge, existing literature does not provide a commonly accepted and generally applicable threshold for conditional probability. Therefore, we calculated the distribution of conditional probabilities in the different versions and we selected the threshold 0.60, which on average filtered out 80% of

all data. Moreover, since this criterion is merged with the criterion conditional probability > chance agreement, we consider such threshold high enough to discriminate significant data.

T3 (Cohen's Kappa) = 0.60. Many tables of Kappa's strength of agreement can be found in the literature, the most relevant of which are [76] [77] [71] . We adopt a threshold value of 0.60. Thus, a constraint > 0.60 corresponds to a "good"/"substantial" agreement in all the proposed ranks.

Step 3: Combine Statistical Association Measures

The next step is an aggregation of the statistical association measures. For each cell of a matrix (a pair of indicators), we compute the sum over all three matrices M5, M6, and M7. The resulting matrix (displayed as M8 in Figure 24) contains the significance score for each pair and version that ranges from 0 (not significant in any of the three methods) to 3 (significant in all three methods).

Step 4: Combine Versions

The fourth and final computation step of the process is the aggregation of the significance score over versions. For every cell in Matrix M8, we compute the mean over the 13 versions of Hadoop, resulting in a single matrix (M9 in Figure 24).

Step 5: Visualize Most Significant Outcomes

In the final step, we visualize the pairs of TD indicators with most significant associations as a graph (Figure 25) and a list of TD indicators most related to interest indicators (Table 22).

RESULTS

We made the following observations from the result. First, the value of TD indicators increases together with the size of Hadoop. The sum of all TD indicators increases in the evolution from release 0.2.0 to release 0.14.0. FindBugs issues ranged from

307 to 486 but the average number of issues per class does not expose the same monotone increasing trend, and the range of the average number of issues per class is [1.30-1.76].

Code smells in the last release (352) are more than twofold the number of code smells in the first release (143), and the average of code smells per class is [0.78-1.01]. Modularity violations have the sharpest increase: they were 8 in the first analyzed release and 37 in the last one (reaching a maximum of 38 in v. 0.13.0). The average number of Modularity violations per class ranged from 0.04 to 0.11. Moreover, we detected in each version two realizations of Singleton design pattern, two realizations of State pattern and six of Abstract factory. However, none of the classes that are participating in these design patterns was affected by grime.

We do not observe a trend in any of the interest indicators. The sum of Bug fixes collected with the defect proneness strategy "inject" ranges from a minimum of 16 (v 0.8.0) to a maximum of 102 (v 0.3.0), while the range of the average number per class is [0.06-0.53]. Version 0.8.0 has no Bug fixes (fixed) and version 0.7.0 is the version with the largest number (590). The average of all classes per version is in the range [0-2.63]. The ranges of Bug fixes (between) are [162-675] and [0.55-3.01], respectively for their total by version and average of all classes by version. Finally, the Change likelihood range is [0.006-0.017] per class.

Figure 25 shows the resulting directional graph of the TD pairs and interest indicators that were on average significant in more than one statistical measure (overall mean score in Matrix M9 > 1). The nodes of the graph show the indicators. The color (or shade) of the node indicates which TD indicators were derived from the same TD detection technique. The directional edges are further labeled by their association strength (ranging from 1 to 3). And lastly, the direction indicates the conditional properties inherited from the conditional probability metric. For example, Modularity violations and Bug fixes are associated, meaning that if a class has a Modularity violation, then it is also likely that such a class will have Bug fixes. However, the reverse statement (i.e. classes containing Bug fixes are not as likely to have Modularity violations at the same time) is not necessarily true, and does not show in the graph.

Figure 25 represents the strongest associations found in our analysis. The graph contains 24 relationships (edges), one of them between TD indicators (nodes) belonging to different techniques (colors or shades), three of them between TD indicators and interest indicators or size, and the remaining 20 are among TD indicators detected by the same technique or among the interest indicators.

As for correlations between TD indicators and interest indicators, Dispersed Coupling points to classes that are more defect prone. Modularity Violations do not strongly co-occur with code smells or ASA issues but are likely to point to defect and change prone classes.

Seven out of the twelve ASA/FindBugs issue types appear in the graph. The strongest associations (average score ≥ 2) are between Style and Low and Bad Practice and Medium. We also observe an association between a FindBugs issue (High) and a Code Smell (Intensive Coupling). Four out of ten code smells show up in the graph. Brain Class and Brain Method code smells are related in both directions, as well as Dispersed and Intensive Coupling (but only one direction).

Lastly, defect prone classes tend to be also change prone, and vice-versa. The size metric Number of Methods does not shoot up in the graph indicating that neither the TD indicators nor the interest indicators are very strongly associated with size.

RQ 1: Which techniques tend to report problems in the same sets of classes?

The results shown in Figure 25 lead to our first finding in response to RQ1: *Different TD techniques point to different classes and therefore to different problems.*

The only arc in Figure 25 that relates two different types of TD identification approaches is Intensive Coupling and FindBugs High priority issues. A method exhibits intensive coupling if it "calls too many methods from a few unrelated classes" [67]. FindBugs High priority issues are those issues thought to have higher probability to detect serious problems in the code. The direction of the association indicates that classes with many High priority issues have methods affected by Intensive Coupling. A possible explanation for this relation is that both of these indicators point, more than any others, to generally poorly designed code.

Looking at the associations inside the boundaries of the techniques, we observe a characteristic of all FindBugs issues in the graph: significant relationships are only revealed between priority and type categories, which are not independent indicators and are constructed by the FindBugs authors[12]. Therefore this relationship is not a surprising result. A follow up analysis revealed that 81-87% (depending on version) of all classes contain FindBugs issues of only one single category.

Shifting the focus of the results analysis to the code smells group, we observe three associations between particular code smells: Brain Class → Brain Method (2.0), Brain Method → Brain Class (1.15) and Dispersed Coupling → Intensive Coupling (1.23). The first relationship is stronger in the direction → Brain Method and it indicates that Brain classes are more prone to contain Brain methods. This observation can be explained by the way Brain Class code smells are detected using Marinescu's detection strategy [67] [68]: the Brain Class detection requires that the inspected class contains at least one Brain Method. Therefore the conditional probability as defined in Section IV of P(Brain Method | Brain Class) is always 1.0. The second relationship between code smells is Dispersed Coupling → Intensive Coupling (1.20). While the latter indicates that a class has methods that invoke many functions of a few other classes, the former shows classes having methods invoking functions of many other classes. Their association demonstrates that classes in Hadoop having the former of the coupling smells also have the latter smell, which intensifies the problem of coupling. No other relationship within different code smells was found in this analysis.

We also observe that modularity violations are not strongly related to any other indicator. This confirms and validates one of the findings reported in Wong et al [55], who found that 40% of modularity violations in Hadoop are not defined as code smells and are not detectable using existing approaches.

To conclude, the 4 TD detection approaches (modularity violation, code smells, grime, and ASA issues) have only very little overlap and are therefore pointing to different problems. Within the broad approaches, relations are stronger (as one would expect).

[12] Each bug pattern is assigned a priority and category by the FindBugs authors. Some categories are biased towards single priorities: e.g., correctness is considered more often to be of high priority.

However the data also shows that many code smells and some ASA issue types are not inter-related (i.e. the ones not showing in Figure 25) indicating that even at a lower level indicators point to different problem classes.

RQ2: Which TD indicators correlate with the interest indicators defect- and change-proneness?

Turning to RQ2, our major finding concerning defect-proneness is summarized as follows: The dispersed coupling code smell and modularity violations are located in the classes that are more defect-prone.

For each TD indicator (Column 1), Table 22 reports the average score obtained in matrix M9 for the association between the indicator and defect proneness (columns 2-4) and change proneness (column 5). TD indicators are listed in the same order as Table 21, but those with average score less than or equal to 0.3 in all associations with interest indicators are not shown.

Figure 25 shows that no single FindBugs indicator has a very strong relationship (>1) with Bug fixes. However, when investigating less correlated indicators we find the strongest FindBugs indicator to be Multithread Correctness having a borderline value of 1.0 (Table II). This category is very specific but ties very well into the studied application; Hadoop has to deal with both distributed data storage and computations. Previous work [17] [33] [38] reported that only a small percentage of FindBugs issues are actually related to bug fixes. This is supported by our results and a follow up analysis: Multithread Correctness issues make up only 5.3% of the total of FindBugs issues found in Hadoop.

Another strong relationship with Bug fixes (in two approaches, between and fixed) involves the Dispersed Coupling code smell. In a related work [78], Dispersed Coupling was highly correlated with bug fixes only when the prevalence of this smell increased during the evolution of the software. We observe a border value (1.0) also for one other code smell: the God Class indicator has an average score of 1.0. Zazworka et al. [69] reported in their previous work that in an industrial system god classes contained up to 13 times more defects than non-god classes.

The last indicator strongly related to Bug fixes (between) is Modularity violations, which are located in the same classes where the more bug fixes are found (but not in all of them).

Although defect prediction is not a goal of this work, it is useful to look at measures of precision and recall to further describe the relationships we've found. We used the two TD indicators most strongly related to bug fixes and the two border value indicators to predict, in each version, classes with at least one bug fix (strategy between). We observe in Table III high precision and low recall values. Each of the four indicators points out a small subset of defect-prone classes very well. When using all four indicators together recall can be raised to 0.33 by trading off some precision.

The second part of RQ2 was concerned with change proneness. The following summarizes this result. Modularity violations point to change prone classes.

Change-prone classes might indicate maintenance problems (e.g., classes that have to be changed unusually often are candidates for refactoring). We labeled on average the top 25% most frequently changed classes as "change-prone."

The results indicate that Modularity Violations are strongly related to change likelihood. Table 22 shows that the highest average scores are Modularity violations (1.38), Dispersed coupling (0.92) and God Classes (0.85). This fits expectations since all of the three approaches claim to identify maintenance problems. Modularity violations and Dispersed coupling point to classes that have collaboration disharmonies. The God class code smell identifies classes that implement multiple responsibilities and should be refactored (e.g. split up into multiple classes).

Further, the relation between defect and change proneness shows that these issues are interconnected. Explanations for the phenomena can be that maintenance problems lead to less correct code, or that many quick-and-dirty bug fixes lead to less maintainable code.

Finally, we point out that a large set of TD indicators (i.e. 9 out of 25) do not show significant associations with defect or change proneness. This proportion suggests that these indicators either point to different classes of quality issues (e.g. FindBugs type Performance) or to none at all. Therefore these results can be further used to tailor TD indicators towards quality attributes of interest. If one is most interested in defect and

change proneness issues in Hadoop (or similar software) we suggest analyzing for dispersed coupling and modularity violations.

Table 22. Association of TD indicators with interest indicators

	TD Indicator	Bug fixes (between)	Bug fixes (inject)	Bug fixes (fixed)	Change likeli-hood
Mod Viol	*Modularity violations*	**1.23**	0.23	0.54	**1.38**
Code Smells	*God Class*	**1.00**	0.23	0.77	0.85
	Brain Class	0.62	0.23	0.46	0.62
	Tradition Breaker	0.69	0.31	0.38	0.69
	Feature Envy	0.54	0.15	0.31	0.15
	Brain Method	0.77	0.23	0.54	0.46
	Intensive Coupling	0.54	0.00	0.08	0.15
	Dispersed Coupling	**1.31**	0.23	0.54	0.92
	Shotgun Surgery	0.31	0.00	0.08	0.08
FindBugs issues	*High*	0.92	0.08	0.46	0.62
	MT Correctness	**1.00**	0.08	0.46	0.69
	Correctness	0.92	0.15	0.46	0.62
	Performance	0.31	0.00	0.08	0.08
	Style	0.31	0.00	0.08	0.38
Size	*Number of Methods*	0.62	0.00	0.08	0.08

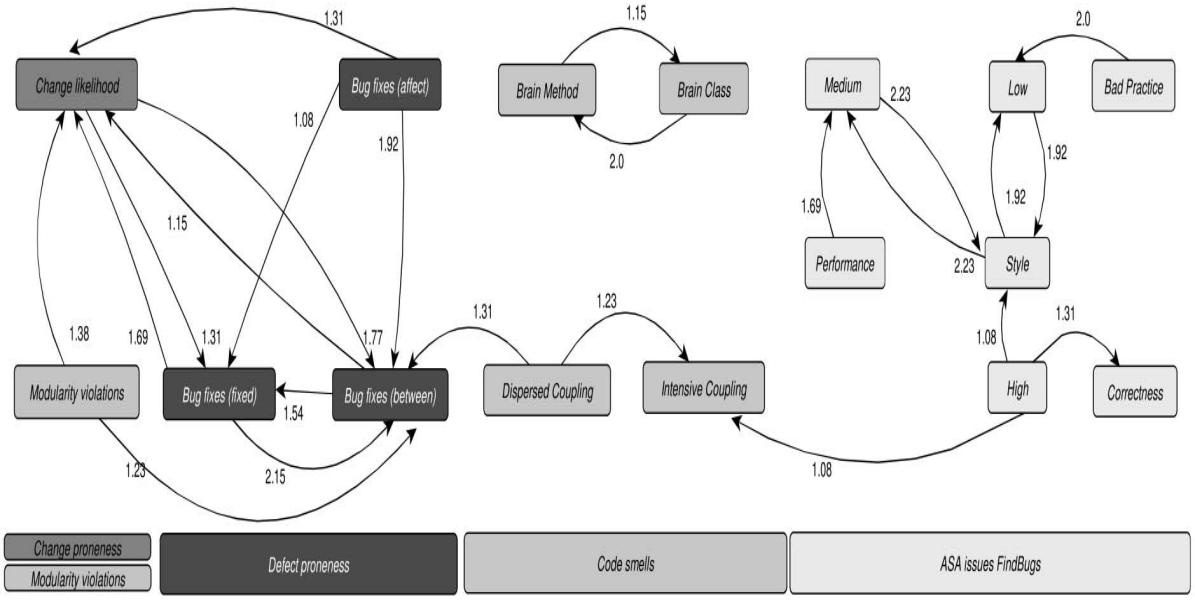

Figure 25. Graph of top ranked pairs (average score > 1)

THREATS TO VALIDITY

We list the threats to the validity and generalizability of the study following the structure proposed by Wohlin et al. [7], who identify four categories: construct, internal, conclusion and external threats.

A first conclusion threat concerns the impact of thresholds T1, T2 and T3 applied on Step 2 on the results. We documented the choice of thresholds based on values discussed and adopted in the literature, and we adopted higher values to decrease the level of uncertainty. The collection and aggregation of different statistical measures also lowers the risk associated with this threat.

A further statistical point of discussion is the loss of measurement resolution in data transformation from ranges [0..N] to the range [0,1]. The transformation was required

to compute Cohen's Kappa and the conditional probability. To limit this threat we investigated distributions carefully to find reasonable thresholds (e.g. for Number of Methods we decide on a threshold of 11 for a top 25% cutoff). Moreover, the choice of at least one occurrence as criterion for transformation was driven by a preliminary analysis of distribution that revealed that TD and interest indicators were equal to zero on average in 90% of the classes.

Another conclusion threat is derived by the decision to not normalize measures by size. Our choice was based on past experiences [69] and from the analysis of the results of the current study. shows a correlation plot between size and number of defect fixes for each file over all versions. The 3rd order polynomial trend line shows that the correlation is not simply linear, e.g. a class twice as large is not twice as defect-prone. This analysis suggests that a linear normalization by number of methods is not required.

Moreover, the choice of this size metric rather than other size metrics is also a threat (construct). We examined whether the Total Number of Methods correlates with other size metrics in the different Hadoop versions. We obtained almost perfect correlations with Total Number of Statements, Total Number of Lines of Code (that include comments) and Total Number of Files: 0.9958, 0.9950 and 0.9711 respectively. In addition to that, we

checked, for each version, the correlation between the Number of Methods and two other metrics, i.e. the Number of Lines of Code and the Number of Statements in Java classes. We also obtained at this granularity very high correlation, respectively 0.8975 and 0.8780. We conclude that using Number of Methods as measure of class size is equivalent to lines of code and did not affect results.

A further construct threat is the selection of outer classes, ignoring nested classes. At this point in time, the tools used did not allow the collection of all metrics for nested classes. Therefore the validity scope of our results is limited to outer classes only.

We believe that the findings of this work apply only in the context analyzed (external threat). They may apply in similar applications, but we are not aware of any other published results that can be compared to ours. However, although results cannot be generalized, they contribute to begin composing the TD landscape.

3.3. A CASE STUDY OF EFFECTIVELY IDENTIFYING TECHNICAL DEBT

The TD metaphor has facilitated discussion among practitioners and researchers by providing a familiar framework and vocabulary from the financial domain. Since the metaphor appears to be sound and intuitively understandable by all stakeholders involved in the software development process, it has potential to become a truly universal language for communicating technical tradeoffs.

Our vision, however, goes beyond facilitating communication, to the development of a set of such tools, inspired by the technical debt metaphor, which stakeholders can use in today's software projects. Identification is essential to transform the technical debt in a project into a manageable body that would allow one, such as with a portfolio of financial debt, to better control the current debt situation. Technical debt identification approaches broadly consist of methods to elicit technical debt instances from humans (i.e. developers and other stakeholders), and methods that rely on automated tools of various kinds to detect potential debt in the source code. Human, manual approaches are likely to be more time-consuming, but have two advantages (at least in theory) over automated approaches. One is that they might be more accurate, i.e. more likely to identify technical debt that is most significant, while automated analyses may reveal many anomalies that turn out to be unimportant. The other advantage is that human stakeholders might be able to provide additional important contextual information related to each instance of technical debt (e.g. effort estimates, impact, decision rationale, etc.) that is difficult or even impossible to glean from analysis tools.

The first contribution of our work is the evaluation of human elicitation of technical debt. We propose and evaluate a technical debt backlog [79] that can be used to capture, store, and communicate essential properties of technical debt that can feed into further decision making processes about debt repayment. Besides the template, our case study gives some insight into the dynamics of eliciting technical debt from a team of developers, all familiar with different aspects of the system being analyzed.

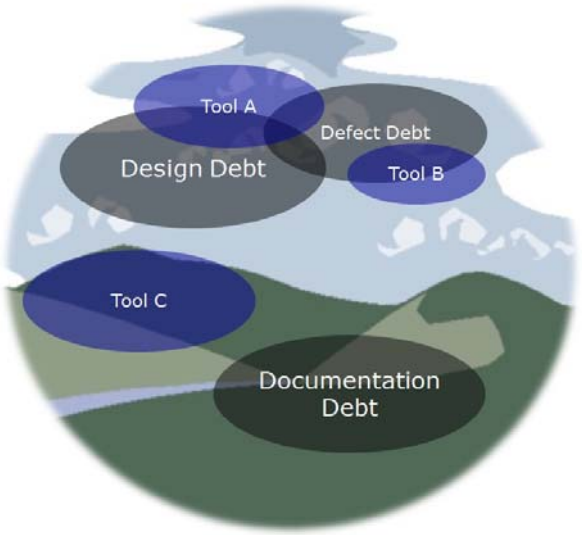

Figure 27. The Technical debt Landscape

As a second contribution we evaluate the utility of tool support for technical debt identification. Some recent research has addressed the issue of how close automated approaches can get to the accuracy and informativeness of manual technical debt identification. Some of these studies have indicated that it is possible to identify certain classes of potential technical debt (in particular design debt) with computer-assisted methods [54] [80]. Moreover, they have demonstrated that detection approaches can succeed in finding issues that are of value to developers [54]. However, despite the fact that these approaches point to system code fragments that need improvement, it is not clear yet if they point to the most important technical debt, from software project stakeholders' point of view. Based on the current state-of-the-art, we studied three automated approaches (code smells, automated static analysis issues, and collection of code metrics), and how their output compares to technical debt that is elicited from humans. This understanding can help address questions such about how tools can best be used, instead of or in addition to manual approaches, in the identification of technical debt.

This study, and others like it, will help evolve the technical debt landscape, as conceptually outlined in Figure 27 [52]. The landscape lays out the different types and flavors of technical debt that exist in real software projects with respect to their importance and overlaps, and how those types are best identified by tools and other methods. Such an evolved landscape is essential to achieve our vision of building an effective toolkit for identifying, quantifying, and managing technical debt.

BACKGROUND AND RELATED WORK

According to Seaman and Guo [79], the management of technical debt can center on a technical debt list, which is similar to a task backlog. The backlog contains technical debt items (in the following simply referred to as items), each of which represents a task that was left undone, but that runs a risk of causing future problems if not completed. Each item includes a description of what part of the system the debt item is related to, why that task needs to be done, and estimates of the technical debt's principal and interest, as well as some other attributes, as shown in Table 23.

The principal refers to the cost to fully eliminate the debt, i.e. to completely repair the technical imperfection. Depending on the type of technical debt this can translate into different kind of activities, such as, adding missing documentation, refactor code that is hard to maintain, or maintaining a set of regression tests to align with the code and requirements. The cost of techincal debt repair might be in some cases understood better than in others. For example, adding missing documentation might be more straight forward to estimate than a more complex code refactoring. Seaman and Guo propose to initially estimate the principal on a rough ordinal scale from low to medium to high, that allows to some extend to understand effort and can contribute in iteration planning. To further help estimating principal, historical effort data can be used to make a more accurate and reliable estimation beyond the initial high/medium/low assessment. For example, if a debt item is a set of classes that need to be refactored, the historical cost of modification of those classes can be used as the future modification cost (principal of the debt item) estimation.

Table 23. The Technical Debt BackLog

ID	*Technical debt identification number*
Responsible	*Person or role who should fix this TD item*
Type	*design, documentation, defect, testing, or other type of debt*
Location	*List of files/classes/methods or documents/pages involved*
Description	*Describes the anomaly and possible impacts on future maintenance*
Estimated principal	*How much work is required to pay off this TD item on a three point scale:* ***High/Medium/Low***
Estimated interest amount	*How much extra work will need to be performed in the future if this TD item is not paid off now on a three point scale:* ***High/Medium/Low***
Estimated interest probability	*How likely is it that this item, if not paid off, will cause extra work to be necessary in the future on a three point scale:* ***High/Medium/Low***
Intentional?	***Yes/No/Don't Know***

The second main compound of technical debt is interest, which is composed of two parts: (1) the interest probability is the probability that the debt, if not repaid, will make other work more expensive over a given period of time or a release; (2) the interest amount is an estimate of the amount of extra work that will be needed if this debt item is not repaid.

Interest probability can be estimated using historical data like usage and defect. In addition, it is also important to consider time variable because probability varies over the time. For example, the probability that a particular module contains hidden defects can be estimated based on the past defect profile of that module. If that module will be deployed soon to the user, the interest probability will be higher because the chance of those defects be identified is bigger.

Finally, interest amount can also be estimated using historical data. In addition to the financial properties of technical debt, several properties that support decisions on repayment are worth capturing:

1. The type of debt can be helpful to tailor debt payment to project critical quality characteristics. For example, known defect debt will be differently perceived in life critical software applications. Currently known types, besides latent defect debt, are: design debt (an imperfection of the software's design or architecture negatively affecting future maintenance), documentation debt (missing, outdated, or incomplete documentation), and testing debt (missing test cases, test cases that are not executed, or missing test plans). Studies like these will contribute to a more complete set of types of debt.
2. Was the original decision to go into debt made intentional or unintentional? This information can help to understand how explicit debt and technical debt decisions are managed in a project. In an ideal case any decision to incur debt is made intentionally to reduce the risk of surprises caused by unintentional, potentially even unknown debt.
3. Who is responsible for fixing the technical debt. This information is important to understand the basis on which principal and interest was assessed. For example, the effort involved in eliminating debt might depend on the developer who originally designed a piece of code.
4. Where is the technical debt located? This information is important to understand impact on product, relationships between items, and ripple effects in source code when repaying the debt. For example, debt might be cheap to eliminate but resist in parts of a software system that are risky to modify. Or, several items might point to the same parts of code and fixing this code might eliminate multiple items at the same time.

The process of managing technical debt using this approach starts with detecting technical debt items to construct the technical debt list. The next step is to measure the debt

items on the list by estimating the principal, interest amount and interest probability. Then the debt items are monitored and decisions can be made on when and what debt items should be paid or deferred.

In this work, we are focused on the first step of this process: TD Identification. We can use different strategies to find TD items for each TD type. Two automated strategies that have been proposed to support the identification of technical debt in software projects are identification of code smells and issues raised by automatic static code analysis tools, aka ASA issues.

Past studies have shown that some code smells are correlated with defect- and change-proneness [69]. In this study, we use CodeVizard [81] to detect a set of 10 code smells as proposed in [67].

As far ASA, also for this study we selected FindBugs.

In addition to code smells and issues, in this study we are also interested in collecting basic structural code metrics for size and complexity to study whether any relationship with TD items exists.

CONTEXT OF THE STUDY

The study was conducted at Kali Software, a small software development company located in Rio de Janeiro, Brazil, that develops primarily web applications written in Java and based on the MVC framework. The project we studied consisted of a small application over 25K non-commented lines of code. It is a database-driven web application for the sea transportation domain. It has undergone a full product lifecycle (elicitation, design, implementation, deployment, and maintenance). The project team is composed of five professionals: two developers, one maintainer, one tester, and one project manager who also plays the role of the requirements analyst.

GOAL AND RESEARCH QUESTIONS

The goal of the research is to evaluate the human elicitation of technical debt ad how it relates with the output of tools for technical debt identification.

The study's research questions are:

(RQ1) Do tools output correspond to technical debt from a developer perspective?

(RQ2) How much do different developers technical debt items overlap?

(RQ3) How hard is the technical debt item report template to fill in?

Given the exploratory nature of this case study and the small amount of TD items collected (21), we preferred to answer the research questions in a qualitative way rather than building hypotheses and test them.

PROCEDURE AND DATA COLLECTION

The study has been implemented with two phases: (1) Training on Technical Debt and (2) Collection of Technical Debt Items.

Phase 1: Training on Technical Debt

For the first phase, the development team has been trained on technical debt theory using a PowerPoint presentation followed by an opportunity for Q&A. Given to practical constraints, the presentation was done via Skype. All the material was in Portuguese since this is the natural language of the development team.

During the training, only abstract technical debt items have been used as examples (for instance: "technical debt items on" house repair or car repair) to avoid bias on identifying technical debt items during the second phase of the study.

Phase 2: Collect Technical Debt Items

The second phase was composed of two parallel activities: manual and automatic TD identification, i.e. collecting TD items from the development team and collecting the output of tools analysis on the source code.

For the manual identification of TD, the development team (project manager, developers, and testers) has been asked to report technical debt items individually. For this, we provided the team components with a short questionnaire to both report the TD items through the Technical Debt back Log (question 1) and provide information about the difficulty of documenting debt items (questions from 2 to 5). The respondents were asked to document up to five of the most pressing technical debt items they knew of in the current version of the software.

The questions are the following ones:

1. If you were given a week to work on this application, and were told not to add any new features or fix any bugs, but only to address Technical Debt (i.e. make it more maintainable for the future), what would you spend your time on?
2. How difficult was it to identify TD items?
3. How difficult was it to report TD items (i.e. fill in the template)?
4. How much effort did you need to identify and document all the TD items?
5. Which are the most difficult fields to fill in / which are the least difficult ones?

All answers were given as free text.

In parallel to the questionnaire, we applied the CodeVizard and FindBugs tools to the latest version of the subject project source code, in order to identify code smells and ASA issues. The data described, for each file (i.e. class) in the code base, how many of each type of code smell were identified, and how many of each type of ASA issue were present. Each FindBugs issue has a category, (e.g., Performance, Correctness, etc.), and a priority from 1 (highest) to 3 (lowest).

As a result of this analysis, a list of CV and FindBugs results that agree and that disagree (are not mentioned) with the survey results is created and analyzed.

Regarding the structural metrics, we selected and computed for each file the following ones: Lines of Code, McCabe's Cyclomatic Complexity, Density of Comments, and Sum of Maximum Nesting of all Methods in a Class.

Lines of Code and McCabe's Cyclomatic Complexity are widely used in the literature of defect and maintainability prediction (e.g., [46] , [59]). Density of Comments was selected to study whether highly commented code might have a relationship with technical debt, while Max Nesting measures complexity in depth (the higher is the nesting, the more complex is the code).

The metrics were computed with ad-hoc scripts/tools.

RESULTS

Results in Figure 28 show how the 21 technical debt items identified by the software team, each represented as a colored box, were distributed over project roles and types of debt.

As the legend indicates, each box has three faces, corresponding to principal (front), interest probability (right side) and interest amount (up). Each face can be green, yellow or red with respect to the estimation of the team member (respectively low, medium and high). An "i" on the front face indicates whether the debt was intentionally introduced or not.

Figure 29 shows the results of automated identification approaches (FindBugs, Code Smells, Metrics) compared to the items reported by the development team. The boxes refer to the same elicited technical debt items as are shown in Figure 28. An S on the front face shows which technical debt items were given a location by the subjects that pointed to source code.

For each automatic approach, we pre-filtered the results and we choose only the best predictors of technical debt: FindBugs Priority 1 issues (highest priority), MAX

nesting for metrics, and Intensive Coupling for code smells. For every indicator, the top 10% cut off is considered in Figure 29. The top 10% cutoff value means that a human would have to inspect only the top 10% of all source code files with respect to the tool output (in this project 30 out of 303 files) to catch all files associated with technical debt.

The answers to the research questions follow.

RQ 1-Do tools output correspond to technical debt from a developer perspective?

We observe in Figure 29 that the three automated approaches (code smells, ASA, and code metrics) do about equally well in identifying defect debt. By selecting the three best predictors for defect debt as shown in Figure 29 (i.e., Max Nesting, Intensive Coupling, and High Priority FindBugs issues), and looking at the top 10% of files for each of the predictors, permitted to capture all affected source code components.

On the contrary, only five out of twelve reported items of type documentation debt, testing debt, and usability debt were related to source code files, the remaining seven were related to artifacts (e.g. requirements documents and test plans) other than source code. The five TD items located on source code are design debt, one is located to a source file identified by all automatic approaches, and the last one only to Intensive Coupling top 10% and FindBugs Priority 1 top 10%.

For design debt, automated approaches capture about half of the technical debt items, although ASA issues and code smells identify more than traditional code metrics.

Summarizing, these results lead to answer RQ1 in the following way: tools can support the identification of defect debt in this project, but not other types of debt that were found by developers.

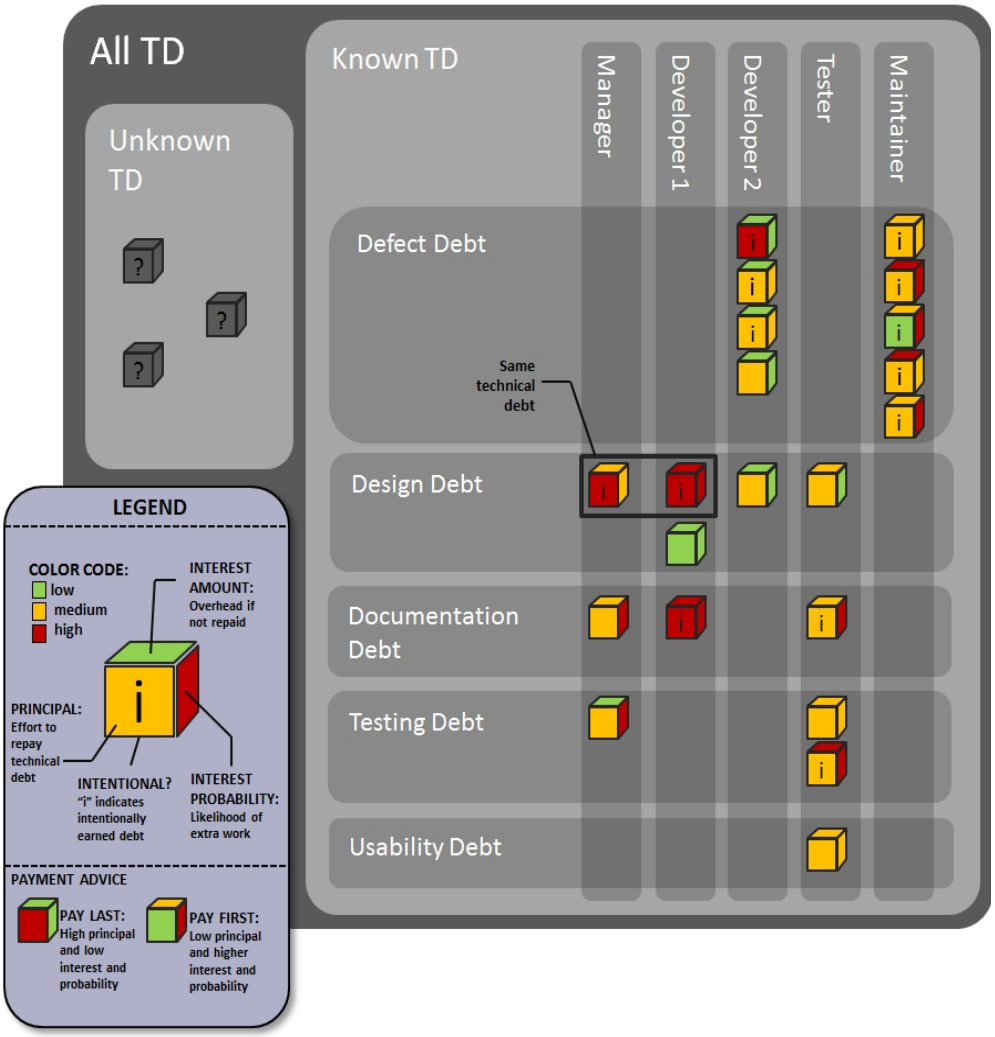

Figure 28. Results of the human elicitation of TD items

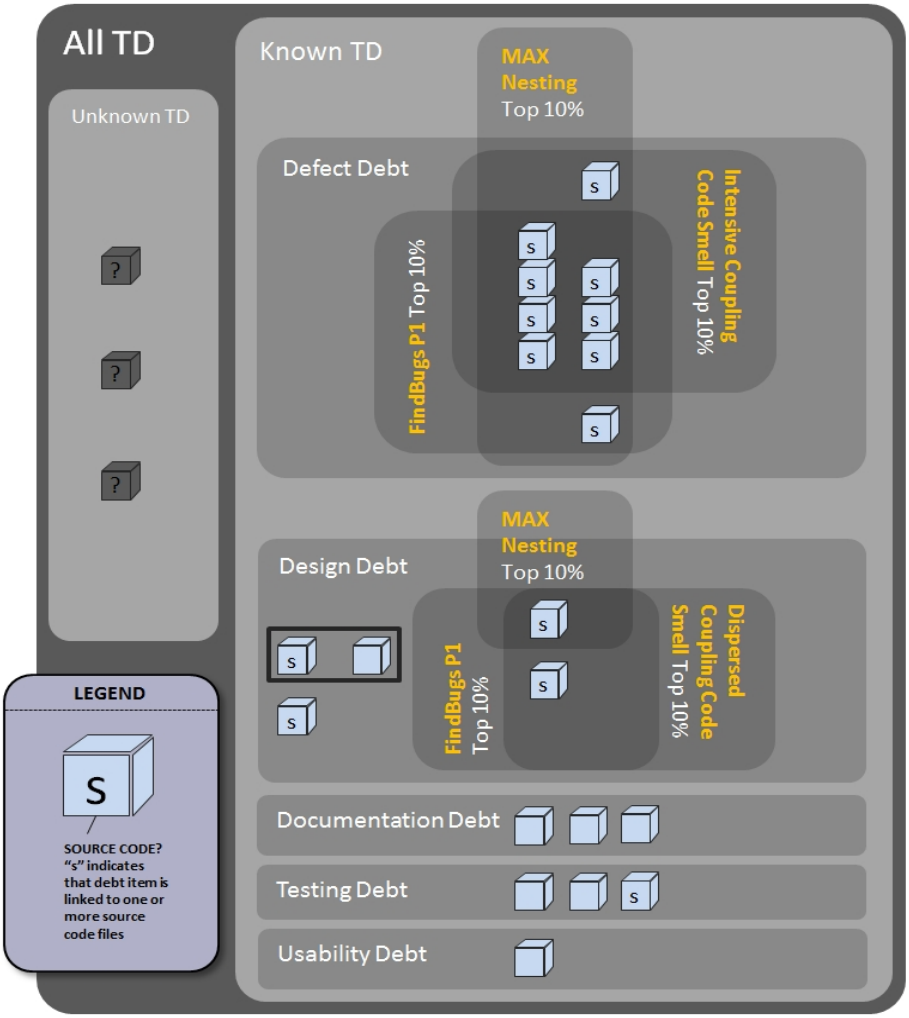

Figure 29. Results of the tools compared to human elicitation.

RQ2-How much do different developers technical debt items overlap?

Only one technical debt item was reported by two different stakeholders (the manager and one developer). None of the remaining 19 items were reported by more than one stakeholder. This result indicates that, in this project, technical debt knowledge is dispersed and perceived differently by different stakeholders. The software tester reported the widest range of technical debt types including one previously unknown type, usability debt, which in this case had to do with the lack of a common user interface template.

RQ 3-How hard is the technical debt item report template to fill in?

The five subjects of the study reported that it took between 50 minutes and 2 hours to identify and document the technical debt items (average of 19 minutes per item). Answers about difficulty of the task ranged from "easy" to "difficult/high" (all answers were given as free text). Subjects agreed that the fields principal, interest amount, and interest probability were the most difficult to fill in, although we did not ask for estimates beyond high, medium, and low. On the contrary, location, type, and responsible were commonly noted as the least difficult fields. These results indicate that, in this project, the initial elicitation of technical debt items could be done in reasonable time, but that the key financial parameters of technical debt were difficult to estimate and might require better process or tool support in future.

DISCUSSION

Results of research questions showed that in the project studied:

- the tools used could properly identify files affected by defect debt but no other types of debt identified by developers in other artifacts;
- different stakeholders identified different debt;
- the initial elicitation of technical debt items could be done in reasonable time, but that the key financial parameters of technical debt were difficult to estimate;

Some additional observations, beyond or complementary to the scope of the research questions, are possible from the data collected.

The first observation is that the majority of items reported falls into the defect debt category. This indicates that known defects are of concern to the development team of this project.

Secondly, the colors of the TD items, indicating principal, interest amount, and interest probability are rather equally distributed among the different type of debts. This suggests that debt characteristics are not tied to the type of debt, i.e. no type of debt has noticeably higher overall interest or principal.

Thirdly, we find intentionally earned debt in almost every category, except usability. This is especially interesting for defect debt. Many of the defect debt items were requirements that were not fully implemented. The intentionality of these items indicates that a decision was made to not fully implement those requirements, most likely due to time constraints, which makes these instances conceptually different from defects caused by unintentional programming mistakes.

Further, many TD items could not have been found by the tools or metrics since the artifacts they were located are not included in the static code analysis. This suggests that a focus on source code as the single source of technical debt is too narrow, as developers reported a significant number of such items among their most important. Future studies might consider including or proposing tools for other kinds of development artifacts

affected by technical debt. Finally, we think that the figure and color coding hints at how this information can be further used to manage debt and make clearer decisions on which debt to pay. For example, items that have generally a low principal (e.g. green principal), but yellow or red interest characteristics are good candidates for paying off first, since their return on investment is more favorable than for other items. This idea of a cost/benefit decision approach has been previously proposed and discussed in [82].

THREATS TO VALIDITY

As with any case study, especially of a small project such as this one, threats to external validity are significant. We accept these threats, and attempt to trade off breadth for depth, by doing a thorough analysis of a small case, yielding deeper insights that would not have been possible in a much larger sample.

In Figure 29, we chose the code smell, ASA category, and code metric that were the best predictors of technical debt. In practice, however, these choices cannot be made a priori. Our motivation for this approach was to determine simply if any of the automated approaches were related to the technical debt elicited from developers. Further work is needed to determine whether the choices we have made hold in all situations.

An important construct threat derives from the following assumption made in the design of this study: we assumed that the perceptions of software developers about the technical debt in their code can serve as a "ground truth" against which other types of technical debt identification can usefully be compared. However, the ultimate and authoritative "ground truth" for studies of technical debt would be a measure based on future maintenance effort associated with technical debt items. That is, a "real" technical debt item is one that leads to higher maintenance effort than would have been incurred if the debt did not exist. However, measuring "real" technical debt in this way was not possible in this study, nor is it in many studies. For the study in this paper, the assumption we have made represents a threat to construct validity in the sense that the technical debt reported by the developers might not lead to future increases in maintenance cost.

CONCLUSIONS AND CONTRIBUTIONS

We have conducted two empirical studies to understand how ASA impact maintainability in terms of Technical Debt.

We conducted an inductive study within a Brazilian software company, in which we compared ASA and Code Smells with the manual elicitation of Technical Debt.

We have presented and evaluated how the technical debt backlog can be populated by developers through a common technical debt template, and how existing tool approaches can help to identify certain types of debt. We have further shown that different stakeholders know about different debt in their project, indicating that technical debt elicitation should include a range of project team members. *Aggregation*, not consensus, would appear to be the most effective approach to combining the input from different team members. In addition, three different automated approaches - code smells, ASA issues, and traditional code metrics - did well in pointing to source code files with defect debt, and also could point to a partial set of files with design debt.

We encourage practitioners to use the proposed template in their projects and to share results and experiences (e.g. at www.technicaldebt.umbc.edu). It will require evidence from a variety of environments to build a full picture of how different technical debt identification approaches interact, overlap, and are (or are not) synergistic. This evidence is necessary to further refine and to bring into focus the technical debt landscape.

A first step towards defining the TD landscape was done in the analysis of 13 versions of the Hadoop projects, in which we combined four different TD identification techniques: code smells, ASA, modularity violations and grime. We computed the association of these indicators with two TD interest indicators: change and defect proneness.

The main findings of this study are:

- Different TD techniques point to different classes and therefore to different problems. There are very few overlaps among the results reported by these techniques.
- Dispersed coupling, god classes, modularity violations and multithread correctness issues are located in classes with higher defect-proneness.
- Modularity violations are strongly associated with change proneness.

Our results indicate that the issues raised by the different code analysis techniques are in different software classes. Moreover, only a subset of the problematic issue types identified by these techniques is shown to be more defective or change prone. This is consistent with the result of earlier work where these techniques were applied independently ([33] [38] [69] [70]).

These findings contribute to building an initial picture of the TD landscape, where TD techniques are loosely overlapping and only a subset of them is strongly associated to software components' defect and change proneness.

Implications for Practice

In practice, results indicate that multiple technical debt indicators should be used instead of only one of the investigated tools. As recommendation to practitioners, these initial results evidently show that different tools point to different problems in a code base. The use of a single tool or single indicator (e.g. a single code smells) will only in rare cases point all project important technical debt issues. It also shows that within the set of selected approaches none supersede another, making none of them dispensable. In the current state of research we cannot yet give a more complete recommendation on which indicators are best for signalling specific quality shortcomings, however, our results give some preliminary advice on which indicators to start with when looking for TD related to defects and maintenance bottlenecks, namely: Modularity Violations, God Class, Dispersed Coupling, and MT Correctness issues.

Implications for Research

Since results indicate that there might not be a project independent one-size-fits-all tool to detect technical debt, but rather a tailoring process to the right subset of indicators required, future research should be concerned about investigating and showing connections between TD techniques, types of technical debt, effect and tailoring towards project specific software quality characteristics. Future work should also investigate other TD indicators when they become available to broaden the landscape.

As more specific advice for future research directions, we recommend to extend the interest indicators towards a broader range of software quality aspects, besides defect and change-proneness as investigated here. Further, we recommend to extend this type of quantitative study with qualitative insights, e.g. from practitioners that investigate if the studied approaches point to the most important kinds of technical debt.

4 PERFORMANCE EFFICIENCY

4.1. DEFINITIONS

Performance efficiency is “the performance relative to the amount of resources used under stated conditions. Resources can include other software products, the software and hardware configuration of the system, and materials (e.g. print paper, storage media)”.

Performance efficiency is composed of two sub-characteristics:

- Time behaviour: “degree to which the response and processing times and throughput rates of a product or system, when performing its functions, meet requirements”
- Resource utilization: “degree to which the amounts and types of resources used by a product or system, when performing its functions, meet requirements”.

We focused our analysis on Time behaviour, and we assessed whether refactoring the code to remove ASA issues improve the processing times of software functions.

We performed two studies: a pilot study in laboratory settings, and an experiment in industrial settings.

4.2. QUANTITATIVE ASSESSMENT OF THE IMPACT OF AUTOMATIC STATIC ANALYSIS ISSUES ON TIME EFFICIENCY: A PILOT STUDY

As we have seen in Section 4.1, efficiency, in terms of time behaviour, is measured by computation or approximation of the execution time of the software function under study. A fundamental concept in this computation is that the execution time is not deterministic, but it has a certain variation depending on the input data or on different contexts of the platform in which it is executed. For this reason, for a given code there is a best-case execution time (BCET), i.e. the shortest possible execution time, and a worst-case execution time (WCET), i.e. the longest possible execution time. Whilhelm et al. [83] identified in literature and industrial practices two approaches to determine the BCET and the WCET. The first approach is characterized by static methods: the code and the possible paths are analyzed and, combining different techniques, upper and lower bounds for the execution time are provided. This methodology does not take into account the hardware and the environment on which the code is executed, hence the bounds overestimate the WCET and underestimate the BCET. The second methodology is measurement-based: the code, or a portion of it, is executed on a given hardware or a simulator for a set of inputs.

Then, WCET and BCET are obtained from observation: these methods provide estimates and not bounds, and they usually underestimate the WCET and overestimate the BCET. Despite the high number of techniques developed for both approaches, the problem for WCET analysis in the field of Java applications has not been deeply examined yet: Harmon and Klefstad conducted a survey of WCET analysis for Real- Time Java [84], but they were able to find fewer than twenty publications addressing the problem. Source code annotations as instructions to WCET tools ([85] [86]), low-level and high level analysis of bytecode ([87] [88]) and Java-native processors ([89]) are the most common solutions proposed, however it is very difficult to find a methodology to obtain precise and generalizable bounds for the WCET and the BCET in Java: the main motivation is the overhead and the variability introduced at execution time by VM services (e.g. automatic

memory management)[90] . However, since the object of our experiment is very simple code, makes measurement simpler for us. For instance, we test code with only one possible path, thus we can exclude all static measurement-based approaches.

Moreover, we are interested in the comparison of execution times: for this reason, we think it is considerable to abandon the usual concepts of WCET and BCET and adopt average values and confidence intervals. We focused on a specific ASA tool that is FindBugs v1.3.9. FindBugs uses analyzers called Bug Detectors to search for simple bug patterns. These bug detectors contain numerous heuristics to filter out or deprioritize warnings that may be inaccurate, or that may not represent serious problems in practice. FindBugs warnings are organized in 369 issues or bug patterns[13] , grouped subsequently into Categories (Correctness, Performance, Security, etc) and priorities (high, medium, or low), based to the severity of the problem detected. Both categories and priorities are assigned by tool's authors, based on their wisdom and experience reviewing warnings in industrial and university contexts. A subset of FindBugs patterns is part of category Performance: they are supposed by tools' authors to have negative impact on Performance, i.e. the efficiency of the code.

GOAL DEFINITION

Let us consider a simple code fragment:

```
Collection<Integer> col = new LinkedList<Integer>();
...
col.removeAll(col);
```

If FindBugs were run on the above code it would signal the issue DMI USING REMOVEALL TO CLEAR COLLECTION. The description of the issues provided by the tool is: “If you want to remove all elements from a collection *c*, use `c.clear()` , not `c.removeAll(c)`”. In response to such a notification from the tool the developer should

[13] The full list of patterns is available at http://findbugs.sourceforge.net/bugDescriptions.html

refactor the code according to what is suggested by the issue description, i.e. replacing `col.removeAll(col)` with `col.clear()` . The research question we would like to answer with this study is whether the issue represents an actual threat to time efficiency. In other words: does refactoring out the issue yields a code that exhibits an improved time efficiency?

We define our experiment following the GQM template [34].

Object of study. The object of the study is represented by the issues signaled by FindBugs.

Purpose. The purpose of the experiment is to identify those issues that impact time efficiency and quantitatively assess the delay introduced. Having experimental evidence of this impact, programmers can be sure that if they refactor the code deleting these issues, the system will be faster.

Perspective. The perspective is from the point of view of programmers of Java applications that take care of performance issues in delivering their software.

Quality focus. Efficiency is the quality characteristic that we address. We focus our experiment on time behavior, i.e. the amount of time to perform one or more operations.

Context. The context is artificially developed code. We conduct our tests on code developed ad hoc to violate the issues that might impact negatively the efficiency-time behavior of the code.

Summarizing, our goal is defined by the following scheme:

Evaluate a subset of FindBugs issues
for the purpose of assessing the actual impact
compared to their removal by refactoring
with respect to the time efficiency of code
from the viewpoint of Java programmers
in the context of archetypal ad-hoc code.

EXPERIMENT PLANNING

A. Context and Variable Selection

Although one of the categories defined by the FindBugs tool is named Performance, we think that also issues belonging to other categories could actually affect the time efficiency of code. For this reason, we select from FindBugs site a subsection of issues that might have a negative impact on the time efficiency with respect to the following set of criteria.

A Issue belongs to category performance

B Issue has a negative impact on performance with respect to expert judgment. The selection is made by the authors of this paper: two of them are professors of Java Programming course at Politecnico di Torino since more than ten years, whilst the first author is a second year PhD Student assisting the professors in the Java Course since four years. The experts read the description of the issues and for each of them classified them into one of the following categories (and implicitly assigned the relative score):

a) the issue impacts negatively the time efficiency of code (score: +1)
b) the issue does not impact negatively the time efficiency of code (score: -1)
c) no decision (score: 0)

Each issue is assigned a score that is the sum of the scores corresponding to the categories selected by the experts. Then an issue is selected for the experimentation when the total score is ≥2.

C Refactoring does not change functionality. For instance, the issue DLS OVERWRITTEN INCREMENT looks for code that performs an increment operation and then immediately overwrites it (e.g., i++; added in a for loop to skip an iteration). A possible code refactoring action would be to delete the offending increment: however, this action could change the functional behavior of the code, hence the issue is not selected.

D Efficiency does not depend on local (e.g. network) factors. For example, the issue DMI BLOCKING METHODS ON URL has a negative impact on performance because the *equals* and *hashCode* method of *URL* perform domain name resolution, thus

this can result in a performance hit. However, this case is out of our interest because the cause of delay is the network and not the code. For the same reason we do not include in the experimentation DMI COLLECTION OF URLS .

E Identification of one issue per equivalence classes. The aim is to pick only one issue from each set of similar issues. In fact some issues are redundant, or one is a generalization of many others. For instance, consider the issue BC IMPOSSIBLE INSTANCEOF: it is signaled when the *instanceof* operator will always return *false*, hence this is a useless operation that might lead to a delay. A similar issue is BC VACUOUS INSTANCEOF, that is complementary to the previous one, because it is signaled when *instanceof* test will always return *true*. Therefore latter issue is representative also for the former one.

An issue is selected if it satisfies the following combination of the five criteria: $(A \vee B) \wedge C \wedge D \wedge E$.

Table 24 shows all the issues selected as objects in the experiment. The first column indicates the numerical ID of the issue, the second column indicates its name, the last indicates whether the issue belongs to category performance or not (criterion A).

VARIABLE SELECTION AND HYPOTHESES FORMULATION

Since the goal of the study is to evaluate the relationship of issues with time efficiency the only dependent variable is the execution time t. We will consider two variants of the same code: either containing the issue (I) or with the issue refactored out (R). Therefore the main factor we use is the code type, $C \in \{I,R\}$.

In addition we measure and control other independent variables:

- the specific issues ($Issue \in 1..20$) in the set of issues selected as described above,
- the platform (P), both hardware and software, where the experiment is conducted,

– the batch run (B) of each specific experiment.

Given our original research question and the selected variables we can formulate our null and alternative hypotheses, where the subscript indicates the level of the factor.

$H_0: t_I \geq t_R$

$H_a: t_I < t_R$

INSTRUMENTATION AND EXPERIMENT DESIGN

The instrumentation required for our experiment is a software framework that allow the measurement of the execution times of the two different code fragments. Inspired by the JUnit[14] framework for automated software testing we developed a very simple framework. It consists of an abstract class, Experiment, that can be extended by concrete experimental classes. Each experimental class must provide two methods performWithIssue() and performWithoutIssue() that contain respectively the code including the issue and with the issue refactored out. In addition the method setUp() may be optionally redefined to prepare for the execution. For instance the experimental class for the issue DMI USING REMOVEALL TO CLEAR COLLECTION can be written as follows:

```
public class
DMI_USING_REMOVEALL_TO_CLEAR_COLLECTION
extends Experiment {
  Collection<Integer> col;
  private void setUp(){
       col = new LinkedList<Integer>();
            for(int i=0;i<1000;i++){
                 col.add(Integer.valueOf(i));
        }
```

[14] http://www.junit.org/

```
  }
  public void performWithIssue() {
    col.removeAll(col);
  }
  public void performWithoutIssue() {
    col.clear();
  }
}
```

The execution times of the methods `performWithIssue()` and `performWithoutIssue()` are expected to be in the order of nanoseconds. Unfortunately the standard measurement methods are not able to record precisely times at such order of magnitude. For this reason, the execution of each method is repeated consecutively a very high number of times (e.g. 1 million) to accumulate enough time to be detected by system APIs. We assume that each execution of the measured methods is independent on each other. This is true if no attribute is used except those initialized in the `setUp()` method.

The framework provides the method:

`perform(int nSamples , long nIter)`

that returns the results of the experiment in terms of the execution times. It takes as parameters two integers: the number of measurement samples to be generated (`nSamples`, set to 100 by default), and the number of iterations of the perform methods (`nIter`, set to 1 million by default). At the end of the experiment we will have nSamples samples, each of them representing the execution times of nIter iterations of both perform methods. We decide to have a batch of 6 runs of the basic experiment; each run was carried on at different random times during the day to compensate the possible confounding effect of periodical tasks performed by the operating system.

In addition, since the software and hardware platform is extremely relevant in terms of complexity when compared to the experimented code fragments, we decided to

execute the experiment batch on three different platforms. Table 25 contains the characteristics of the platforms that hosted the experiments.

ANALYSIS METHODOLOGY

The goal of data analysis is to apply appropriate statistical tests to reject the null hypothesis. The analysis will be conducted separately for each issue in order to evaluate

Table 24. Issues selection

Code	Issue	A
1	BC VACUOUS INSTANCEOF	
2	BX BOXING IMMEDIATELY UNBOXED TO PERFORM COERCION	X
3	DLS DEAD LOCAL STORE	
4	DM BOOLEAN CTOR	X
5	DM NEW FOR GETCLASS	X
6	DM NUMBER CTOR	X
7	DM STRING CTOR	X
8	DM STRING TOSTRING	X
9	DMI RANDOM USED ONLY ONCE	
10	DMI USING REMOVEALL TO CLEAR COLLECTION	
11	ISC INSTANTIATE STATIC CLASS	
12	RCN REDUNDANT NULLCHECK OF NONNULL VALUE	
13	REC CATCH EXCEPTION	
14	SBSC USE STRINGBUFFER CONCATENATION	X
15	SIC INNER SHOULD BE STATIC	X
16	SS SHOULD BE STATIC	X
17	UM UNNECESSARY MATH	X
18	UPM UNCALLED PRIVATE METHOD	X
19	URF UNREAD FIELD	X
20	WMI WRONG MAP ITERATOR	X

which one has an actual impact on time efficiency.

First of all we will test the null hypothesis H_0 for each issue across all platforms. Then we will analyze separately the different platforms.

Since we expect the values not to be normally distributed, we will adopt non parametric tests, in particular we selected the Mann-Whitney test [91]. Since the hypothesis is clearly directional the one-tailed variant of the test will be applied. We will draw conclusions from our tests based on a significance level α=0.01, that is we accept a 1% risk of type I error – i.e. rejecting the null hypothesis when it is actually true. Moreover, since we perform multiple tests on the same data – precisely twice: first overall and then by platform – we apply the Bonferroni correction to the significance level and we actually compare the test results versus a α_B=0.01/2=0.005.

After testing our experimental hypothesis, we will also check the potential confounding effect introduced by the co-factors: the platform and the different batch runs. Since the co-factors have more than two levels, we analyze the dependence of execution time on them using the Kruskall-Wallis rank sum test [91]. The null hypotheses we will attempt rejecting is that the co-factors have no effect on the dependent variable (time).

Table 25. List of platforms hosting the experiments

Platform	U	W	M
Operating	Ubuntu 10.10	Windows 7	Mac OS X 10.6.6
System	kernel 2.6.35-25	Home Premium	Darwin 10.6.0
Bits	64	64	64
Processors	2	2	2
Proc. Type	Intel Core 2	Pentium Dual	Intel Core 2 Duo
	T5270	Core T4500	
Proc. Freq.	1.40 GhZ	2.30 GHz	2.66 GHz
Memory	2 GB	4 GB	4 GB
Java SE	1.6.0	1.6.0	1.6.0
build	_22-b04	_23-b05	24-b07

VALIDITY EVALUATION

We identify two important threats to the validity of the experiment. The first threat affects the internal validity: experiments are executed inside an operating system, hence confounding factors could affect final results. Moreover, it is possible that the execution times for individual instructions are not independent from the execution history [83], because of caches and pipelines in processors, that could also cause the appearance of timing anomalies: therefore, we accept that the execution time of individual instructions may vary depending on the state of the processor in which they are executed, because we can not control the processor and avoid the hardware-related problems. However, it is possible to take some counter measures to reduce the noise introduced by the upper levels (OS and VM): we repeat the experiment 6 times on three different operating systems and machines, obtaining overall 1800 samples for each version of the code, and we isolate as much as possible the environment in which the experiment program runs, disabling for instance network and network routines or avoiding to launch the program in the same time of operating system subroutines. Furthermore, the experiment is the only user program that runs in the machine. All these provisions do not delete the confounding factors, but limit them and let us to have a reduced noise on results.

The second threat is a construct threat: if a difference is found, we say that the cause of the difference is the refactoring action. However, the platform on which the code runs is also affecting results. Therefore, there are generalization problems derived from this issue: we try to control this threat by using three different platforms. Another issue is that the code refactoring could not unique for each code smell, and different refactorings can bring to very different improvements of execution times: for this reason, the estimated improvements are specific to refactoring action we implemented.

Table 26. Summary of execution times

Platform:	all			M			U			W		
ID	t_I	t_R	p	t_I	t_R	p	t_I	t_R	p	t_I	t_R	p
1	34.72	34.48	**< 0.001**	55.41	55.32	0.01	47.04	47.13	1.00	1.70	1.00	**< 0.001**
2	8.39	2.78	**< 0.001**	10.03	3.67	**< 0.001**	12.70	2.73	**< 0.001**	2.45	1.93	**< 0.001**
3	68.10	35.27	**< 0.001**	109.43	54.32	**< 0.001**	90.78	46.99	**< 0.001**	4.10	4.51	1.00
4	9.81	5.43	**< 0.001**	10.57	4.19	**< 0.001**	13.63	7.18	**< 0.001**	5.24	4.93	**< 0.001**
5	180.20	183.31	1.00	167.03	155.49	**< 0.001**	237.84	242.74	1.00	135.72	151.69	1.00
6	9.65	4.78	**< 0.001**	10.64	3.07	**< 0.001**	13.77	7.12	**< 0.001**	4.53	4.16	**< 0.001**
7	14.72	5.15	**< 0.001**	17.29	4.20	**< 0.001**	18.88	7.09	**< 0.001**	7.99	4.16	**< 0.001**
8	84.16	88.54	1.00	75.29	75.79	1.00	113.92	121.16	1.00	63.26	68.67	1.00
9	2162.58	1117.11	**< 0.001**	326.57	164.02	**< 0.001**	3687.10	1901.80	**< 0.001**	2474.06	1285.49	**< 0.001**
10	468.66	213.77	**< 0.001**	411.45	210.21	**< 0.001**	728.38	278.32	**< 0.001**	266.16	152.78	**< 0.001**
11	8.70	5.08	**< 0.001**	8.24	4.17	**< 0.001**	13.15	6.84	**< 0.001**	4.70	4.22	**< 0.001**
12	591.41	592.08	0.74	80.33	80.22	0.42	1671.77	1673.89	1.00	22.14	22.14	0.47
13	35.81	35.47	**< 0.001**	55.64	55.08	**< 0.001**	47.25	47.31	1.00	4.54	4.03	**< 0.001**
14	561.95	302.34	**< 0.001**	455.71	268.63	**< 0.001**	767.91	409.92	**< 0.001**	462.24	228.46	**< 0.001**
15	6.98	7.04	0.08	5.28	5.66	1.00	8.82	9.07	1.00	6.83	6.39	**< 0.001**
16	9.71	8.62	**< 0.001**	10.78	8.35	**< 0.001**	13.77	13.45	**< 0.001**	4.57	4.05	**< 0.001**
17	592.05	594.41	0.55	3.84	4.15	1.00	1767.72	1775.06	0.01	4.59	4.01	**< 0.001**
18	537.67	544.22	1.00	462.41	462.53	1.00	707.32	716.04	1.00	443.28	454.10	1.00
19	11.80	11.04	**< 0.001**	13.94	13.89	**< 0.001**	16.23	13.93	**< 0.001**	5.22	5.30	**< 0.001**
20	582.91	539.86	**< 0.001**	558.12	514.13	**< 0.001**	668.55	633.61	**< 0.001**	522.05	471.84	**< 0.001**

We make available on our website [15] the Eclipse project of the experiment framework developed and we invite other researchers to repeat the experiment and compare the results with ours. This is a further strategy control for the threats mentioned above: in

[15] http://softeng.polito.it/vetro/confs/InfQ2011/EfficiencySmells.zip

this way it is possible to build up a benchmark and make the empirical validation of the impact of issues on efficiency more reliable.

ANALYSIS AND INTERPRETATION

The data collected during the experiments are summarized in Table 26, which reports the average execution times expressed in milliseconds, for the three different platforms and separating the execution time of the code containing the issue (t_I) from the execution time of code with the issue refactored out (t_R).

We can immediately observe a wide variability of times and small differences mainly among different issues, but also to a smaller extent between platforms. In order to report in the same diagram such varying values we opted for the rest of this analysis to plot times using a logarithmic scale.

Columns p in Table 26 report the p-values of Mann-Whitney tests carried on overall and by platforms (W, U, M); statistically significant values are reported in bold face. The boxplot of Figure 30 reports the execution times recorded in the experiment, divided by issue, in practice it adds the dispersion to the information provided in the first four columns of the table. Execution times of code containing the issue is drawn in black, while for code with the issues refactored out (R) it is represented in red. A gray background is present corresponding to the issues for which we can reject the null hypothesis. The boxplot in Figure 31 is similar but it reports the execution times recorded in each platform.

We can observe a range of patterns in terms of hypothesis rejection overall (Figure 30) and for specific platforms (Figure 31). On one side, the null hypothesis can be rejected both overall and for every tested platform for issues 2, 4, 6, 7, 9, 10, 11, 14, 16, 19, and 20. At the opposite side, the null hypothesis could not be rejected neither overall nor on any platform for issues 8, 12, and 18. Among the remaining issues: for issues 3 and 13 we rejected H_0 overall and on two out of three platforms, for issue 1 we could reject overall and on 1 platform, and for issues 5, 15, and 17 we could reject only on one platform.

The effect of co-factors on the main dependent variable has been checked with the Kruskal-Wallis test, whose results are reported in Table 27 .

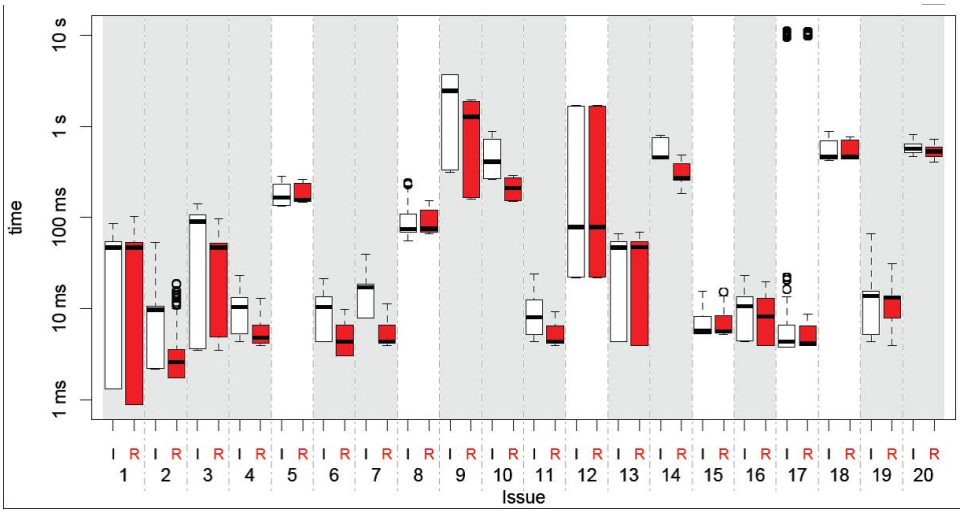

Figure 30. Boxplot of execution times for all issues.

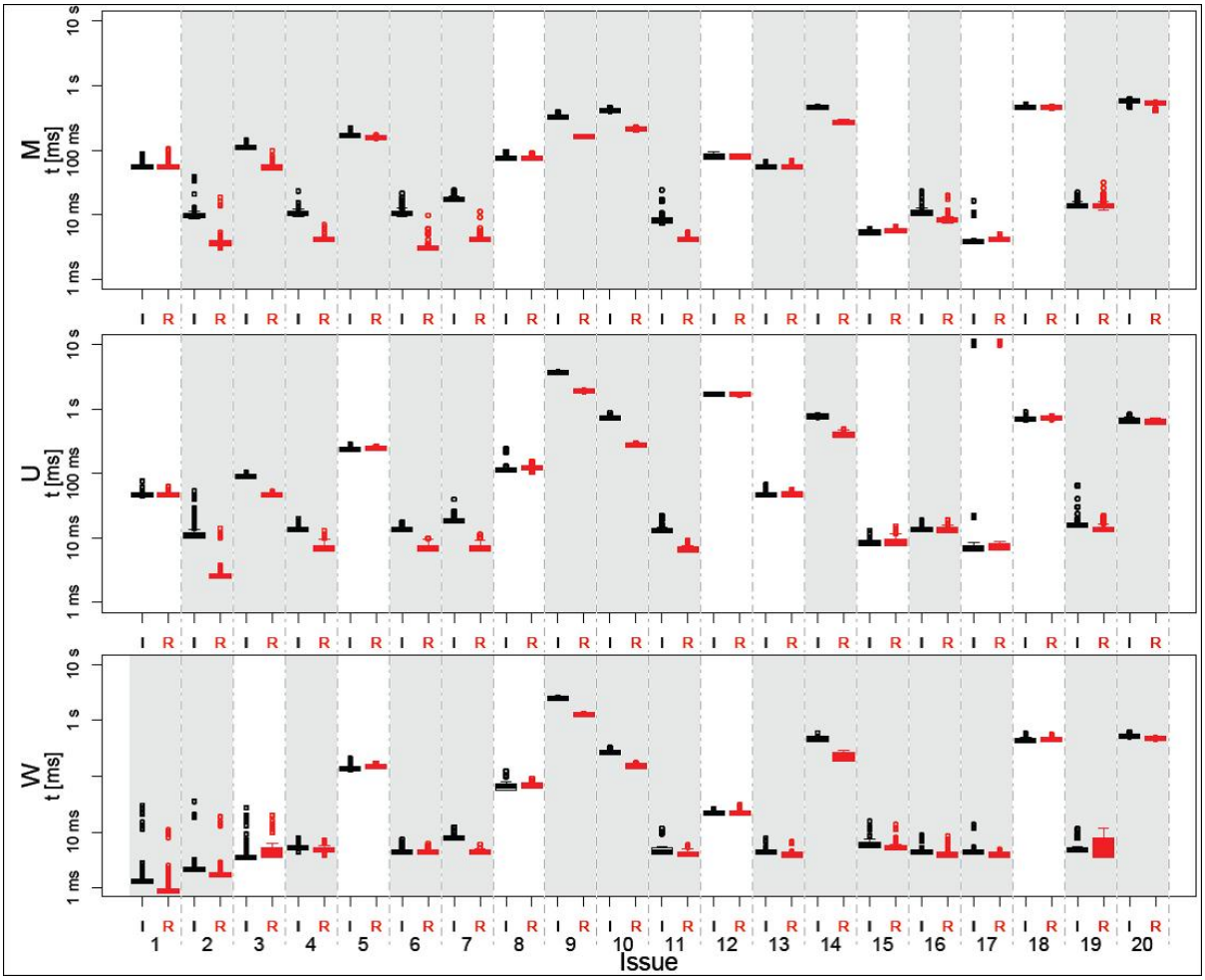

Figure 31. Boxplot of execution times for all issues, per platform

We observe that, concerning the Platform, the hypothesis can be rejected for all issues. While, the batch Run influenced the execution time of 11 out of 20 issues.

DISCUSSION

Based on results in Table 26, 11 of the issues selected have undoubtably a negative impact on time efficiency, since there is a statistically significant difference in all conditions. Such issues are:

2) A primitive boxed value is constructed and then immediately converted into a different primitive type (e.g., *newDouble*(*d*).*intValue*()) instead of performing direct primitive coercion (e.g., (*int*)*d*).

4) A method invokes a Boolean constructor, instead of using *Boolean*.*valueOf*(...)

6) Code uses *newInteger*(*int*) whereas *Integer*.*valueOf*(*int*) should be used, because it allows caching of values to be done by the compiler, class library, or JVM.

7) The *java*.*lang*.*String*(*String*) constructor is used instead of *String* parameter directly.

9) Code creates a *java*.*util*.*Random* object, uses it to generate one random number, and then discards the *Random* object. Subsequently, to generate a new random number, a new *java*.*util*.*Random* object is created. Code should be refactored so that the *Random* object is created once and saved to be invoked each time a new random number is needed.

10) The code removes all elements from a collection *c*, using *c*.*removeAll*(*c*) instead of *c*.*clear*().

Table 27. P-values of Kruskal-Wallis test for co-factors

ID	Platform		Run
1	≤ 0.001	*	0.01
2	≤ 0.001	*	0.60
3	≤ 0.001	*	≤ 0.001*
4	≤ 0.001	*	0.02
5	≤ 0.001	*	0.16
6	≤ 0.001	*	0.02
7	≤ 0.001	*	≤ 0.001*
8	≤ 0.001	*	≤ 0.001*
9	≤ 0.001	*	0.19
10	≤ 0.001	*	0.18
11	≤ 0.001	*	≤ 0.001*
12	≤ 0.001	*	0.04
13	≤ 0.001	*	≤ 0.001*
14	≤ 0.001	*	≤ 0.001*
15	≤ 0.001	*	≤ 0.001*
16	≤ 0.001	*	0.05
17	≤ 0.001	*	≤ 0.001*
18	≤ 0.001	*	≤ 0.001*
19	≤ 0.001	*	≤ 0.001*
20	≤ 0.001	*	≤ 0.001*

11) A class allocates an instance of a class that only supplies static methods. The refactoring action is to use the static methods directly using the class name as a qualifier.

14) Code builds a *String* using concatenation in a loop instead of using *StringBuffer*.

16) A class contains an instance final field that is initialized to a compile-time static value. Since the field is immutable for each object of the class, it should be static.

19) A field which is never read

20) Code accesses the value of a *Map* entry, using a key that was retrieved from a keySet iterator. It is more efficient to use an iterator on the *entrySet* of the map, to avoid the *Map.get*(*key*) lookup.

More than half of the issues (nr 2, 4, 6, 7, 9, 11) concerns a useless creation of objects. The other issues are related to different problems, relating to inefficient, albeit functionally correct, set of operations.

Being known the number of times that the code containing the issues is invoked, it is possible to estimate the average delay that each of these issues bring to the code. The code fragments invoke issues only once to minimize the confounding factors: therefore the total number of invocations is 1 million times. However, issues 14 and 20 are inside a for cycle of respectively 5 and 10 iterations, thus they are executed 5 and 10 million of times. Table 28 contains the estimated delays, in nanoseconds, for each issue and platform, computed aggregating the measurements of the different batches.

All issues concerning the useless creation of objects (except nr 9) have similar

Table 28. Mean expected delay [ns] of verified issues.

		Platform		
Issue	**Iterations**	**U**	**W**	**M**
2	1 M	8.23	0.43	6.14
4	1 M	6.43	0.43	6.29
6	1 M	6.67	0.43	7.40
7	1 M	11.81	3.91	13.02
9	1 M	1786.00	1185.80	159.60
10	1 M	449.20	113.00	201.10
11	1 M	6.29	0.43	3.90
14	5 M	74.74	51.16	37.81
16	1 M	0.40	0.43	2.36
19	1 M	2.21	0.44	0.11
20	10 M	4.36	4.96	4.07

unitary delays: few nanoseconds (3 to 11) in environments U and M, less than 1 in environment W. Similar delay is for the wrong map iteration (issue 20). The issue nr 9 (useless Random object) has the highest delay, that is in the order of magnitude of 1 μ*s*. Also issue 10 (emptying the content of a collection) exhibits high delays (in the order of magnitude of some hundreds of *ns*), whereas issue 14 causes a delay of tens of *ns*. The smallest delays are those ones of issue 16, a field that should be static, and issue 19, a field that is never read. In real contexts these numbers can easily reach the order of *ms* and *s*: in real projects there are millions of lines of code where there can be millions of these simple issues or even billions if they are inside *for*/*while* cycles. Moreover, these figures may be directly relevant in Real Time Applications, where the usage of Java is steadily increasing (for instance, Boeing has adopted real-time Java in drone aircrafts, and the United States Navy decided to use it in its next-generation battleships [84].

On the basis of the above findings, we can assert that likely also the following issues, which were not object of our experiments, have impact on time efficiency, because related to the previous eleven: BX BOXING IMMEDIATELY UNBOXED, BX UNBOXED AND COERCED FOR TERNARY OPERATOR, DM BOXED PRIMITIVE TOSTRING, DM FP NUMBER CTOR (similar to issues 2, 6), DM STRING VOID CTOR (similar to issue 7), issue UUF UNUSED FIELD (similar to issue 19).

Moreover, we observe that issues 8, 12 and 18, instead, do not have any negative impact on performance. We further investigate this fact computing the estimated differences between the two set of execution times ($t_I - t_R$). Issue DM STRING TOSTRING (nr 8, call *toString*() on a *String*) has negative differences both overall and in every platform; issue RCN REDUNDANT NULLCHECK OF NONNULL VALUE (nr 12, check of a known null value) has a significant difference only in platform U, whilst UPM UNCALLED PRIVATE METHOD (nr 18, a private method never used) is significantly different under all conditions. These data shows that the refactored code of these three issues performs worse than the original code: this is an unexpected result. Nevertheless, it is probable that optimization enforced by the compiler and/or the hardware deletes the negative effect of the three issues at run time. All the three issues concern useless

operations and bad programming practice: even if they do not impact the efficiency, refactor the code is worthy to increase its maintainability or decrease its complexity. Similar issues in Findbugs, not selected for our experiment, are : RCN REDUNDANT (COMPARISON OF NULL AND NONNULL VALUE, COMPARISON TWO NULL VALUES, NULLCHECK OF NULL VALUE, NULLCHECK WOULD HAVE BEEN A NPE) that are similar to issue nr 12, and UMAC UNCALLABLE METHOD OF ANONYMOUS CLASS that is similar to issue 18.

4.3. EXECUTION TIME EFFICIENCY IMPROVEMENT BY MEANS OF CODE ISSUE REFACTORING: A CONTROLLED INDUSTRIAL EXPERIMENT

The execution time efficiency of a software function can be assessed by measuring the execution time: the shorter the better, trivializing the concept.

Excluding the obvious hardware techniques, time efficiency can be improved selecting appropriate algorithms, leveraging compiler optimization capabilities, and relying on execution infrastructures such as operating systems and virtual machines.

Java offers many benefits: portability, security, dynamic program composition through dynamic class loading and automatic memory management. However, these advantages become drawbacks when dealing with performance. A possible explanation is the overhead introduced at execution time by the features mentioned above, that make Java applications several times slower than equivalent compiled C programs [90]. Moreover, the high level of hardware abstraction used by Java produces byte code without any knowledge of the underlying CPU, thus without an efficient optimization.

Most Java optimization techniques operate statically [92], others try to improve the performance of a program while it executes [90]. Considering the three code representation levels [93] – high-level representation (HLR), directly interpretable representation (DIR), and directly executable representation (DER) – the most common static and dynamic techniques operate at the DIR or DER level. However, recent studies [94] demonstrated that optimizations at DIR or DER levels do not guarantee lower execution time. For this reason we faced the Java optimization problem at HLR with a new approach based on Automated Static Analysis (ASA). ASA focuses on the HLR with the goal of identifying issues and possible improvement areas.

A common drawback to most ASA tools is the huge number of code issue detections that is signaled to the developers. The challenge in adopting ASA tools is to customize them in order to detect only relevant issues.

The problem we address in this paper is to identify the issues having an impact on time efficiency and empirically assess them.

In particular we used FindBugs, a widely used ASA tool for the Java code, to analyze the source code of a web application developed by a large Italian company. The goal is to identify the performance problems of the application and verify whether they derive from bad programming practices detectable by means of FindBugs.

To do that, we set up the empirical methodology already applied in the previous study.

GOAL AND RESEARCH QUESTION

As we have seen in Section 4.1, Performance efficiency is defined in ISO-IEC 25010 [42] as "the performance relative to the amount of resources used under stated conditions". Based on the type of resources under measurement, two efficiency sub-characteristics are specified by the standard: time behavior and resource utilization. We focus our study on time behavior, i.e. the amount of time to perform one or more operations.

Code performance issues, revealed through static analysis, are cues that indicate potential efficiency problems. Therefore a refactoring operation that removes the issue will probably remove the problem too. Apparently pure speculation does not always lead to the identification of the right issues, as our previous experiment in Section 4.2 showed. Only empirical evidence can provide a credible support to performance issues as indicators of actual problems. Our aim is to empirically assess the impact on time efficiency of code issues detectable by ASA.

We formalize the goal of our study according to the GQM template [34]:

Analyze	code performance issues on source code
For the purpose of	identifying their effect
With respect to	execution time efficiency
From the point of view of	developers
In the context of	an industrial Java web application

The selected epistemological approach consists in the empirical comparison of the original application with a modified version with the issues removed by means of refactoring. The research question we aim to answer is:

Does the refactoring of code issues affect the time efficiency of the application?

A possible evidence of an effect, where the refactored version is faster than the original, would confirm the role of the issue as a performance problem predictor.

CONTEXT DESCRIPTION

The study was conducted in the IT department of a large Italian company (about 21000 employees in 110 countries). The focus was on a web application developed in Java.

The application is a J2EE, servlet-based software developed with the modeling tool Web Dynpro, a client-independent UI builder in the SAP NetWeaver platform. Web Dynpro is used to develop user interfaces for business applications, based on the model-view-controller paradigm [95]. The use of Web Dynpro minimizes manual UI coding because programmers use visual tools to design and reuse components. However, some customization of components and implementation of additional logic are performed manually on the generated code.

The application size is 18142 NCSS (non-commented source statements), it has 49 dependencies on external libraries and it is compiled with Java 1.4.2. The application runs on SAP NetWeaver 7.01 sp0.

The application architecture conforms to the MVC pattern: the user interacts with a View component that takes care of the presentation, user requests are processed by the Controller component, and the Model component holds the data communication with the Controller through a network connection.

The main functionality of the application is to let the user search and filter documents from the company repository, which contains thousands of documents in XML format. In our study we focus on the two basic operations: search and filter. Figure 32 shows the interactions taking place among the main components to carry out the two operations. The application, accessed via a web browser, is part of a pool of web applications and servlets hosted in the SAP portal of the company.

Concerning the search operation: the user enters one or more keywords k_s in a text box and submits her request pushing the search button (*onActionSearch(k_s)*). The controller, invoked by *executeSimpleSearch(k_s)*, receives the text from the view and performs some checks. Finally it forwards the request to a web service located on a remote machine. The web service is in charge of looking in the repository for documents containing k_s and sending them back to the controller. Data are sent back to the view, which builds up a table containing a row for each document retrieved and a column for each different property of documents (e.g.: title, owner, date, etc.).

Once results are displayed on the screen, the user can filter them specifying a keyword or key phrase k_f corresponding to a property p. The view transmits the keyword k_f and the property p to the controller (*onActionFilter*(k_f,p)), which in turns selects, among the documents already retrieved from the web service, those in which p contains *kf*.

It is important to emphasize that search and filter operations must be executed in sequence because the filter operates on the results of a search operation.

The web service component is outside the scope of this work because we had no access to its source code. In addition since the service resided on a remote host geographically far from the application host, the network delay plays a significant and unpredictable role.

The server runs on a virtual machine with operating system Microsoft Windows XP Professional Vs 2002 Service Pack3. CPU of the physical machine is Intel Xeon CPU E7430, 2.13GHz, and RAM is 3 GB, Operating System of the hosting machine is Windows Vista.

DETECTED ISSUES AND SELECTION

The automatic static analysis was conducted on the View and Controller components. The FindBugs tool detected 109 issues. The issues belong to 14 distinct types out of 369 detectable by the tool.

An issue can be detected in different places of the code: we call them occurrences or detections. Table 29 reports for each issue type (first column) the number of detections (second column). In addition the table reports the category of each issue type according to the FindBugs classification (third column). Just three out of the 14 detected issue types, counting 48 detections, belong to the performance category according to the classification of the FindBugs tool.

It is important to remark that, according to the FindBugs classification, each issue type belongs to exactly one category. However, issues belonging to other categories may have a performance impact too. For instance, the issue DLS DEAD LOCAL STORE belonging to category Style could impact the time efficiency of code. In fact, the issue detects an instruction that assigns a value to a local variable, but the value is not used in any subsequent instructions: time is wasted in a useless operation.

So, we decided to set up an expert assessment procedure to identify, among the detected code issues, which have a potential impact on performance.

The experts are the three academic authors of this paper: two of them taught object-oriented programming for more than ten years, another is a PhD candidate working as teaching assistant in a Java course since six years.

The procedure they followed is straightforward:

- each expert carefully read the issues description and classified them into one of the following categories which is associated with a score:
  - issue with a negative effect on time efficiency of code (score: +1)
  - issue without a negative effect on time efficiency (score: -1)
  - no decision (score: 0)
- each issue is assigned a total score that is the sum of the scores assigned to the issue by the experts.
- An issue with a total score greater or equal than 2 is considered performance relevant (fourth column of Table 29).

Since the refactoring operations are fundamental in our investigation, we analyzed the corresponding refactoring operation for each issue type. The goal was to identify those

refactoring that could be implemented without affecting the behavior of the code. For instance, the issue DLS OVERWRITTEN INCREMENT does not satisfy these requirements for the following reason: since the issue detects a double write operation on a loop counter (eg. twice i++), the associate refactoring action is to delete the second write operation. However this change will alter the functional behavior of the code. Or must be supervised by the developer to check the refactoring is correct. The fifth column in Table 29 indicates which issues might be refactored preserving the functional behaviour.

Yet another selection criterion is the number of detections. The impact of issues with a few detections is expected to be smaller than the impact of issues detected more times. We filtered out less frequent issues in the following way: we computed the distribution of the issues occurrences, and we decided that a reasonable limit is the median, which takes out 50% of the issues. The resulting inclusion criterion is: issue frequency > 2. Issues satisfying this criterion have the number of detections in bold in the second column of Table 29.

The last column of Table 29 indicates which issues satisfied all criteria and were selected for the experimentation: this resulted in total of 26 occurrences all together.

EXPERIMENT PLANNING

We structured our study as a controlled experiment and we planned it following a typical approach for experiments in software engineering [7]. We list and describe in the following subsections the main steps performed:

A. Treatment selection

B. Hypothesis formulation

C. Variable description

D. Experiment Design

E. Analysis method

F. Threats to validity

A. Treatments selection

Table 30 reports the details about the three issue types that satisfy all conditions in the selection process, out of the 14 detected.

The treatment is the program with and without issues. Since there are three issues we manually refactored the original program three times, removing one issue type at a time, in order to understand the effect of each issue type. In addition we included also a version of the program where all detections of the three issues types were removed. Overall we have these four treatments plus the original program:

Table 29. Issues and their characteristics

Issue ID	Detections	FindBugs Category	Performance Relevance (Expert)	Function Preserving Refactoring	Selected in Experiment
URF UNREAD FIELD	**43**	**Performance**	✓	–	–
BC UNCONFIRMED CAST	**27**	Bad Practice	–	–	–
DLS DEAD LOCAL STORE	**20**	Style	✓	✓	✓
DM BOOLEAN CTOR	**3**	**Performance**	✓	✓	✓
REC CATCH EXCEPTION	**3**	Style	✓	✓	✓
BC VACUOUS INSTANCEOF	2	Style	✓	✓	–
NM CONFUSING	2	Bad Practice	–	✓	–
SIC INNER SHOULD BE STATIC	2	**Performance**	✓	✓	–
SIO SUPERFLUOUS INSTANCEOF	2	Correctness	✓	✓	–
DB DUPLICATE BRANCHES	1	Style	–	–	–
NP LOAD OF KNOWN NULL VALUE	1	Style	–	–	–
NS DANGEROUS NON SHORT CIRCUIT	1	Style	✓	–	–
RV CHECK FOR POSITIVE INDEXOF	1	Style	–	–	–
SA FIELD DOUBLE ASSIGNMENT	1	Style	–	–	–

Table 30. Details about selected issues

Issue	Description	Refactoring action
DLS DEAD LOCAL STORE	An instruction assigns a value to a local variable. However, the value is neither read nor used in subsequent instructions.	Delete the instruction and, if the variable is not used anymore, delete also the variable.
DM BOOLEAN CTOR	Creation of new instances of java.lang.Boolean wastes memory, because Boolean objects are immutable.	Use Boolean:valueOf() instead of the constructor to use objects from a pool
REC CATCH EXCEPTION	A method uses a try-catch block that catches Exception objects, but Exception is not thrown within the try block	Remove the try-catch block

Table 31. Variables

	Variable	Description	Type	Role
Independent	Version	The program version obtained by refactoring	Nominal: {OR, DS, IBC, UE, ALL}	Block
	Operation	The type of operation, either search or filter.	Nominal: {S, F}	Block
	Size	The size category of the document set retrieved: small, medium or large	Ordinal: (SM, M, L)	Block
	Execution order	The order in which operations of a given version are executed	Ordinal: {1,2,3…36}	Random
Dependent	t	Time to perform the operation	Ratio	Block

- OR (Original): the original application,
- DS (Dead Stores): refactored version with DLS DEAD LOCAL STORE occurrences removed,
- IBC (Inefficient Boolean Constructor): refactored version with DM BOOLEAN CTOR occurrences removed,
- UE (Useless Exception): refactored version with REC CATCH EXCEPTION occurrences removed,
- ALL: refactored version with occurrences of all the three issues removed.

After performing the refactoring, we run a suite of functional tests to ensure that refactoring did not introduce errors.

B. Hypothesis formulation

On the basis of the research question formulated, we define a formal hypothesis that can be verified by means of statistical tests. As customary in empirical investigations we define a null hypothesis, which we will try to refute, and an alternative one, which we aim at confirming:

H_0: $t_{OR} < t_{Refactor}$

H_a: $t_{OR} \geq t_{Refactor}$

Where t_{OR} is the time taken by the original application and $t_{Refactor}$ is the time taken by the refactored version.

The above null hypothesis will be tested for each refactored version of the program (DS, IBC, UE, and ALL).

C. Variable description

The dependent and independent variables considered in the experiment are summarized in Table 31. The treatment in the experiment consists in applying the refactoring transformation to the application in order to remove issues. So the main independent variable is the Version of program (OR, DS, IBC, UE, and ALL).

The second independent variable is related to the two operations – search and filter – performed by the application. Since they are profoundly different, we control the specific Operation performed in any experimental.

Both the search and filter operation time may depend on the number of documents retrieved from the repository. Therefore we defined another co-factor representing the document set Size as a categorical ordinal variable, with three values: small, medium, and large. Table 32 reports, for each category, the number of documents retrieved from the repository through the search and filter operations. The repository available for the experiment contained 179 documents (retrieved by the large search) and it is a small portion of the repository actually used, which is composed of thousands of documents.

The last co-factor is the order in which operations are performed, that might impact the execution time due to problems like memory degradation.

We could block [96] all independent variables but the operation execution order, which is randomized.

Finally, the study measures only one dependent variable, the execution time, t. We measure the average execution times and the relative confidence intervals. For this purpose we used a profiler tool: the only one suitable for our context was CA Wily Introscope Release 8.0.2.0 (Build 470970). The tool allowed us to monitor the web application through instrumentation and record the execution time of the application. All the runs were automatically launched and managed through Selenium tools. The raw metrics that we were able to collect from the software execution are:

t_{WDA}: time in SAP J2EE - **W**eb**D**ynpro **A**pplication that corresponds to the time employed by the application to complete the request. Looking at the sequence of the operations in Figure 32, t_{WDA} is the time elapsed between the message 1: *onActionSearch()* and the related return message. As a consequence it includes controller, view and model. It is also close to the total time observed by the users, except the browser-server communication delay.

t_{WS}: time in **W**eb**S**ervice that accounts for the time spent for acquiring the results of web service computation. Looking at Figure 32, it is the time spent in the "Net" cloud plus the time spent in the Model.

Since the web service is not considered in the experiment, as far as the search operation is concerned we must exclude the time spent in that component, including the time devoted to the communication; such time is represented by t_{WS}. Therefore the execution time, depending on the operation, is measured as follows:

$$t_{Search} = t_{WDA} - t_{WS}$$

Table 32. Documents set size vs operation

	Documents retrieved	
Size Category	**Search**	**Filter**
Small (SM)	1	1
Medium (M)	100	9
Large (L)	179	12

$t_{Filter} = t_{WDA}$

D. Experiment Design

The obvious design to adopt is a full factorial design, where a triplet (Version, Operation, Size) corresponding to the values of the block independent variables characterizes each experimental task. For instance, (IBC, S, SM) indicates that the IBC version is used to perform the search operation (S) on a small document set (SM), whereas (DS, F, L) indicates DS version, performing a find operation (F) on a large document set (L). As a consequence, considering both the main factor and the co-factor, the original generic hypothesis H0 must be specialized into Nhp detailed hypotheses:

Nhp = NRefactor • NOperation • NSize

where:

- NRefactor = (NVersion - 1) = 4: is the number of Versions but the original one, which is the reference version to compare with,
- NOperation = 2: is the number of operations,
- NSize = 3: is the number of document set sizes.

As a result we derived 24 detailed hypotheses.

In order to statistically test the detailed hypotheses, thus taking into account other potential confounding factors, we need to collect several execution time measures for each triplet (Version, Operation, Size).

To plan the collection of measures, it is important to recall that search and filter operations must be executed in pair therefore each experimental task included two operations: first a search and then a filter Operation; the execution time ought to be collected independently for each operation.

As far as Size is concerned, tasks pairs can be combined in six different sequences corresponding to the possible permutations of the three levels of the variable, i.e. 1)SM-M-L, 2)SM-L-M, 3)M-SM-L, 4)M-L-SM, 5)L-M-SM, 6)L-SM-M.

We organized the task execution in buckets: each bucket consisted of six distinct sequences; each sequence was made up of three tasks pairs each having different document set sizes. Since the order of the task pairs may represent a confounding factor, the order of the sequences within the buckets was randomized. All the tasks in a bucket were executed with the same Version, and we decided to collect 36 measures for each triplet (Version,

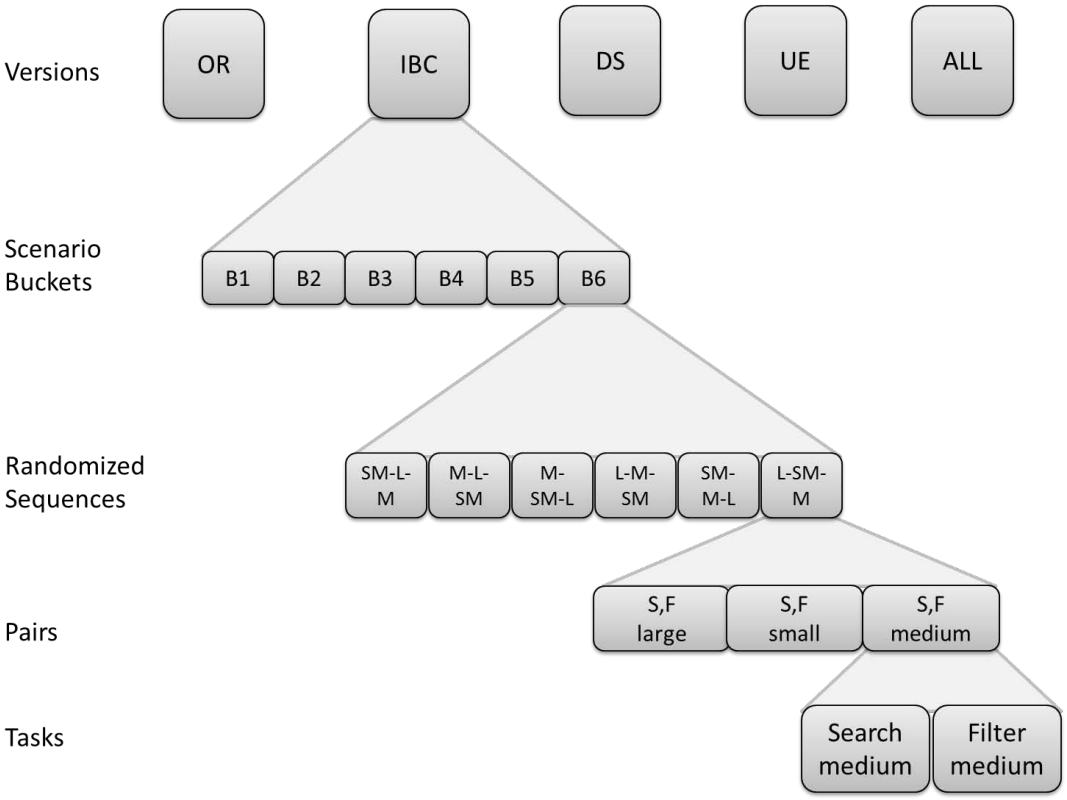

Figure 33. Experiment Design

Operation, Size). We achieved this by using each Version in six different buckets, as depicted in Figure 33 : the figure summarizes the overall design and the organization of buckets, sequences and tasks.

E. Analysis Method

In the analysis of data we followed a four steps approach.

First we perform a post-survey adjustment aimed at identifying and removing possible outliers and locating potential missing data.

In order to gauge the variability of the data we used the dispersion index and the coefficient of variation [97]. The first index is defined as the ratio of the variance to the mean, whereas the second one is a normalized measure of the dispersion of a probability distribution and it is defined as ratio of the standard deviation to the absolute value of the mean.

We identify outliers using a conservative criterion based on interquartile distance: a value is considered an outlier if its distance d from the third quartile (q3) is $d > 3 * (q3-q1)$, where q1 is the first quartile.

Then we provide an overview of the data by means of descriptive statistics; in addition we attempt matching measured data to known distributions. We perform test of normality also to determine the statistical methods to be used in the results analysis. The fit to gamma distribution is tested because execution times typically fit this distribution. In fact, usually execution times are modeled with an exponential distribution and the M/M/1 queue is the model that better fits execution times of web based applications [98] [99]. Since the execution of the search and filter is composed of a sequence of Java methods, assuming the execution times of the single Java methods exponentially distributed, the execution time of the whole operation might be modeled with a gamma distribution, which is the sum of exponential distributions.

Distribution fitting is performed by means of two distinct goodness of fit tests: the Kolmogorov Smirnov (KS) test [100] and the Anderson Darling (AD) test [101] with estimated critical values for gamma distribution from Shawky and Bakoban [102]. KS test is more stringent than AD test. AD test is more sensitive test but has the disadvantage that

critical values must be calculated for each distribution. KS test is more conservative because the critical values do not depend on the specific distribution, hence it is distribution independent.

As a third step we test the hypotheses outlined above. Due to the nature of the hypotheses we used one-tailed tests to compare two populations. In particular we use t-test [97] if the data is normally distributed (according to the tests outlined above), otherwise we adopt the Mann-Whitney test [103]. Since the same sample (OR) is used in multiple comparisons – four times, corresponding to the four refactored versions –, the probability of type I error is increased. A common practical remedy consists in applying the Bonferroni correction [104]. We will reject the null hypothesis with a p-value < $\alpha/4$

The fourth and latest stage consists in checking for the effect of additional co-factors. In particular we focused on the execution order of the tasks. For this purpose we formulate the null hypothesis that no trend exists in the temporal sequence of execution times. To test the hypothesis we perform the Sign test of Cox and Stuart [97] for the detection of a monotone trend.

For all the hypotheses tests we set a confidence level of 95%, that is we accept a probability $\alpha = 5\%$ of committing a type I error, i.e. rejecting the null hypothesis when it is true.

F. Threats to validity

We evaluate the threats of our study dividing them in external and internal threats.

A first external threat occurs for results generalization. Although delays computed are verified and consistent, they could change in other environments. A more powerful platform could mitigate the effect of the issues, whereas a less powerful platform could amplify them. However, we expect an amplification of delays when the application runs in a real usage scenario, with dozens of parallel accesses and with other web application concurrently running on the same SAP Netweaver server.

A second generalizability threat regards the effect of the document set size on execution times: we expect not homogeneous results and especially when the document set is small the execution times might be too short to be significantly compared. Moreover, the repository of documents used for the experiment is only a portion of the actual repository

used for the experimentation. Given that we only have 3 levels for the size co-factor, we will not be able to build a comprehensive model for the effect of the document set size on execution times, and as a consequence of the possible effect of the code issues delays on larger search/filter operations.

The last external threat is related to the generalizability of the performance refactorings to other software systems. This study shows that the performance refactoring adopted improve the execution time efficiency in a SAP module, but we do not know whether this is a special case due to the nature of the software or these selected refactoring actions might improve performance in other software system in general.

Regarding the internal threats, a first confounding factor is the fact that the server used for the experiment is not completely isolated. Although we asked employees to do not use it during experiments, we could not prevent it. We registered a few accesses to the virtual machine and a few requests to the SAP Netweaver server during the experiment. However, the small number of accesses let us consider the impact negligible on results.

Confounding threats related to the order of runs are verified and controlled in the experiment design. Finally, all measurements were done through Wily Introscope, and lack of precision of this tool could bring construct threats: we assume this possibility an outside chance.

RESULTS

A. Post-experiment adjustement

We adjusted the data by looking at their variability and searching for outliers.

We found that in 22 filter and 15 search task executions the time samples have a coefficient of variation > 1, i.e. the majority of execution times are more than twice the mean. Moreover, all task samples have dispersion indexes >> 1, which means over-dispersed data.

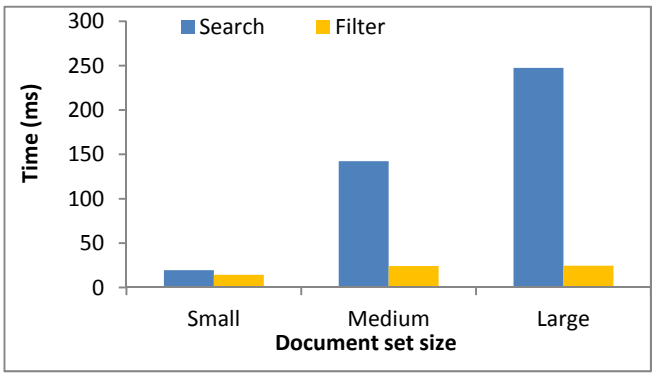

Figure 34. Execution times vs. Document size and operation

Based on our interquartile distance criterion for identifying outliers we observed a significant number of outliers: 123 out of 1245 samples, i.e. about the 10% of whole dataset. The very large variability of data and the high number of outliers observed motivates outliers' removal. However we conduct two analysis on results, firstly keeping outliers, secondly removing them to check the stability of the analysis: since results did not change after removing the outliers, hereinafter the diagrams and values refer to the data without outliers.

B. Data description

Figure 34 reports the average time to complete the search (blue) and filter (yellow) tasks. We can observe a huge difference between the two operations and a time increasing with the size of the handled document set, especially for the search. This is a confirmation of the assumption we made while designing the experiment.

Mean and median execution times of all operations are presented in Table 33 with variability. Standard deviation values indicate a large variability in all task types.

We check whether data distribution fits the normal and the gamma distribution. Results of the normality tests for Search and Filter operations show that in no combination of version-operation-size data are neither normally distributed (for both AD and KS tests) nor follow the Gamma distribution (for both tests as well).

C. Co-factor analysis

We measured the effect of the order of tasks execution, which was randomized in the experiment design. We investigate whether there is a dependence of execution times on task order. The null hypothesis that no trend exists is rejected in two cases for search, (all decreasing: DS-S-M (p-value = 0.0176) and IBC-S-L (p-value=0.0461). The test was rejected in one filter: IBC-F-M, p-value=0.005, decreasing trend again. Since only 4 times out of 30 showed trends for different sizes and versions, we can exclude that execution times systematically degrade or improve due to the order of tasks execution.

D. Hypotheses testing

Figure 35 reports the boxplot of execution times for the filter operation, while Figure 36 reports the boxplot for the search operation: the colored boxplots indicate the original application. The vertical lines group boxplots according to the document set size (first small, then medium, finally large).

To test the hypotheses, we compare the execution of each refactored version (DS, IBC, UE, ALL) to the original version (OR) time. Table 33 reports the mean and median difference together with the test p-value. Since the data is not normally distributed we opted for the non-parametric Mann-Whitney test.

In medium and large searches, the refactored versions were always faster than the original one as we can easily observe in Figure 36. The p-values reported in Table 33 are all far smaller than the α threshold therefore we can safely reject the null hypothesis for the search operation on medium and large sized document sets.

Conversely, based on the information gathered by boxplot observation and represented by p-values we cannot reject the null hypothesis for search operation on small document sets.

Concerning the Filter operation, for large document sets the boxplot in Figure 35 provides a clear indication of a significant difference that is confirmed by test results.

Therefore we can reject the null hypothesis for the filter operation on large document sets.

Less clear-cut and consistent results could be found for the medium and small document sets. Regarding medium sized sets, the null hypothesis can be rejected for all cases but for the IBC version. While as far as the small document set is concerned, in general the null hypothesis could not be rejected except for the combined refactoring version (ALL).

Table 33. Summary of results

Document set size	App version	Search						Filter					
		Mean	**Sd**	**Delta means**	**Median**	**Delta median**	**p-val**	**Mean**	**Sd**	**Delta means**	**Median**	**Delta median**	**p-val**
Small	OR	16.71	12.74		16			9.97	7.70		15		
	DS	21.06	19.33	-4.35	16	0	0.819	9.49	7.86	0.45	15	0	0.55
	IBC	15.67	12.31	1.05	16	0	0.476	9.48	7.77	0.45	15	0	0.39
	UE	13.93	11.50	2.79	15.5	0	0.253	5.84	7.67	4.09	0	15	0.02
	ALL	12.52	16.19	4.20	0	1	0.062	5.22	7.50	4.71	0	15	**0.01**
Medium	OR	138.35	16.91		140.5			20.77	7.59		16		
	DS	66.15	15.45	77.50	63	78	**<0.001**	13.80	8.24	6.97	16	0	**0.01**
	IBC	73.43	13.15	62.50	78	63	**<0.001**	14.26	8.82	6.50	16	0	0.02
	UE	68.23	15.65	77.50	63	63	**<0.001**	10.85	7.38	9.92	15	1	**<0.001**
	ALL	59.09	11.40	78.50	62	78	**<0.001**	10.48	7.54	10.28	15	1	**<0.001**
Large	OR	225.69	15.89		219			31.19	0.40		31		
	DS	99.35	28.00	126.33	94	125	**<0.001**	17.06	11.03	14.13	16	15	**<0.001**
	IBC	111.40	32.59	114.29	109	110	**<0.001**	17.06	8.01	14.13	16	15	**<0.001**
	UE	101.03	16.34	124.65	94	125	**<0.001**	13.86	5.07	17.33	16	15	**<0.001**
	ALL	95.70	17.82	129.99	94	125	**<0.001**	12.36	6.58	18.83	16	15	**<0.001**

E. Discussion

The design of this experiment allows us, starting from observed differences in execution time, to draw conclusion about the time efficiency impact of detected code issues – the main construct of the design –.

For the large document set we observed a significant search time difference ranging between 114 and 130 ms (delta of the means) between OR and DS, IBC and UE versions. Considering the number of documents in the set and hypothetically assuming a cumulative time gain per document, such difference means that issues are responsible for around 1ms of delay per retrieved document.

Concerning the filter operation, the differences of the means range between 14 and 19ms. That corresponds to a delay per document of 1ms up to 1.5ms per filtered document.

The above differences are reduced for medium-sized documents sets, and for the filter operation and the IBS issue it is not statistically significant.

No statistical significance, except in one case, could be observed for the small-sized document set.

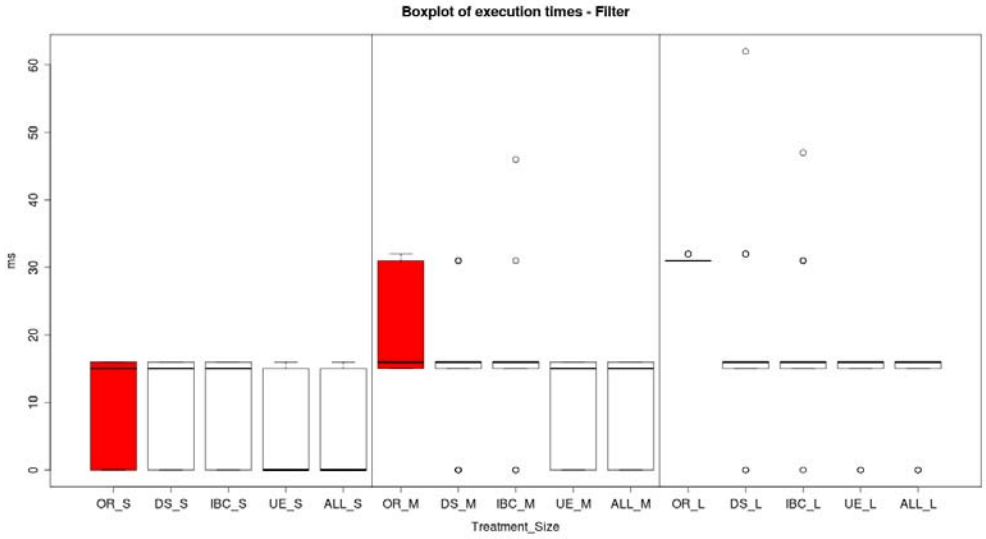

Figure 35. Boxplot of execution times. Filter

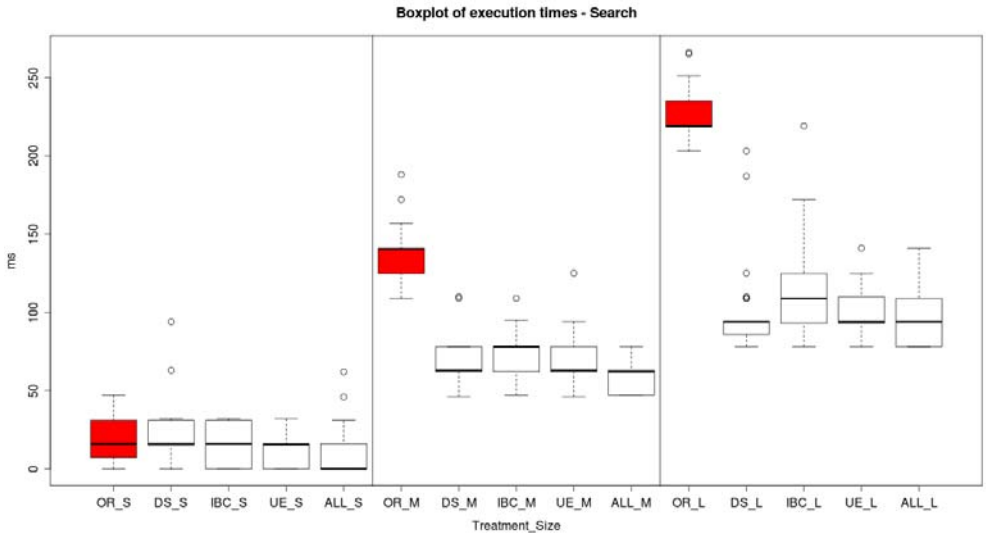

Figure 36. Boxplot of execution times. Search

To summarize, we observe an impact of FindBugs issues on time efficiency in medium and large searches (number of documents retrieved ≥ 100) and large filters (number of documents to filter ≥179).

Given an estimated impact of issues in the order of a millisecond per document, practically significant time efficiency degradation can occur when hundreds, thousands or millions of issue invocations take place (for example, inside loops).

Moreover, the speed of web application is an important characteristic because it might deeply affect the user experience. In fact, it is experimentally estimated ([105], [106], [107] and [108]) that:

- 0.1 second is about the limit for having the user feel that the system is reacting instantaneously
- 1 second is about the limit for the user's flow of thought to stay uninterrupted, even though the user will notice the delay.

So delays should be between 0.1 and 1.0 seconds to avoid negative impact on the user experience.

In our experiment, the difference between the refactored and the original version was up to 130ms for the search operations, i.e. beyond the 0.1 limit of the user perception. Considering that issues are responsible for a delay of about 1 ms per retrieved document, when the document set is more than 1 thousands of documents the overall delay becomes ≥ 1 second. The combined effects of issues deserve some considerations: removing all issues together (ALL) results into a speed-up that is smaller than the sum of the individual speed-ups achievable by removing individual issues. In practice we observe a sub-linear sum of delays when we remove all the occurrences of the three issues.

We conclude this discussion asserting that, based on our results, not all the inefficiencies of the source code can be mitigated through compilers and virtual machines optimizations, as also found by Lau et al. [94]. As a consequence, bad programming practices can lead to performance inefficiencies and ought to be refactored. From this perspective, we see two important applications of Automatic Static Analysis to code development:

- it can drive programmers to develop a more efficient code and to refactor existing code;

- it can be applied to code generation tools to produce a more efficient code: as a matter of fact, 15 out of the 26 issues detections of the experiment were located in code automatically generated.

In both cases, we see the strong need for triaging and empirical evaluation of issues that may have a negative impact on performance.

RELATED WORK

We summarize in this section the related work in the field of Java optimization (A), Automatic Static Analysis (B) and execution time measurement (C). The majority of discussion is devoted to the first topic: we describe the several approaches commonly adopted to optimize Java code, listing their benefits and drawbacks and motivating our choice to focus on HLR level (i.e, source code level) to optimize Java programs.

A. Java optimization

1) Static optimization

The first approach to improve Java code is static, i.e. before execution. Kazi et al. [92] conducted a survey on this topic: they present the transformation of Java byte code to an intermediate source code as one of the first attempts to statically optimize code (e.g. [109], [110]): unfortunately this technique does not guarantee portability. For this reason, other techniques were developed without impairing portability. For example, it is possible to apply standard compiler optimizations directly to the Java byte code (e.g. [111] [112]): however, only a limited number of optimization techniques can be applied to byte code, because the whole program structure is not available at this level. The two approaches presented above share a common drawback: they are all based on performance predictions. Moreover, if the code is written in an inefficient way, the static optimizations are not powerful make the program efficient. Lau et al. [94] faced this problem, realizing an experiment that demonstrated the possible inefficiency of such techniques. They modified the IBMs J9 JIT compiler to measure the total number of cycles spent in a given method.

Afterwards they designed an experiment with the goal of evaluating the impact of optimizations on 101 methods selected from 20 benchmarks. The methods were compiled twice, with the two highest optimization levels of the J9 JIT compiler (O4 and O5). Subsequently, they ran them and compared the execution times of these two versions, obtaining that the highest level (O5) offered substantial speedups only for a small number of methods, whereas modest effects were observed on the majority of methods. Moreover, level O5 actually degraded performances relative to O4 for about a third of the methods, with a negative pick of -21%. Then, the authors repeated the experiment focusing only on a specific optimization technique, i.e. method inlining. They adopted a subset of the methods used in the former experiment, comparing the execution time of the methods normally compiled and the execution time of the methods compiled with inlining optimization. They obtained that 7 methods were improved by over 10%, but 5 methods degraded by over 10%. These results demonstrate that static optimizations work for the average case, and they do not assure that code will be actually faster. The possible motivations are two: overhead introduced by the common services of VMs (e.g.: automatic memory and thread management, dynamic loading, and so on) and limits of static optimizations.

Our approach is also static but, being located at HLR, limits the two main flip sides highlighted above: it guarantees portability and it works at the very first level of the code.

2)Dynamic optimization

The second optimization approach that we present is dynamic, i.e. during program execution. Techniques belonging to this approach dynamically translate Java bytecode into native machine code (this is the case of JIT [113] compilation techniques) using adaptive runtime mechanisms that are executed by the virtual machine during program execution.

Arnold et al. reviewed adaptive optimization technologies [90]. The goal of adaptive optimization technologies is to improve performance by monitoring a program's behavior and using this information to drive optimization decisions. Arnold et al. [90] identified three main categories: 1) selective optimization, 2) profiling techniques for feedback-directed optimization (FDO) and 3) feedback-directed code generation. With

selective optimization (1) the VM determines at runtime which parts of the program should be optimized: this solution was firstly proposed by Hansen [114], afterwards many others have implemented the concepts of his doctoral work. For instance the SELF-93 implementation [115] applied many of Hansens' techniques, whereas the SELF-91 [116] was the base for many techniques of the HotSpot Server VM [117].

FDO techniques (2) are focused on the collection of profiling information. Smith [118] identified three advantages for these techniques: 1) they use dynamic information that cannot be inferred by static optimizer technologies, 2) they enable the system to change and revert decisions if conditions change, and 3) runtime binding allows more flexible and easy-to change software systems. Moreover, FDO can be implemented using different profiling techniques (runtime service monitoring, hardware performance monitors, sampling, and program instrumentation). The last pool of techniques is the feedback-directed code generation (3), whose aim is to improve the quality of the code generated by an optimizing compiler using different feedbacks. Among these three categories, selective optimization (1) has obtained a major impact on production systems, serving as a core technology in many production VMs.

Dynamic optimization solves many limitations of compiler optimization, but they add overhead during software execution. Moreover, many profiling-based optimizations can work only on a subset of the possible application inputs.

3)Various techniques

We also found a pool of others techniques, which we list here because they are applicable only in specific environments.

A first example is the parallelization of loops or recursive procedures ([119], [120]). This static approach produces high benefits only on source code with a significant amount of parallelism.

A second approach is the development of ad-hoc Java processors (e.g., [121] [122] [123] [124]) that execute the Java byte code as their native instruction set. This solution permits to highly improve the time efficiency of programs, but the cost to pay is a processor

optimized just for Java, therefore other programming languages achieve poor efficiency on them.

Higher performance is also achievable by focusing on those JVM features such as garbage collection, exception handling or thread synchronization that are the bottlenecks of the execution time because they add overhead (for instance [125]).

A totally different set of problems arises when Java is executed in parallel and distributed environments [126]. The bottleneck in this case is the performance of the message passing, multithreading and synchronization mechanisms.

The few approaches listed in this section do not represent a comprehensive list of all various techniques. However, this small set is sufficient to draw the following consideration: though very efficient ad-hoc solutions are present in the literature and are applied in industry, they work very well only for very specific requirements and types of applications, and they are not generalizable.

4)High level optimization

Little effort is dedicated to the exploration of techniques to optimize HLR, i.e. the code written by programmers or generated through modeling systems. Many books on Java source code optimization can be found in the literature, but no empirical evaluations that assess and quantify the benefits of applying optimization at HLR level. Kernighan [127] reserved a chapter of his book to performance, identifying possible bottleneck in programs and suggesting simple profiling mechanisms and strategies to tune programs. Few years later, Shirazi [128] offered to programmers a wider collection of tips about efficient programming, related to Object creation and reuse, efficient use of Strings, Exceptions, variables, or suggestions on how to tune loops. The author also provided some comparisons of timings and performance tuning checklists for developers.

Other books dedicated to the performance problem from a programmer perspective are written by Wilson and Kesselman [129] and Bentley [130]. Moreover, there is a lot of work in the "grey" literature (for instance [131] [132] [133]), but no evaluations are provided.

This work provides a quantitative assessment of the impact of HRL optimizations detectable by automatic static analysis. Our former study [134] was conducted with a similar goal, but in a laboratory setting. In that work 20 FindBugs issues were selected and for each of them two source code fragments were prepared: one containing the issue and the corresponding refactored version, functionally identical but without the issue. Then three different platforms, isolated from network and other user programs to limit overhead due the execution environment, were used to execute a large number of times the code fragments measuring the execution time of both code versions. The authors found that eleven issues have an actual negative impact on performance in all platforms (up to 6 times slower). Among the eleven issues, the inefficient boolean constructor (IBC) was also proved to degrade the time efficiency. The dead store DS had a negative impact on two platforms over three. The useless exception (UE) was not tested.

The methodology used in this paper is similar; however we proved the effect of code issues in a real industrial application, with real usage scenarios.

B. Measurement of execution time

The efficiency of a software function, in terms of time behavior, is evaluated by computing or measuring the execution time. Measuring the execution time is not trivial because it is not deterministic, but it has a certain variation. In fact, the program behavior can be modeled with the following equation [135]:

Program Behaviors = Inputs + Code + Environments. The running environments consist of all the elements in the execution platform (architecture, virtual machine, parallel threads or processes, operating system, and so on); program inputs determine the path executed or data to be accessed. Finally, the program code determines the set of instructions that may be executed in a path. The variability coming from software (operating system, programs, processes and threads) could make execution times unpredictable. For this reason, there is a best-case execution time (BCET), i.e. the shortest possible execution time, and a worst-case execution time (WCET), i.e. the longest possible execution time. It is possible to use two techniques to determine the BCET and the WCET [83]. The first approach is the use of static methods: the code and the possible paths are analyzed and,

combining different techniques, upper and lower bounds for the execution time are provided. This methodology does not take into account the hardware and the platform on which the code is executed, hence the bounds overestimate the WCET and underestimate the BCET and the variability is determined only by input data and code. The second method is measurement-based: the code, or a portion of it, is executed on a given hardware or a simulator for a set of inputs, and the pair WCET/BCET is obtained from direct observation. This method provides estimates and not bounds, underestimating the WCET and overestimating the BCET. Despite the high number of techniques developed for both approaches, the problem for WCET analysis in the field of Java applications has not been deeply examined yet: Harmon and Klefstad conducted a survey of WCET analysis for Real-Time Java [84], but they were able to find fewer than twenty publications addressing the problem, and typical issues like the precision of measures and their generalizability are still open.

We adopted in this work a measurement-based approach to measure the execution time, using a profiling tool.

4.4. CONCLUSIONS

In the first experiment, we set up an experiment to quantitatively assess the impact of selected FindBugs issues on time efficiency. We selected, through expert judgments, 20 representative issues and for each one we compared the average execution time of a code fragment containing that issue against the same code with the issue refactored out. The measurements were conducted on three different platforms.

Experts' examination of issues revealed that, mainly because FindBugs issue taxonomy is exclusive, a few issues having a potential impact on performance in fact do not belong to the Performance category. Moreover experiment revealed that 3 out of 11 verified issues do not belong to the Performance category, while 2 out of 3 unverified issues belong – apparently without justification – to that category. Overall, based on our findings we can select 11 issues that have a proved and quantified impact on time efficiency

The second experiment was performed with an industrial application. We quantitatively assessed the impact on performance of three code patterns identified as issues by FindBugs. The issues are: dead store to local variable, useless try-catch block and inefficient construction of Boolean objects. We instrumented an industrial Java web application on two commonly executed operations: a search that retrieves documents from a remote repository, and a filter that works on the output of the search. A co-factor included in the experiment design is the document set size, for which we identified three levels: small, medium and large number of documents retrieved.

We observe a significant effect of issues on searches on medium and large document sets, and on filter on large documents sets. The estimated overall delays ranged from 19 ms (filter) to 130 ms (search), and the execution time of the refactored code version was up to two times faster. Moreover, the delay inserted in large searches is beyond the threshold of the user perception of 100ms, thus implying a perceivable effect. We also observed that issues are responsible from 1 ms to 1.5 ms of delay per retrieved document: the time efficiency degradation for thousands of documents has a noticeable impact.

The information we provide is useful to practitioners to estimate potential delays introduced by the three code patterns in similar applications and to tune the ASA tools to identify and refactor them on source code.

In both experiments, results proved that bad programming practices can lead to significant performance degradation. Automatic Static Analysis can play an important role in this scenario, either driving programmers or tuning code generation tools to produce more efficient code.

5 FUTURE RESEARCH CHALLENGES

This chapter is devoted to the future research challenges for Automatic Static Analysis.

The first challenge is ASA for multi-language projects. Most software systems are complex and composed of a large number of artifacts. To realize each different artifact specific techniques are used resorting to different abstractions, languages and tools. Successful composition of different elements requires coherence among them. Unfortunately constraints between artifacts written in different languages are usually not formally expressed nor checked by automatic static analysis tools; as a consequence they can be a source of problems. We explore the role of the relations between artifacts written in different languages by means of a case study on the Hadoop open source project, in which we quantify the phenomenon and investigate its relation with defect proneness.

The second identified challenge is ASA for energy efficiency.

The assessment of the impact of software over IT power consumption is still at its initial phase. However, recent works suggest how this contribution might be even more significant as the hardware evolves towards more powerful and scalable architectures. Thus, optimizing software in terms of energy efficiency is one of the challenges that both research and industry will have to face in the next few years. Previous works have shown how software optimization might be achieved through identifying and refactoring code patterns that negatively impact a certain software characteristic, such as maintainability or efficiency. Those code patterns have been defined as Code Smells. If we consider energy efficiency as a software characteristic, we can apply the same idea of Code Smells identifying those code patterns, hereby defined as Energy Code Smells, which might increase the impact of software over power consumption.

5.1. LANGUAGE INTERACTION AND QUALITY ISSUES: AN EXPLORATORY STUDY

Most software projects nowadays are polyglot, i.e. files written using different languages interact with each other. Wampler et al. [136] introduced a special issue on this topic writing "Most teams are by necessity MPP [Multi-Paradigm programming] teams now. No one writes in a single language anymore. Even trivial applications have a general-purpose language, SQL, JavaScript, CSS, and dozens of frameworks, each of which includes an external DSL [Domain Specific Language] (usually in XML) that is its own mini language (the syntax is XML, but the XMLSchema defines the semantics)".

Given this scenario we seek to study the effects of language interaction and eventually evolve development techniques and supporting tools to consider these aspects. Nowadays tools used by developers help them only to verify the consistency internal to a language, i.e. consistency within a set of artifacts written in the same language. For example, editors check that an expression in Java code invokes a Java method which exists in the codebase, either in the same file or in another Java file. On the other hand there are major limitations in verifying the consistency across the language boundaries. For example can tools help the developer to understand immediately if a piece of XML code used for configuration refers to a really existing Java class? Normally currently available tools cannot do this because they are not aware of the cross-language semantics.

While the issue of language interaction is already very relevant today, the appearance of language workbenches [137] let us suppose that this issue is going to become even more important in the future. For example, with Xtext [138] and GMF [139] we can create, textual and graphical DSLs with custom editors integrated in the Eclipse platform with a minimal effort. Other tools like Intentional Software [140] and the Meta-Programming System [141] fully support the Language Oriented Programming paradigm [142] and are based on projectional editing. The existence of these tools and their usage in industrial projects [143] seem to indicate that the interaction between languages in projects will increase in the future.

Pfeiffer et al. [144] conducted a study related to language interaction. They realized a tool named *GenDeMoG* to mine inter-languages interaction based on text analysis. Their work was motivated by observing the amount of errors introduced by undocumented relations that cross the language border (i.e., they involve modules written in different languages) and the resulting complexity.

Our hypothesis is that in the long run we need to support cross language development, including design, modeling, and validation. To reach this goal we first need to start understanding the effects of languages interaction: this work is intended as a first step in that direction.

DEFINITIONS

Before stating our goals and translating them into actionable research questions, we define how we do identify and measure the languages interaction. We provide here a list of definitions used throughout the rest of the paper.

Module: we considered a module each single file.

We consider a commit[16] as a unit of work, consequently we suppose that files committed together are related.

Intra-language commit (*ILC*): a commit containing a set of modules with the same extension.

Cross-language commit (*CLC*): a commit containing modules with different extensions.

Cross-language commit for an extension (CLC_{ext}): a CLC containing that includes modules with the extension *ext*.

Defect fix: a commit executed to fix a defect.

We consider a module to be *cross language* when it is related to modules written in a different language (e.g., a Java file loading the configuration from an XML file). To measure how much a module is cross language we analyze its history: if the module was

[16] We refer to the term commit as used in the context of version control systems.

frequently committed with files written in other languages we consider that as an indicator of interaction between the module and those files. This interaction is measured through different variants of the *cross language ratio* (CLR).

Cross language ratio of a module ($\boldsymbol{CLR_m}$): the CLR of a module *m* is the fraction of cross-language commits in which *m* was involved with regard to the total number of commits regarding the module (both intra-language and cross-language):

$$CLR_m = \frac{\#\ CLC}{\#\ CLC + \#\ ILC}$$

Cross language ratio of a module with regard to an extension ($\boldsymbol{CLR_{m,ext}}$): the CLR of a module *m* considering as CLC only the commits involving *m* and a module with extension *ext*:

$$CLR_{m,ext} = \frac{\#\ CLC_{ext}}{\#\ CLC_{ext} + \#\ ILC}$$

Cross language ratio of an extension ($\boldsymbol{CLR_{ext}}$): for each extension *ext* we compute its cross language ratio as the mean of the CLR_m considering all modules having extension *ext*:

$$CLR_{ext} = \frac{\sum CLR_m\ , m \in ext}{\#\ *.ext}$$

Cross language ratio of an extension *extA* with respect to an extension *extB* ($\boldsymbol{CLR_{extA,extB}}$): the mean of $CLR_{m,extB}$ among all modules *m* with extension *extA*:

$$CLR_{extA,extB} = \frac{\sum CLR_{m,extB}\ , m \in extA}{\#\ *.extA}$$

Cross Language Module (CLM): a module is cross language if its CLR is ≥ t_{CLM}%, where t_{CLM} is a threshold to be defined.

Intra Language Modules (ILM): a module is intra language if its CLR is < t_{ILM}%, where t_{ILM} is a threshold to be defined.

GOALS, RESEARCH QUESTIONS AND METRICS

The goal of this preliminary study is two-fold. Firstly we investigate the level of languages interaction in a common project. Secondly, we verify whether the level of interaction is related to quality problems. We look at defects as a proxy of software external quality. We identify two research questions related to the first goal.

RQ1 How much interaction is there among the languages present in a project?

The interaction is computed as the percentage of CLC among a set of commits. First we consider all type of commits (RQ1.1), then (RQ1.2) we consider separately the commits related to a particular activity (e.g., improvement, bug fixing, new feature).

Once we have defined the size of the phenomenon by answering to RQ1, we will go deeper considering the behavior of each single extension.

RQ2 Which extensions interact more?

The second research question is answered at two levels, i.e. firstly investigating the relationship between one extension versus all the other extensions (RQ2.1), then analyzing the most interacting pairs of extensions (RQ2.2).

We answer RQ2.1 computing the CLR_{ext} for each extension, while we answer RQ 2.2 computing the $CLR_{extA,extB}$ for all pairs of extensions.

The last research question is related to the second goal, i.e. investigating whether a high interaction between languages might result in higher defect proneness.

RQ3 Are Cross Language Modules more defect-prone?

We answer RQ 3 computing the number of Cross Language Modules (CLM) with and without defects, and the number of Intra Language Modules (ILM) also with and without defects. Then we compare the two proportions with/without defects by means of the F-test to see whether the proportion of Cross Language Modules with defects is different from the one of Intra Language modules.

This metric is computed at three granularity levels:

- considering all files regardless of their extension (RQ3.1),

- considering for each single extension its level of interaction with all the other extensions as aggregate (RQ3.2),
- considering interaction between specific ordered pairs of extensions (RQ3.3).

CASE STUDY

This exploratory study aims at understanding the phenomenon of language interaction and derived quality issues. We also use it to investigate whether the methodology defined above is applicable. We selected as a case study Apache Hadoop[17], which is a set of libraries to support distributed data processing. We selected Hadoop because it is a mature project (it is supported since April 2006) and it is used in many industrial applications (e.g., Yahoo, and Facebook).

Our methodology for computing the metrics defined above is based upon the fact that Hadoop uses SVN[18] to manage artifacts versions and JIRA[19] to track not only defects but any other activity that can be associated with software artifacts. Those elements are called "JIRA issues", and each project has its own set of issues. Example of JIRA issues are the implementation of a new feature, a single implementation task, a bug report, and so on. Hadoop developers established links between commits in the SVN code repository to JIRA issues by systematically including issue ids in their SVN commit comments.

We downloaded the SVN log from the Hadoop repository (last revision retrieved is the 1233090, from 01/18/2012, the first available revision is the 776174 from 5/19/2009). We also extracted all JIRA issues from the Apache JIRA database.

We computed all modules CLR_m and observed their distribution: about 30% of modules have CLR_m between 0 and 0.1, and about 55% files have CLR_m between 0.9 and 1. Given these percentage and given that the remaining files have a positive (right) skewed distribution, we decided to use as thresholds $t_{CLM}=t_{ILM}=50\%$ to define CLM and ILM modules.

[17] http://hadoop.apache.org
[18] http://subversion.tigris.org/
[19] http://www.atlassian.com/software/jira/overview

RESULTS AND DISCUSSION

Table 34 reports the percentage of cross language commits in the Hadoop repository: 53% of all commits (first column) are CLC, i.e. containing files of different languages. Looking at the portion of CLC related to the different activities (i.e., JIRA issues), we observe that their percentage varies with respect to the type of issue (from 2nd to last column in Table 34). It goes from a minimum of 5% in commits related to Test up to a maximum of 45% in Sub Tasks (since not all issues are linked to JIRA issues, the mean "All" in the first column is not related to the other means in the following columns).

RQ 1.1 answer: *the 53% of commits in Hadoop are cross language.*

RQ 1.2 answer: looking at the single activities, we derive that writing/modifying tests or fixing bugs are activities that involve mainly a single language, while adding new features is an activity that involves multiple types (or at least extensions).

We now proceed to RQ 2.1 and 2.2. Table 35 contains the top 5 extensions in terms of number of files: c, sh, properties, xml and java. Among them, four extensions correspond to programming languages and one is used for configuration files. Subsequently, we compute the $CLR_{extA,extB}$ for all combinations of the five extensions . Table 36 reports the $CLR_{extA,extB}$.

RQ 2.1 answer: all most common extensions in Hadoop are highly interacting with other extensions (i.e., CLR_{ext}, > 0.50).

RQ2.2 answer: *the most frequent interactions ($CLR_{extA,extB} \geq 0.50$) are: C-XML (0.83), Properties-Java (0.54), XML-Java (0.52), C-Java (0.51), C-sh(0.50). Border values are: Java-XML (0.48), sh-XML (0.47) Properties-XML (0.46), and XML-Properties (0.43).*

We observe that the only pairs with frequent interactions in both directions are Java-XML and Properties-XML. All the other pairs have frequent interactions in only one direction. For instance, CLR_{XML-C} = 0.04 and CLR_{C-XML}=0.83 means that most of the commits involving C contain also XML files, but not the other way around.

Table 34. Percentage of cross language commits (RQ 1)

All	Bug	Improvement	New Feature	Sub task	Task	Test
0.53	**0.12**	**0.26**	**0.30**	**0.45**	**0.26**	**0.05**

Table 35. CLRext (RQ 2.1)

CLR_{ext}	Nr files	Extension
0.96	**49**	**c**
0.87	**114**	**sh**
0.72	**75**	**properties**
0.71	**320**	**xml**
0.59	**4328**	**java**

Table 36. CLR_(extA,extB) (RQ 2.2)

extA/extB	C	Java	Properties	Sh	XML
C	-	0.51	0.10	0.50	0.83
Java	0.01	-	0.28	0.04	0.48
Properties	0	0.54	-	0.36	0.46
Sh	0.09	0.22	0.24	-	0.47
Xml	0.04	0.52	0.43	0.24	-

Table 37. Odds ratio of the defectivity in respect to the relation between pairs of extensions (RQ 3.3)

	C	Java	Properties	sh	XML
C	**-**	**Inf**	**0**	**0**	**Inf**
Java	2.79	**-**	0.32	0.43	**0.96**
Properties	**Inf**	**1**	**-**	12.08	**0.94**
Sh	**3.55**	4.45	17.17	**-**	7.44
Xml	3.83	**0.95**	3.22	4.73	**-**

We now focus on the last RQ, i.e. on the relation between languages interaction and defect proneness. Table 38 contains metrics to answer RQ 3.1 (first line) and RQ 3.2

(from 2nd to last line). The following columns contain, in the order: the number of ILM with no defects and then with at least one defect, the number of CLM with no defects and then with at least one defect, the p-value of the F-test and finally the odds ratios (which is greater than 1 when CLM are more defect prone than ILM).

RQ 3.1 answer: considering all extensions, ILM are more defect prone that CLM (about 5 times less).

RQ 3.2 answer: considering the five most common extensions, we observe that three extensions (XML, Properties and C) have CLM with higher defect proneness, while two extensions (Java and Sh) exhibit the opposite relation.

Among the above differences, only *all extensions* and *Java* are statistically significant (p-value ≤ 0.05).

Finally, Table 37 contains the odds for each pair of extensions to answer to RQ 3.3. We report in bold the values for which we obtained a p-value ≤ 0.05. We observe 7 pairs for which ILM are less defect prone than CLM, 12 pairs with CLM more defect prone than ILM and one pair with odds ratio =1. We consider only values with p-value ≤ 0.05 to answer RQ 3.3.

RQ 3.3 answer:

- *four extension pairs have CLM more defect prone then ILM (C-Java, C-XML, Properties-C, Sh-C),*
- *five extension pairs have ILM more defect prone then CLM (C-Properties, C-sh, Java-XML, Properties-XML, XML-Java)*

Table 38. Relation between classification in ILM and CLM and presence of defects (RQ 3.1 and 3.2)

	RQ	MN	MY	CN	CY	P	Odds
all	**2**	**1891**	**225**	**2875**	**89**	**0.000**	**0.26**
c	**2.1**	**2**	**0**	**46**	**1**	**1.000**	**Inf**
java	**2.1**	**1692**	**201**	**2239**	**25**	**0.000**	**0.09**
properties	**2.1**	**19**	**1**	**45**	**7**	**0.429**	**2.92**
sh	**2.1**	**10**	**5**	**64**	**13**	**0.162**	**0.41**
xml	**2.1**	**96**	**11**	**184**	**24**	**0.851**	**1.14**

- *one extension pair have exactly same defect proneness (Properties-Java).*

We notice that interactions where CLM results more defect prone involve always the C files. While interactions where ILM results more defect prone involve mainly XML, however C is also present. An interesting fact is that the pair Sh-C is in the first set, the pair C-sh is in the second. Besides these considerations, we do not have an unique answer for RQ3. However, we observe that having languages interacting with other languages is related to higher defect proneness for certain languages (mainly C) and specific interactions.

THREATS TO VALIDITY

Internal: in this exploratory case-study different aspects were not considered. In particular we did not examine all the possible confounding factors influencing the defect proneness of the modules. Among them the age and the size of modules (expressed in LOC, for example) are the most relevant ones.

We discriminated between modules on their names while the same module can change name in the course of the project. We grouped the files by their extension while a different extension could not always indicate a different language.

Construction: we are unable to measure directly the interaction between modules written in different languages and consequently we use as a proxy their concurrent presence in the same commits, which may be an imprecise approximation.

External: another threat is due to selection bias: we have no particular reason to believe that Hadoop is representative of other software projects. Of course having considered only one project generalization of the results presented is not possible at all.

CONCLUSIONS AND FUTURE WORK

Although we do not have unique answers, the results and observations from this exploratory study let us understand that the problem is worthy to be investigated. In fact we observed that more than half of the commits in Hadoop are cross language (at least according to our definition). However we also observed that this property depends on the type of the activities and the extensions of the modules.

Commits related to testing or fixing bugs involve mainly a single language, while adding new features or doing implementation sub-task are activities which involve multiple languages (or at least extensions).

Looking at the single extensions, we verified that the most common extensions are frequently changed together with files with different extensions. Frequent interactions are generally not symmetric, and many of them involve XML.

When we look at defect proneness, we observe that for Java modules the interactions with other languages (as an aggregate) is not problematic at all: we observed that Java CLMs files are ten times less defect prone than ILMs. However, when looking at single pairs of interactions, we notice that several pairs have CLM significantly more defect prone then ILM, especially C modules. Finally, the widespread interaction between Java and XML apparently is not related to defect proneness.

Today Automatic Static Analysis tools cannot cross the boundaries of the languages and are not able to analyze the interactions between languages [2]. This study represents a first step in understanding the phenomenon of languages interaction and it let us hypothesize that the interaction of languages might be problematic for specific languages interactions.

The new challenge for Automatic Static Analysis is to become polyglot.

5.2. DEFINITION, IMPLEMENTATION AND VALIDATION OF ENERGY CODE SMELLS

The issue of sustainability is starting to be addressed among the software engineering community. Although during the First International Workshop on Green and Sustainable Software[20] there was a common agreement that sustainability is and will be a key aspect of software, it is still unclear how to design sustainable software. While for other characteristics (reliability, performance, security, etc.) processes and metrics have been proposed and widely investigated by the SE community, as regards sustainability the discussion is still in its initial phase. In addition to that, software sustainability has an intrinsic difficulty because the topic invests not only technological aspects, but also economic, social and environmental, which are under the broad umbrella of sustainability as defined in 1987 by the Bruntland commission .

Among the kaleidoscope of aspects related to software sustainability, one of the most visible is the energy (or, alternatively, power) consumption of software systems. Indeed, software does not consume energy directly, however it has a direct influence on the energy consumption of the hardware underneath. In fact, applications and operating systems indicate how the information is processed and, consequently, drive the hardware behaviour: previous work [145] suggested that software can increase the total power consumption of a computer system up to 10%. This and other initial findings [146] open investigation spaces on the optimization of energy and power consumption of IT devices acting on the software instead of the hardware. Moreover, nowadays the same software runs on multiple devices, thus it might be more productive and feasible for software houses to green the single software rather than relying on the greening of all the hardware implementations underneath (that could require competences commonly not owned by software houses).

Optimizing a software product in terms of energy efficiency has also some issues. The absence of a standard procedure, or a benchmark, to compare systems is the most

[20] http://greens.cs.vu.nl/?page_id=1262

prominent one. This is because software is intangible and it is deployed on devices with their own specifications and features. This makes really difficult to standardize a transparent, platform-independent measuring system for every software system.

Another consideration must be done regarding software architectures. During the last years, software engineers always tried to increase the number of software layers - that is, for improving interoperability, abstraction, decoupling, etc. However, the steep increase of software layers directed the optimization efforts only on each layer ("horizontal" optimization) and not across them ("vertical" optimization). Since energy efficiency directly relates with hardware technologies, a more intense communication flow between hardware and software is needed to achieve significant optimizations. In this sense, embedded systems make a perfect case study, because their architecture is simplified by design, and also because power consumption issues acquire a peculiar importance, for operational reasons (most embedded systems are battery-powered).

For this reason our work uses an embedded system as the test-bed to validate a new approach for the design and implementation of sustainable software. We investigate, and here we also introduce the goal and main contribution of this study, how software can be optimized by identifying code patterns that use in a sub-optimal way the hardware resources. These code patterns ought to be refactored in order to improve the energy efficiency of the software at run time. We define and name the code patterns Energy Code Smells, inspired by the well-known book of Fowler and Beck [62].

GREEN CODE SMELLS: BACKGROUND AND DEFINITION

The term "code smells" was coined by Fowler and Beck [62] referring to poor implementation choices that make the software difficult to maintain. These bad implementation practices can be characterized as patterns in source code. For instance, the smell "Long Method" refers to a method that has grown too large: typically, the longer is the method the more difficult is to maintain it. One or more refactoring actions are associated to code smells: for example all you have to do to refactor a Long Method is to

extract parts of the method that seem to go nicely together and make a new method. As a result the original method is shorter and easier to maintain. The goal of identifying and refactoring code smells is to make the code more understandable and flexible to evolution, i.e. more maintainable, and many studies in the literature have been devoted to this aspect [69] [147]. However refactoring code smells might have an effect also on other properties of the software, such as the portability, the testability or, as in the case of this work, the energy efficiency of the code. As a consequence, we take inspiration by the original work of Fowler and Beck and we introduce the concept of smells into the Green IT community, introducing the Green Smells:

A Green Smell is an implementation choice that makes the software execution less energy efficient.

Since software has different levels of abstractions and organizations, Green Smells can be located at code, design or architectural level. Therefore, Green Code Smells are implementation choices at source code level (code patterns) that make a sub-optimal usage of the hardware resources underneath. As a consequence, they provoke a higher energy (or alternatively, power) consumption.

VALIDATION OF ENERGY CODE SMELLS

The aim of our research is to identify Energy Code Smells. In addition to that, we are also interested in understanding whether the Energy Code Smells also degrade the performances of the application in terms of execution time. We set up two research questions for our investigation:

RQ1. Which code patterns have an effect on power consumption (i.e. which code patterns are Energy Code Smells)?

RQ2. Code smells that have an effect on execution time do also have an effect on energy consumption (i.e. are Energy Code Smells also Performance Smells) ?

The epistemological approach adopted for this research is the empirical one. We set up an experiment observing two dependent variables: power consumption (W) for RQ1

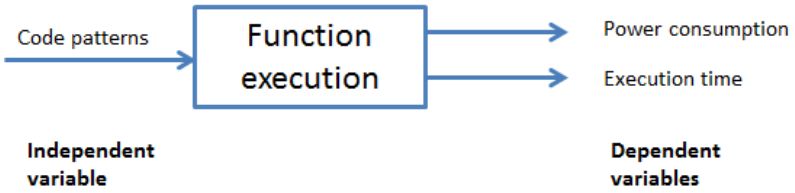

Figure 37. Experiment design

and execution time (ms) for RQ2. The two dependent variables are measured on the execution of C++ functions running on an embedded device. The choice of the embedded device has several advantages, the main two being:

- it has no operating system and thus confounding factors in the experiment are minimized;
- it runs on a battery and it really needs energy efficient code.

In other terms, refactoring Energy Code Smells in such an environment might lengthen the life of the battery.

The potential Energy Code Smells selected for the experiment are code patterns used by two popular static analysis tools. For each code pattern selected for the experiment, we set up a C++ function with two implementations, one that violates the code pattern (thus contains a Code Smell) and the refactored one without the violation. Therefore the treatment is the refactoring of the smell and it is possible to observe an effect on the two variables by comparing the measurements on the two versions of the code. Figure 37 represents the design described.

POTENTIAL ENERGY CODE SMELLS SELECTION

As introduced above, the software that runs on the selected device is C++ code. In order to identify Energy Code Smells on C++ code we look at already existing code patterns. In particular, we examined patterns implemented by Automatic Static Analysis

(ASA) tools. The two tools selected for this study are Cpp-Check[21] and Findbugs[22]. CppCheck is a well-known static analysis tool for C/C++ which contains many patterns regarding a variety of desired software properties: safety, portability, performance, etc . An example of C/C++ pattern on portability is "*64 bits portability*", i.e. assign address to int or long. An example of checked pattern on performance is instead "*Address not taken*" of the category "Memory leaks", which detects when the address to allocated memory is not taken. In order to identify which patterns can be considered relevant for energy efficiency, two of the authors carefully read all patterns and selected independently which ones could cause a higher power consumption of the Waspmote. All conflicts (a pattern selected by

[21] http://cppcheck.sourceforge.net/, last visited on September 13th, 2012
[22] http://findbugs.sourceforge.net/, last visited on September 13th, 2012

only one expert) were resolved in a reconciliation meeting, where patterns were discussed and a final decision taken. In addition to the Cpp-Check patterns, we also reviewed the patterns of another static analysis tool, Findbugs. It is similar to Cpp-Check, but it analyzes Java code. The same two authors reviewed all FindBugs patterns and decided firstly if they can be applied to C++ code, then whether they might be related to energy efficiency. The selection process ended up with the patterns shown in Table 39.

Subsequently, we wrote for each of the patterns a pair of C++ functions, one

Table 39. Potential Energy Code Smells Selected for validation

Pattern Name	Pattern Description	Tool
Parameter By Value	Passing a parameter by value to a function	CppCheck
Self Assignment	Assignment of a variable to itself. (e.g., x=x).	CppCheck
Mutual Exclusion OR	OR operator between two mutually exclusive conditions (thus always evaluating to true).	CppCheck
Switch Redundant Assignment	Redundant assignment in a switch statement: for example, assigning a value to a variable in a case block without a following break instruction, then re-assigning another value to the same variable in the subsequent case block.	CppCheck
Dead Local Store	A statement assigning a value to a local variable, which is not read or used in any subsequent instruction.	FindBugs
Dead Local Store Return	A return statement assigning a value to a local variable, which is not read or used in any subsequent instruction. (i.e. return(x=1);)	FindBugs
Repeated Conditionals	A condition evaluated twice (e.g., x==0 — — x==0).	FindBugs
Non Short Circuit	Code using non-short-circuit logic boolean operators (e.g., & or |) rather than short-circuit logic ones (&& or ||). Non-short-circuit logic causes both sides of the expression to be evaluated even when the result can be inferred from knowing the left-hand side.	FindBugs
Useless Control Flow	Control flow constructs which do not modify the flow of the program, regardless of whether or not the branch is taken (e.g., an if statement with an empty body).	FindBugs

containing a potential smell and another one refactored without that smell. For example, the "Non-Short Circuit Logic" pattern has the following two functions:

```
void NonShortCircuit With(){
      int count = 0;
      int total = 345;
      if ( count > 0 & total / count > 80 )
      count=0;
}
void NonShortCircuit Without(){

      int count = 0;
      int total = 345;
      if ( count > 0 && total / count > 80 )
      count=0;
}
```

The function NonShortCircuit With() is the one with the potential smell "*Non-short circuit logic*". The smell is in the line *if*(*count*>0 & *total*/*count*>80) because the AND operator is single & and so both predicates in the expressions will be evaluated at run-time. In the function, NonShortCircuit Without() the code is refactored replacing & with &&. All functions are available online for the sake of replication[23].

EXPERIMENT SETUP

A. Context: the WASP

The device used for the experiment is the Waspmote V1.1 (Libelium Comunicaciones Distribuidas S.L. Esso). The hardware architecture is based on a ATmega 1281 microcontroller with a CPU frequency of 8 MHz and 8KB of SRAM. It has no operating system: programs are directly loaded on a FLASH memory of 128 K. This

[23] http:softeng.polito.it/greensmells/

architecture well suites our experiment because no other threads run in parallel with the chosen program, thus eliminating any software noise for the energy measurement. The device is basically a motherboard with connectors to plug in other elements such as sensors, wireless modules (ZigBee, XBee, Bluetooth), GSM/GPRS modules and a GPS (Global Positioning System) module. For this reason it is used in different fields, such as Smart Metering, Building Automation, Agriculture etc. It runs on a lithium battery (3.7V and 1150mAh), so the energy consumption of software has a key role here. To compile and load the C++ programs it is sufficient to use the IDE provided by the manufacturer and connect it to a computer via USB cable.

B. Experiment setup

The objective of the experiment is to measure power consumption and execution time on each function pair, in order to evaluate if the potential smell affects the two dependent variables. We divide the experiment in two parts: one for measuring power consumption, and another one for the execution time.

Measuring power consumption and execution time for a single function is a challenging task because usually execution is too fast to get reliable data. We control this threat repeating each function 1 million of times, that makes one sample. We collect 50 sample in order to reach statistical significance. Each function pair is loaded on the Waspmote and evaluated two times: the first one for the execution time, the latter one for the power consumption.

No specific instrumentation was needed to obtain the execution time, because the Waspmote embeds a Real-Time Clock (RTC) with a millisecond accuracy. We measure the execution time of every loop (i.e. 50 measurements).

On the other side, analyzing power consumption is more complicated. The only way to obtain a precise measure of the power consumption is using a power meter. The RTC is powered by an auxiliary battery, which makes it completely independent from the main power supply. Therefore it is possible to power the Waspmote with a constant voltage

(VG = 3.7 V) by means of a generator, and use a shunt resistor[24] to measure the current intensity. An analog to digital converter (ADC) connected to the PC reads the voltage drop across a resistor R of 1 Ω. The current flowing in the circuit can be computed by measuring the voltage drop on the resistor ($I = V_{ADC}/R$). The instant power consumption value can be computed as:

$$P = V_L \cdot I = (V_G - V_{ADC})\frac{V_{ADC}}{R} = \frac{V_G V_{ADC} - V_{ADC}^2}{R}$$

Figure 38 represents the circuit described.

The device used to measure the power consumption has a frequency of 49KHz, i.e. it gets 49000 measurements each second. In order to precisely measure the power consumption relative to the execution of the function pairs, we inserted a sleep interval at the beginning of the data acquisition to exclude the peak of device power on, and we filtered out, through a threshold, all the measurements corresponding to the idle consumption between the iterations of the function execution. As shown in Figure 39, the threshold filters out the transient and includes only the peaks corresponding to the actual execution of the function.

C. Analysis methodology

For each research question we derived a pair of null and alternative hypotheses to test.

[24] http://en.wikipedia.org/wiki/Shunt_(electrical)#Use_in_current_measuring, last visited September 13th, 2012

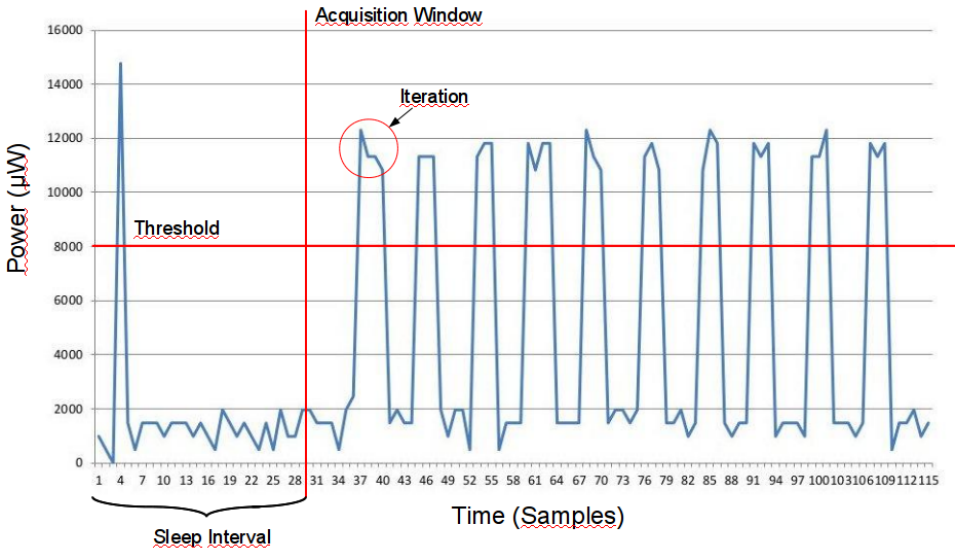

Figure 39. Sampling current intensity: an example

RQ1:

$H1_0 : P_{with} \leqslant P_{without}$

$H1_a : P_{with} > P_{without}$

where P is the power consumption of the function, with and without the potential smell. If the refactored version of the function consumes less than the function with the

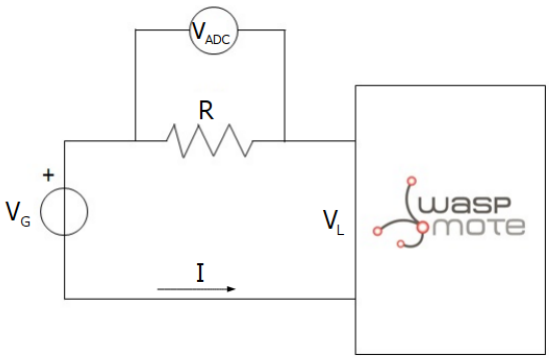

Figure 38. Circuit built to measure the power consumption

smell, the null hypothesis is rejected in favor of the alternative one. As a consequence we consider the pattern an Energy Code Smell. The hypothesis is tested with the Mann-Whitney test, given $\alpha=0.05$.

RQ2:

$H2_0 : T_{with} \leqslant T_{without}$

$H2_a : T_{with} > T_{without}$

where T is the execution time of the two functions. If the smell has a negative impact on performance, the refactored function will be faster and the null hypothesis is rejected. In that case, we consider the pattern a Performance Smell. In order to answer RQ2, we compare which Energy Code Smells are also Performance Smells. We also use Mann-Whitney and $\alpha=0.05$ to test the hypotheses.

At the end of the experiment each function has 50 measurements of execution time and about 25 millions of power measures. Then, after filtering out values below the idle threshold (8mW), we obtained about 8 million values for power measurement, on which we ran the analysis.

RESULTS

We report results on the power consumption and execution time respectively in Table 40 and Table 41. The two tables report the name of the smell, the means and their difference for both the dependent variables, the p-value of the Mann Whitney test and the difference in percentage of the power consumption (or execution time) between the execution of the code with the smell and the execution with the refactored code.

We observe from Table 40 that all power consumptions ranged from 40mW to about 42mW. Five code patterns over nine have a p-value < 0.05 (in bold) and therefore the null hypothesis is rejected for them. The code patterns are:

- DeadLocalStore
- NonShortCircuit

- ParameterByValue
- RepeatedConditionals
- SelfAssignment

Overall the saved power consumption is less than 1%. The answer to RQ1 is:

five code patterns (DeadLocalStore, NonShortCircuit, ParameterByValue, RepeatedConditionals, SelfAssignment) are Energy Code Smells, and their impact is in the order of μW.

Focusing on performance, from Table 41 becomes evident that there is no difference in execution time. The null hypothesis is rejected only for MutualExclusionOr, however the magnitudo order is μ seconds. We also notice that DeadLocalStores are about 5 times slower.

Thus, our answer to RQ2 is: *Energy Code Smells are not Performance Smells*.

Table 40. Results of power consumption

Smell name	Mean with smell (µ*W*)	Mean w/o smell (µ*W*)	Diff. Means (µ*W*)	P-value	Impact%
DeadLocalStoreReturn	41241	41278	-37	1	-0.09
DeadLocalStore	40249	40205	44	**< 0.01**	0.11
MutualExclusionOR	40758	40772	-14	1	-0.03
NonShortCircuit	41113	41043	70	**< 0.01**	0.17
ParameterByValue	40967	40723	244	**0**	0.60
RepeatedConditionals	41155	41126	29	**< 0.01**	0.07
SelfAssignment	40952	40879	73	**< 0.01**	0.18
SwitchRedundantAssignment	40724	40756	-32	1	-0.08
UselessControlFlow	41051	41142	-91	1	-0.22

Table 41. Results for execution time

Smell name	Mean with smell (*ms*)	Mean w/o smell (*ms*)	Diff. Means (*ms*)	P-value	Impact%
DeadLocalStoreReturn	3288.76	3288.74	0.02	0.41	6.08e-04
DeadLocalStore	17707.34	17707.38	-0.04	0.66	-2.26e-04
MutualExclusionOR	3540.76	3540.60	0.16	**0.04**	4.52e-03
NonShortCircuit	3288.74	3288.80	-0.06	0.76	-1.82e-03
ParameterByValue	3288.76	3288.74	0.02	0.41	6.08e-04
RepeatedConditionals	3288.80	3288.74	0.06	0.24	1.82e-03
SelfAssignment	3288.66	3288.78	-0.12	0.90	-3.64e-03
SwitchRedundantAssignment	3540.58	3540.62	-0.04	0.65	-1.13e-03
UselessControlFlow	3288.80	3288.74	0.06	0.24	1.82e-03

DISCUSSION

We identified five smells which provoked a higher power consumption of the Waspmote in the use cases prepared for the experimentation. However, we observe that the saved power is less than 1 %. A first motivation resides in the implementation choices: the function pairs executed only differ in a single instruction, and the operations are done with primitive types (e.g., integer). The motivation of such implementation was the exclusion of any possible confounding factor in the analysis, but the drawback of such a choice is a very small achievement in energy efficiency improvements.

Let us take dead stores as example: the smell *DeadLocalStore* is implemented with an integer (we save a value on a variable and immediately overwrites it with another integer). Using a struct with several members is totally different and might lead to a higher impact, because the resulting compiled code requires the CPU to produce more instructions and interact more intensively with the memory. If increasing the complexity of the data structure will result in still negligible power consumption saving, the next step is to increase the logical complexity of the function, i.e. comparing complete algorithms that are functionally equivalent but differ in the implementation. A further step is to move the focus towards the comparison of functionally equivalent design choices. Understanding the impact of Green Code Smells over real power consumption could also contribute to build more precise models of the power consumption of software. As a matter of fact, it may be possible to categorize software instructions beforehand in terms of energy efficiency, then subsequently use this information in order to predict the resulting energy efficiency of a complete software product.

Yet another research direction that is suggested by this first leap is: can the impact of Green Code Smells be higher in code that drives an hardware resource with higher energy needs? For instance the impact on the code that handles the GPS transmitter is expected to be very different from the one used in this experiment, where the small functions use only CPU and RAM, besides in a not intensive way. The same investigation approaches can be applied to the domain of execution time. As can be noticed from the results, all the execution times are equal, exception given for the *DeadLocalStore* function pairs. We have observed that Green Code Smells do not degrade the performances, but we cannot generalize the findings for more complex code structures and usage scenarios, with different hardware resources involved (e.g, sensors).

THREATS TO VALIDITY

In this section, we expose the threats to validity that might affect our study.

As regards construct validity, our main threat regards instrumentation. We carefully evaluated the precision of our measures, comparing them with the specifications from Waspmote manufacturers. During our experimentation, the difference between actual and expected values was negligible and inside the specified ranges. As far as conclusion threats are concerned, in order to increase the statistical reliability of the results, we collected a relevant amount of values (e.g., every function is looped 1 million times for power consumption measurement resulting in 25 millions of samples). Internal validity is represented by confounding factors such as other processes running during execution. However, the Waspmote does not have an operating system and the only thread in execution during the tests is the code loaded. As regards external validity, we do not aim at generalizing our results to a family of embedded devices. This study aims at assessing the existence of the Green Code Smells in a single context: other empirical validations are necessary for other environments or devices.

RELATED WORK

The analysis of software energetic impact on embedded systems was initially conducted by Tiwari in 1994 [148]. The author proposed a software power model based upon the CPU instruction set, determining those instructions which had a higher cost in energy terms.

Tiwari's model, although limited in some cases, was the enabler of the energy optimizations in compilers for embedded systems. Power-aware compilers try to optimize the resources a program needs for its execution, through different techniques, such as loop unrolling, software pipelining and recursion elimination [149]. In order to achieve higher optimizations, higher-level techniques are needed, such as algorithmic and data optimizations, that is a research direction closer to our work. The algorithmic optimization has a high potential, but it is also a hard and time-consuming task, with no guaranteed results. Data optimization is based upon the efficient use of the system architecture. For example, as regards embedded systems, often software libraries are used for emulating

floating-point hardware components. Those libraries do not take into account the architecture of a specific system, thus their usage often leads to a high power consumption and low performance. Šimunić et al. [150] show that by removing those libraries and optimizing the source code, it is possible to significantly reduce power consumption (up to 77%).

In terms of benchmarks, SPECpower [151] is an initiative to extend existing SPEC benchmarks to power and energy measurement. SPECpower ssj2008 reports the energy efficiency in terms of overall *ssj_ops*/*watt*. This metric represents the sum of the performance measured at each target load level (in *ssj_ops*) divided by the sum of the average power (in watts) at each target load including active idle.

In battery-powered systems, it is not enough to analyze algorithms based only on time and space complexity. Several research proposed energy aware algorithms for specific functionalities, such as supporting randomness [152] or focusing on cryptographic [153]. In particular, previous works by Bunse et al. [154] address the relationship between energy and performance optimizations, which is one of the research questions of the present work. Authors analyzed different implementations of several sorting algorithms, showing that implementations optimized for energy performed differently with respect to those optimized for performance. For example, one of their assumptions is that the use of recursion in a sorting algorithm might improve performance, but definitely increases power consumption. This finding shows how energy efficiency issues in software might lead to counter-intuitive conclusions. This is the reason why, in our work, we adopted an empirical approach in order to assess and validate our beliefs. In our opinion, more empirical studies focused on assessing the impact of software over power consumption might discover even more unexpected behaviors.

CONCLUSIONS

This is an exploratory study: we defined for the first time the concept of Energy Code Smells and we performed a first validation to understand not only the impact, but also

the boundaries of the concept. We identified some Energy Code Smells starting from code patterns implemented by two common Automatic Static Analysis tools - namely, CppCheck and FindBugs. We performed an experiment, on an embedded system, in order to assess the energetic impact of those code patterns, taking also into account the impact on execution time, to determine whether Energy Code Smells are also performance smells. Our experimental results showed that some of the code patterns actually have an impact over power consumption. This impact, however, is in the magnitude order of μW. Our future research works will be devoted to analyzing more complex data structures and using hardware resources which could increase this impact with respect to the overall power consumption. As regards time analysis, only one pattern had an actual impact over execution time (a few μ seconds), and it is not identified as an Energy Code Smell. Thus, we conclude that Energy Code Smells are not Performance Smells.

However, results suggest that the target and applicability of Energy Code Smells should be refined with further investigations. The lessons learned in this exploratory study let us identify several research threads that the research community might address, such as the identification of Energy Code Smells that are higher-level constructs and use more complex data structures, the identification of Energy Design Smells and the use of more complex systems as test beds.

The new challenge for ASA is to help *greening* the software identifying Energy Code Smells.

6 SUMMARY AND CONCLUSIONS

This PhD work empirically evaluated the impact of Automatic Static Analysis (ASA) on Software Quality, taking as reference model the ISO/IEC 25010 Software Product Quality Model. We conducted several experiments and case studies to understand whether applying ASA could improve certain software quality characteristics. We considered four quality characteristics: Functional Suitability and Reliability, Performance efficiency and Maintainability.

Regarding Functional Suitability and Reliability, we conducted two case studies analyzing the issues detected by the popular ASA tool FindBugs on two pools of similar small programs (85 and 301 programs respectively), each of them developed by a different student of the B.Sc. in Computer Engineering. By analyzing the changes and test failures in both studies, we observed that a small percentage of issues (about 3%) were related to known defects in the code.

We also conducted another study on an industrial web application with the Resharper tool analyzing the capability of the issues categories to identify the most defect prone files and components of an industrial web application. We found that resharper issues are better in identifying faulty modules than indicators of size and complexity. Using them we were able to find out problems associated to difficulty in the design of the code or a limited knowledge of the possibilities offered by the C# language.

Regarding Performance-Efficiency, we conducted two experiments with FindBugs on which we empirically proved that refactoring code on the basis of certain FindBugs issues will improve its execution time. In the first experiment, we selected 20 issues and for each of them we set up two source code fragments: one containing the issue and one containing the corresponding refactored version, functionally identical but without the issue. We set up three different platforms, isolated from network and other user programs, and then we executed the code fragments measuring the execution time of both code versions. We found that eleven issues have an actual negative impact on performance in all platforms (up to 6 times slower). In the following industrial experiment we quantitatively

assessed the impact on time efficiency of three code patterns: dead store to local variable, useless try-catch block and inefficient construction of Boolean objects. We refactored an industrial Java web application removing the three code patterns and we observed that the refactored software was about two fold faster (100 milliseconds less) than the original application.

Also regarding Maintainability we performed two case studies, focusing on the problem of Technical Debt. Technical Debt is a metaphor for the problem of high costs in the maintenance phase as a consequence of shortcuts taken during the development process (e.g., poor commented code, poor design, etc.). In the first case study we compared four TD techniques (code smells, automatic static analysis issues, grime buildup, and modularity violations) and applied them to 13 versions of the Apache Hadoop open source software project. Although ASA issues were not directly associated with Maintainability issues, those with higher priority were associated with classes with intensive coupling. A possible explanation for this relation is that both indicators point, more than any others, to generally poorly designed code. Moreover, the study demonstrated that the four approaches for TD identification have very little overlap and are therefore pointing to different problems in source code. These findings were confirmed by a second study in which we evaluated human elicitation of TD and compared it to automated identification. We asked a development team to identify technical debt items in artifacts of a software project. We provided the participants with a TD backlog and a short questionnaire. In addition, we also collected the output of three tools to automatically identify technical and compared it to the human elicitation. Our results show that aggregation, rather than consensus is an appropriate way to combine developer-reported debt, and that ASA tools used are especially useful for identifying defect debt but cannot help in identifying many other types of debt.

The last part of the thesis is devoted to new research challenges for ASA. We identified and proposed to the research community two innovative directions for future studies: ASA to green the software, and ASA for multi-language projects.

The motivation of the first suggestion resides in the fact that nowadays the spread of mobile and embedded devices makes energy efficiency a key requirement in many

software applications: we envision energy efficiency as a software quality characteristic, to be optimized through the refactoring of appropriate code patterns. Therefore we overtook the path towards the identification of those code patterns (defined *Energy Code Smells*) that might increase the impact of software over power consumption. For our purposes, we performed an experiment consisting in the execution of several code patterns on an embedded system. These instances of code patterns were executed in two versions: as in the in the experiment for Performance Efficiency, the first code pattern contains a code issue that could negatively impact power consumption, the other one is refactored removing the issue. We measured the power consumption of the embedded device during the execution of each code pattern: our results show that some Energy Code Smells actually have an impact over power consumption (in our test bed the magnitude order was µWatts). Moreover, removing those Energy Smells did not introduce a performance decrease.

The second research challenge is related to another observation: nowadays most software systems are complex and composed of a large number of artifacts. To realize each different artifact specific techniques are used resorting to different abstractions, languages and tools: successful composition of different elements requires coherence among them. Unfortunately constraints between artifacts written in different languages are usually not formally expressed nor checked by supporting tools; as a consequence they can be a source of problems. We explored the role of the relations between artifacts written in different languages by means of a case study on the Hadoop open source project, quantifying the phenomenon and investigating its relation with defect proneness. We observed that more than half of the commits in Hadoop are cross language (according to our definition). However we also observed that this property depends on the type of the activities and the extensions of the modules. Also looking at defect proneness, we observed that the interaction between specific pairs of extensions was related to higher defect proneness. Although we did not get unique answers, the results and observations from this exploratory study let us understand that the problem is worthy to be investigated, and that the agenda of the next years for ASA should include the capability to identify problems deriving from the interaction of multiple language, rather than focusing only on problems inside the boundaries of one specific programming language.

The findings and results summarized above are specific to the contexts of the experiments and case studies conducted. We attempted to generalize results on Functional Suitability and Reliability by a comparison with similar studies. However this was not possible for Performance Efficiency and Maintainability given the absence of similar approaches, and of course also for the new research directions identified. A broader generalization will be possible as soon as further evidence will be collected by means of other empirical studies on the different quality attributes. Although, this thesis provides, besides the first empirical results on the impact of ASA on different software quality characteristics, a *modus operandi* that can be applied to different contexts to enlarge the body of knowledge in the topic and form a *grounded theory* on ASA and software quality. This *modus operandi* can be packaged in a process composed of 5 steps:

- *Elicitation*: elicit quality characteristics and/or sub-characteristic from the various stakeholders, such as software developers, system integrators, acquirers, owners, maintainers, etc. We suggest to start from the characteristics defined by the ISO/IEC 25010, but to not limit to them. As we have seen in Chapter 6, characteristics like energy efficiency are not included in the standard but have relevance in contexts like mobile application development or embedded devices.
- *Prioritization*: prioritize the quality characteristics taking into consideration both the technical and the business view. It might be possible to use support decision systems or risk assessment tools.
- *Measurement*: define the quality measure elements, the measurement functions and the quality measures for each quality characteristic (or sub-characteristic) identified.
- *Assessment:* identify a methodological tool to empirical assess which ASA issues have an impact on the quality measures one or more software project. Results might be compared with previous work in the literature or past experience. Package results in a quality management base in the more appropriate form (e.g.: experience/technical report, database of statistics, wiki, etc).

- *Monitoring, refactoring, reporting:* enable the detection of the relevant ASA issues (according to the prioritization and the assessment) in the development process and when necessary refactor the code by removing the issues enabled. Measure the quality improvement and report in the quality management base.

The goal of this process is twofold: not only improve and monitor the quality of your code through ASA, but also learn from past errors.

7 BIBLIOGRAPHY

[1] P. Tonella, M. Torchiano, B. Du Bois, and T. Systä, "Empirical studies in reverse engineering: state of the art and future trends," *Empirical Softw. Engg.*, vol. 12, no. 5, pp. 551–571, Oct. 2007. [Online]. Available: http://dx.doi.org/10.1007/s10664-007-9037-5

[2] D. Binkley, "Source code analysis: A road map," in *2007 Future of Software Engineering*, ser. FOSE '07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 104–119. [Online]. Available: http://dx.doi.org/10.1109/FOSE.2007.27

[3] S. Johnson, "Lint: A c program checker," Bell Laboratories, Tech. Rep. 65, 1977.

[4] P. Li and B. Cui, "A comparative study on software vulnerability static analysis techniques and tools," in *Information Theory and Information Security (ICITIS), 2010 IEEE International Conference on*, dec. 2010, pp. 521 –524.

[5] P. Louridas, "Static code analysis," *Software, IEEE*, vol. 23, no. 4, pp. 58 –61, july-aug. 2006.

[6] B. W. Boehm, "Software process management: lessons learned from history," in *Proceedings of the 9th international conference on Software Engineering*, ser. ICSE '87. Los Alamitos, CA, USA: IEEE Computer Society Press, 1987, pp. 296–298. [Online]. Available: http://dl.acm.org/citation.cfm?id=41765.41798

[7] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén, *Experimentation in software engineering: an introduction*. Norwell, MA, USA: Kluwer Academic Publishers, 2000.

[8] W. F. Tichy, "Should computer scientists experiment more?" *Computer*, vol. 31, no. 5, pp. 32–40, May 1998. [Online]. Available: http://dx.doi.org/10.1109/-2.675631

[9] V. R. Basili, "The role of experimentation in software engineering: past, current, and future," in *Proceedings of the 18th international conference on Software*

engineering, ser. ICSE '96. Washington, DC, USA: IEEE Computer Society, 1996, pp. 442–449. [Online]. Available: http://dl.acm.org/citation.cfm?id=227726.227818

[10] J. D. Herbsleb, "Msr: Mining for scientific results?" in *MSR*, 2010.

[11] D. I. K. Sjoberg, T. Dyba, and M. Jorgensen, "The future of empirical methods in software engineering research," in *2007 Future of Software Engineering*, ser. FOSE '07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 358–378. [Online]. Available: http://dx.doi.org/10.1109/FOSE.2007.30

[12] B. Meyer, "Empirical research: questions from software engineering," http://esem2010.case.unibz.it/docs/Meyer-ESEM2010-KN.pdf, 2010, keynote ESEM 2010.

[13] J. E. Hannay, D. I. K. Sjoberg, and T. Dyba, "A systematic review of theory use in software engineering experiments," *IEEE Trans. Softw. Eng.*, vol. 33, no. 2, pp. 87–107, Feb. 2007. [Online]. Available: http://dx.doi.org/10.1109/TSE.2007.12

[14] F. Shull, J. Singer, and D. I. Sjøberg, *Guide to Advanced Empirical Software Engineering*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2007.

[15] N. Juristo and A. M. Moreno, *Basics of Software Engineering Experimentation*, 1st ed. Springer Publishing Company, Incorporated, 2010.

[16] B. Boehm and V. R. Basili, "Software defect reduction top 10 list," *Computer*, vol. 34, no. 1, pp. 135–137, Jan. 2001. [Online]. Available: http://dx.doi.org/-10.1109/2.962984

[17] S. Wagner, J. Jürjens, C. Koller, and P. Trischberger, "Comparing bug finding tools with reviews and tests," in *Proceedings of the 17th IFIP TC6/WG 6.1 international conference on Testing of Communicating Systems*, ser. TestCom'05. Berlin, Heidelberg: Springer-Verlag, 2005, pp. 40–55. [Online]. Available: http://dx.doi.org/-10.1007/11430230_4

[18] F. Wedyan, D. Alrmuny, and J. M. Bieman, "The effectiveness of automated static analysis tools for fault detection and refactoring prediction," in *Proceedings of the 2009 International Conference on Software Testing Verification and Validation*, ser. ICST '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 141–150. [Online]. Available: http://dx.doi.org/10.1109/ICST.2009.21

[19] N. Ayewah, W. Pugh, J. D. Morgenthaler, J. Penix, and Y. Zhou, "Evaluating static analysis defect warnings on production software," in *Proceedings of the 7th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, ser. PASTE '07. New York, NY, USA: ACM, 2007, pp. 1–8. [Online]. Available: http://doi.acm.org/10.1145/1251535.1251536

[20] C. Boogerd and L. Moonen, "Assessing the value of coding standards: An empirical study," in *Software Maintenance, 2008. ICSM 2008. IEEE International Conference on*, 28 2008-oct. 4 2008, pp. 277 –286.

[21] ——, "Evaluating the relation between coding standard violations and faultswithin and across software versions," in *Mining Software Repositories, 2009. MSR '09. 6th IEEE International Working Conference on*, may 2009, pp. 41 –50.

[22] MIRA Ltd, *MISRA-C:2004 Guidelines for the use of the C language in Critical Systems*, MIRA Std., Oct. 2004. [Online]. Available: www.misra.org.uk

[23] N. Nagappan and T. Ball, "Static analysis tools as early indicators of pre-release defect density," in *Software Engineering, 2005. ICSE 2005. Proceedings. 27th International Conference on*, may 2005, pp. 580 – 586.

[24] N. Nagappan, L. Williams, J. Hudepohl, W. Snipes, and M. Vouk, "Preliminary results on using static analysis tools for software inspection," in *Software Reliability Engineering, 2004. ISSRE 2004. 15th International Symposium on*, nov. 2004, pp. 429 – 439.

[25] J. Zheng, L. Williams, N. Nagappan, W. Snipes, J. Hudepohl, and M. Vouk, "On the value of static analysis for fault detection in software," *Software Engineering, IEEE Transactions on*, vol. 32, no. 4, pp. 240 – 253, april 2006.

[26] R. Plosch, H. Gruber, A. Hentschel, G. Pomberger, and S. Schiffer, "On the relation between external software quality and static code analysis," in *Software Engineering Workshop, 2008. SEW '08. 32nd Annual IEEE*, oct. 2008, pp. 169 –174.

[27] A. Marchenko and P. Abrahamsson, "Predicting software defect density: A case study on automated static code analysis." in *XP*, ser. Lecture Notes in Computer Science, G. Concas, E. Damiani, M. Scotto, and G. Succi, Eds., vol. 4536. Springer, 2007,

pp. 137–140. [Online]. Available: http://dblp.uni-trier.de/db/conf/xpu/-xp2007.html#MarchenkoA07

[28] D. Hovemeyer and W. Pugh, "Finding bugs is easy," *SIGPLAN Not.*, vol. 39, no. 12, pp. 92–106, Dec. 2004. [Online]. Available: http://doi.acm.org/10.1145/-1052883.1052895

[29] M. Torchiano and M. Morisio, "A fully automatic approach to the assessment of programming assignments," *International Journal of Engineering Education*, vol. 25, no. 4, pp. 814–829, 2009, cited By (since 1996) 1. [Online]. Available: http://-www.scopus.com/inward/record.url?eid=2-s2.0-69949175639&partnerID=40&md5=d82f7fd2f1e4f55f2d5cb5012822c498

[30] S. Kim and M. D. Ernst, "Prioritizing warning categories by analyzing software history," in *Proceedings of the Fourth International Workshop on Mining Software Repositories*, ser. MSR '07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 27–. [Online]. Available: http://dx.doi.org/10.1109/MSR.2007.26

[31] A. Agresti, *An Introduction to Categorical Data Analysis (Wiley Series in Probability and Statistics)*, 2nd ed. Wiley-Interscience, Mar. 2007. [Online]. Available: http://www.worldcat.org/isbn/0471226181

[32] E. N. Adams, "Optimizing preventive service of software products," *IBM J. Res. Dev.*, vol. 28, no. 1, pp. 2–14, Jan. 1984. [Online]. Available: http://dx.doi.org/-10.1147/rd.281.0002

[33] A. Vetro', M. Torchiano, and M. Morisio, "Assessing the precision of findbugs by mining java projects developed at a university," in *Mining Software Repositories (MSR), 2010 7th IEEE Working Conference on*, may 2010, pp. 110 –113.

[34] V. Basili, G. Caldiera, and D. H. Rombach, "The goal question metric approach," in *Encyclopedia of Software Engineering*, J. Marciniak, Ed. Wiley, 1994. [Online]. Available: https://docweb.lrz-muenchen.de/cgi-bin/doc/nph-webdoc.cgi/-000110A/http/scholar.google.de/-scholar=3fhl=3dde&lr=3d&cluster=3d4068380033007143449

[35] D. J. Sheskin, *Handbook of Parametric and Nonparametric Statistical Procedures*, 4th ed. Chapman & Hall/CRC, 2007.

[36] P. Runeson, “Using students as experiment subjects–an analysis on graduate and freshmen student data,” in *Proceedings of the 7th International Conference on Empirical Assessment in Software Engineering*. Citeseer, 2003, pp. 95–102.

[37] S. Kim and M. D. Ernst, “Which warnings should i fix first?” in *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, ser. ESEC-FSE ’07. New York, NY, USA: ACM, 2007, pp. 45–54. [Online]. Available: http://-doi.acm.org/10.1145/1287624.1287633

[38] A. Vetro’, M. Morisio, and M. Torchiano, “An empirical validation of findbugs issues related to defects,” in *Evaluation Assessment in Software Engineering (EASE 2011), 15th Annual Conference on*, april 2011, pp. 144 –153.

[39] N. Ayewah and W. Pugh, “The google findbugs fixit,” in *Proceedings of the Nineteenth International Symposium on Software Testing and Analysis, ISSTA 2010, Trento, Italy, July 12-16, 2010*, P. Tonella and A. Orso, Eds. ACM, 2010, pp. 241–252.

[40] R. Chillarege, I. Bhandari, J. Chaar, M. Halliday, D. Moebus, B. Ray, and M.-Y. Wong, “Orthogonal defect classification-a concept for in-process measurements,” *Software Engineering, IEEE Transactions on*, vol. 18, no. 11, pp. 943 –956, nov 1992.

[41] ISO/IEC, *ISO/IEC 9126. Software engineering – Product quality*, ISO/IEC Std., 2001.

[42] ——, *ISO/IEC 25010. Systems and software engineering – Systems and software Quality Requirements and Evaluation (SQuaRE) – System and software quality models*, ISO/IEC Std., 2011.

[43] A. Vetro’, N. Zazworka, C. Seaman, and F. Shull, “Using the iso/iec 9126 product quality model to classify defects: A controlled experiment,” in *Evaluation Assessment in Software Engineering (EASE 2012), 16th International Conference on*, may 2012, pp. 187 –196.

[44] N. Zazworka, K. Stapel, E. Knauss, F. Shull, V. R. Basili, and K. Schneider, “Are developers complying with the process: an xp study,” in *Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and*

Measurement, ser. ESEM '10. New York, NY, USA: ACM, 2010, pp. 14:1–14:10. [Online]. Available: http://doi.acm.org/10.1145/1852786.1852805

[45] A. G. Koru, D. Zhang, and H. Liu, "Modeling the effect of size on defect proneness for open-source software," in *Proceedings of the Third International Workshop on Predictor Models in Software Engineering*, ser. PROMISE '07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 10–. [Online]. Available: http://dx.doi.org/10.1109/-PROMISE.2007.9

[46] N. Nagappan, T. Ball, and A. Zeller, "Mining metrics to predict component failures," in *Proceedings of the 28th international conference on Software engineering*, ser. ICSE '06. New York, NY, USA: ACM, 2006, pp. 452–461. [Online]. Available: http://doi.acm.org/10.1145/1134285.1134349

[47] A. G. Koru, K. E. Emam, D. Zhang, H. Liu, and D. Mathew, "Theory of relative defect proneness," *Empirical Softw. Engg.*, vol. 13, pp. 473–498, October 2008. [Online]. Available: http://dl.acm.org/citation.cfm?id=1416830.1416834

[48] N. E. Fenton and N. Ohlsson, "Quantitative analysis of faults and failures in a complex software system," *IEEE Trans. Softw. Eng.*, vol. 26, pp. 797–814, August 2000. [Online]. Available: http://dl.acm.org/citation.cfm?id=630824.631250

[49] A. G. Koru and H. Liu, "An investigation of the effect of module size on defect prediction using static measures," *SIGSOFT Softw. Eng. Notes*, vol. 30, pp. 1–5, May 2005. [Online]. Available: http://doi.acm.org/10.1145/1082983.1083172

[50] N. Brown, Y. Cai, Y. Guo, R. Kazman, M. Kim, P. Kruchten, E. Lim, A. MacCormack, R. Nord, I. Ozkaya, R. Sangwan, C. Seaman, K. Sullivan, and N. Zazworka, "Managing technical debt in software-reliant systems," in *Proceedings of the FSE/SDP workshop on Future of software engineering research*, ser. FoSER '10. New York, NY, USA: ACM, 2010, pp. 47–52. [Online]. Available: http://doi.acm.org/10.1145/-1882362.1882373

[51] W. Cunningham, "The wycash portfolio management system," *SIGPLAN OOPS Mess.*, vol. 4, no. 2, pp. 29–30, Dec. 1992. [Online]. Available: http://doi.acm.org/-10.1145/157710.157715

[52] C. Izurieta, A. Vetro, N. Zazworka, Y. Cai, C. Seaman, and F. Shull, "Organizing the technical debt landscape," in *Managing Technical Debt (MTD), 2012 Third International Workshop on*, june 2012, pp. 23 –26.

[53] C. Seaman, Y. Guo, C. Izurieta, Y. Cai, N. Zazworka, F. Shull, and A. Vetro, "Using technical debt data in decision making: Potential decision approaches," in *Managing Technical Debt (MTD), 2012 Third International Workshop on*, june 2012, pp. 45 –48.

[54] J. Schumacher, N. Zazworka, F. Shull, C. Seaman, and M. Shaw, "Building empirical support for automated code smell detection," in *Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, ser. ESEM '10. New York, NY, USA: ACM, 2010, pp. 8:1–8:10. [Online]. Available: http://doi.acm.org/10.1145/1852786.1852797

[55] S. Wong, Y. Cai, M. Kim, and M. Dalton, "Detecting software modularity violations," in *Proceedings of the 33rd International Conference on Software Engineering*, ser. ICSE '11. New York, NY, USA: ACM, 2011, pp. 411–420. [Online]. Available: http://doi.acm.org/10.1145/1985793.1985850

[56] C. Izurieta and J. M. Bieman, "How software designs decay: A pilot study of pattern evolution," in *Proceedings of the First International Symposium on Empirical Software Engineering and Measurement*, ser. ESEM '07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 449–451. [Online]. Available: http://dx.doi.org/10.1109/-ESEM.2007.58

[57] S. Muthanna, K. Ponnambalam, K. Kontogiannis, and B. Stacey, "A maintainability model for industrial software systems using design level metrics," in *Proceedings of the Seventh Working Conference on Reverse Engineering (WCRE'00)*, ser. WCRE '00. Washington, DC, USA: IEEE Computer Society, 2000, pp. 248–. [Online]. Available: http://dl.acm.org/citation.cfm?id=832307.837117

[58] N. Fenton, M. Neil, W. Marsh, P. Hearty, D. Marquez, P. Krause, and R. Mishra, "Predicting software defects in varying development lifecycles using bayesian nets," *Inf. Softw. Technol.*, vol. 49, no. 1, pp. 32–43, Jan. 2007. [Online]. Available: http://-dx.doi.org/10.1016/j.infsof.2006.09.001

[59] M. Riaz, E. Mendes, and E. Tempero, “A systematic review of software maintainability prediction and metrics,” in *Proceedings of the 2009 3rd International Symposium on Empirical Software Engineering and Measurement*, ser. ESEM ’09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 367–377. [Online]. Available: http://dx.doi.org/10.1109/ESEM.2009.5314233

[60] I. Gat and J. D. Heintz, “From assessment to reduction: how cutter consortium helps rein in millions of dollars in technical debt,” in *Proceedings of the 2nd Workshop on Managing Technical Debt*, ser. MTD ’11. New York, NY, USA: ACM, 2011, pp. 24–26. [Online]. Available: http://doi.acm.org/10.1145/1985362.1985368

[61] A. Nugroho, J. Visser, and T. Kuipers, “An empirical model of technical debt and interest,” in *Proceedings of the 2nd Workshop on Managing Technical Debt*, ser. MTD ’11. New York, NY, USA: ACM, 2011, pp. 1–8. [Online]. Available: http://-doi.acm.org/10.1145/1985362.1985364

[62] M. Fowler, *Refactoring: Improving the Design of Existing Code*. Boston, MA, USA: Addison-Wesley, 1999.

[63] C. Izurieta and J. Bieman, “EnglishA multiple case study of design pattern decay, grime, and rot in evolving software systems,” *EnglishSoftware Quality Journal*, vol. I, pp. 1–35, 2012.

[64] Y.-G. Guéhéneuc and H. Albin-Amiot, “Using design patterns and constraints to automate the detection and correction of inter-class design defects,” in *Proceedings of the 39th International Conference and Exhibition on Technology of Object-Oriented Languages and Systems (TOOLS39)*, ser. TOOLS ’01. Washington, DC, USA: IEEE Computer Society, 2001, pp. 296–. [Online]. Available: http://dl.acm.org/-citation.cfm?id=882501.884740

[65] C. Izurieta and J. M. Bieman, “Testing consequences of grime buildup in object oriented design patterns,” in *Proceedings of the 2008 International Conference on Software Testing, Verification, and Validation*, ser. ICST ’08. Washington, DC, USA: IEEE Computer Society, 2008, pp. 171–179. [Online]. Available: http://dx.doi.org/10.1109/-ICST.2008.27

[66] W. J. Brown, R. C. Malveau, H. W. McCormick, III, and T. J. Mowbray, *AntiPatterns: refactoring software, architectures, and projects in crisis*. New York, NY, USA: John Wiley & Sons, Inc., 1998.

[67] M. Lanza, R. Marinescu, and S. Ducasse, *Object-Oriented Metrics in Practice*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2005.

[68] R. Marinescu, "Detection strategies: Metrics-based rules for detecting design flaws," in *Proceedings of the 20th IEEE International Conference on Software Maintenance*, ser. ICSM '04. Washington, DC, USA: IEEE Computer Society, 2004, pp. 350–359. [Online]. Available: http://dl.acm.org/citation.cfm?id=1018431.1021443

[69] N. Zazworka, M. A. Shaw, F. Shull, and C. Seaman, "Investigating the impact of design debt on software quality," in *Proceedings of the 2nd Workshop on Managing Technical Debt*, ser. MTD '11. New York, NY, USA: ACM, 2011, pp. 17–23. [Online]. Available: http://doi.acm.org/10.1145/1985362.1985366

[70] S. M. Olbrich, D. S. Cruzes, and D. I. K. Sjoberg, "Are all code smells harmful? a study of god classes and brain classes in the evolution of three open source systems," in *Proceedings of the 2010 IEEE International Conference on Software Maintenance*, ser. ICSM '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 1–10. [Online]. Available: http://dx.doi.org/10.1109/ICSM.2010.5609564

[71] J. L. Fleiss, *Statistical Methods for Rates and Proportions*, 2nd ed., ser. Wiley series in probability and mathematical statistics. New York: John Wiley & Sons, 1981.

[72] K. El Emam and I. Wieczorek, "The repeatability of code defect classifications," in *Software Reliability Engineering, 1998. Proceedings. The Ninth International Symposium on*, nov 1998, pp. 322 –333.

[73] H.-M. Park and H.-W. Jung, "Evaluating interrater agreement with intraclass correlation coefficient in spice-based software process assessment," in *Quality Software, 2003. Proceedings. Third International Conference on*, nov. 2003, pp. 308 – 314.

[74] J. Cohen, *Statistical power analysis for the behavioral sciences : Jacob Cohen.*, 2nd ed. Lawrence Erlbaum, Jan. 1988. [Online]. Available: http://-www.worldcat.org/isbn/0805802835

[75] J. Evans, *Straightforward statistics for the behavioral sciences*. Brooks/Cole Pub. Co., 1996. [Online]. Available: http://books.google.com/-books?id=8Ca2AAAAIAAJ

[76] J. R. Landis and G. G. Koch, "The Measurement of Observer Agreement for Categorical Data," *Biometrics*, vol. 33, no. 1, pp. 159–174, Mar. 1977.

[77] D. G. Altman, *Practical Statistics for Medical Research (Statistics texts)*, 1st ed. Chapman & Hall/CRC, Nov. 1990. [Online]. Available: http://www.worldcat.org/-isbn/0412276305

[78] M. D'Ambros, A. Bacchelli, and M. Lanza, "On the impact of design flaws on software defects," in *Quality Software (QSIC), 2010 10th International Conference on*, july 2010, pp. 23 –31.

[79] C. B. Seaman and Y. Guo, "Measuring and monitoring technical debt," *Advances in Computers*, vol. 82, pp. 25–46, 2011.

[80] Y. Guo, C. Seaman, N. Zazworka, and F. Shull, "Domain-specific tailoring of code smells: an empirical study," in *Software Engineering, 2010 ACM/IEEE 32nd International Conference on*, vol. 2, may 2010, pp. 167 –170.

[81] N. Zazworka and C. Ackermann, "Codevizard: a tool to aid the analysis of software evolution," in *Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, ser. ESEM '10. New York, NY, USA: ACM, 2010, pp. 63:1–63:1. [Online]. Available: http://doi.acm.org/10.1145/-1852786.1852865

[82] N. Zazworka, C. Seaman, and F. Shull, "Prioritizing design debt investment opportunities," in *Proceedings of the 2nd Workshop on Managing Technical Debt*, ser. MTD '11. New York, NY, USA: ACM, 2011, pp. 39–42. [Online]. Available: http://doi.acm.org/10.1145/1985362.1985372

[83] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström, "The worst-case execution-time problem – overview of methods and survey of tools," *ACM Trans. Embed. Comput. Syst.*, vol. 7, pp. 36:1–36:53, May 2008.

[84] T. Harmon and R. Klefstad, “A survey of worst-case execution time analysis for real-time java,” *Parallel and Distributed Processing Symposium, International*, vol. 0, p. 232, 2007.

[85] G. Bernat, A. Burns, and A. Wellings, “Portable worst-case execution time analysis using java byte code,” in *In Proc. 12th Euromicro International Conference on Real-Time Systems*, 2000, pp. 81–88.

[86] E. Y.-S. Hu, G. Bernat, and A. Wellings, “Addressing dynamic dispatching issues in wcet analysis for object-oriented hard real-time systems,” *Object-Oriented Real-Time Distributed Computing, IEEE International Symposium on*, vol. 0, p. 0109, 2002.

[87] I. Bate, G. Bernat, G. Murphy, and P. Puschner, “Low-level analysis of a portable java byte code wcet analysis framework,” *Real-Time Computing Systems and Applications, International Workshop on*, vol. 0, p. 39, 2000.

[88] I. Bate, G. Bernat, and P. Puschner, “Java virtual-machine support for portable worst-case execution-time analysis,” in *In Proc. 5th IEEE Intl. Symp. on Object-Oriented Real-Time Distributed Computing*, 2002, pp. 83–90.

[89] D. S. Hardin, “Real-Time Objects on the Bare Metal: An Efficient Hardware Realization of the JavaTM Virtual Machine,” *Object-Oriented Real-Time Distributed Computing, IEEE International Symposium on*, vol. 0, pp. 0053+, 2001.

[90] M. Arnold, S. J. Fink, D. Grove, M. Hind, and P. F. Sweeney, “A survey of adaptive optimization in virtual machines,” *Proceedings of the IEEE*, vol. 93, no. 2, pp. 449–466, Feb. 2005.

[91] M. H. . D. A. Wolfe, *Nonparametric Statistical Methods*. New York: John Wiley & Sons, 1973.

[92] I. H. Kazi, H. H. Chen, B. Stanley, and D. J. Lilja, “Techniques for obtaining high performance in java programs,” *ACM Comput. Surv.*, vol. 32, pp. 213–240, September 2000. [Online]. Available: http://doi.acm.org/10.1145/367701.367714

[93] B. R. Rau, “Levels of representation of programs and the architecture of universal host machines,” *SIGMICRO Newsl.*, vol. 9, pp. 67–79, November 1978. [Online]. Available: http://portal.acm.org/citation.cfm?id=1014198.804311

[94] J. Lau, M. Arnold, M. Hind, and B. Calder, "Online performance auditing: using hot optimizations without getting burned," *SIGPLAN Not.*, vol. 41, pp. 239–251, June 2006. [Online]. Available: http://doi.acm.org/10.1145/1133255.1134010

[95] M. Fowler, *Patterns of Enterprise Application Architecture*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2002.

[96] G. E. P. Box, J. S. Hunter, and W. G. Hunter, *Statistics for Experimenters: Design, Innovation, and Discovery*, 2nd ed. Wiley-Interscience, 2005.

[97] L. Sachs, *Applied Statistics–A Handbook of Techniques*, S. S. in Statistics, Ed. Springer-Verlag, 1984.

[98] Y. Fujita, M. Murata, and H. Miyahara, "Analysis of web server performancetoward modeling and performance evaluation of web systems," in *Proceedings of IEEE SICON*, 1998.

[99] Z. Liu, N. Niclausse, and C. Jalpa-Villanueva, "Traffic model and performance evaluation of web servers," *Perform. Eval.*, vol. 46, no. 2-3, pp. 77–100, Oct. 2001. [Online]. Available: http://dx.doi.org/10.1016/S0166-5316(01)00046-3

[100] I. M. Chakravarti, J. Roy, and R. G. Laha, *EnglishHandbook of methods of applied statistics*. Wiley New York, 1967.

[101] M. A. Stephens, "Edf statistics for goodness of fit and some comparisons," *Journal of the American Statistical Association*, vol. 69, no. 347, pp. 730–737, 1974. [Online]. Available: http://dx.doi.org/10.2307/2286009

[102] A. I. Shawky and R. A. Bakoban, "Modified goodness-of-fit tests for the exponentiated gamma distribution with unknown shape parameter," 2009. [Online]. Available: http://interstat.statjournals.net/

[103] M. H. . D. A. Wolfe, *Nonparametric Statistical Methods*. New York: John Wiley & Sons, 1973.

[104] K. Strassburger and F. Bretz, "Compatible simultaneous lower confidence bounds for the holm procedure and other bonferroni-based closed tests," *Statistics in Medicine*, vol. 27, pp. 4914–4927, 2008.

[105] R. B. Miller, "Response time in man-computer conversational transactions," in *Proceedings of the December 9-11, 1968, fall joint computer conference,*

part I, ser. AFIPS '68 (Fall, part I). New York, NY, USA: ACM, 1968, pp. 267–277. [Online]. Available: http://doi.acm.org/10.1145/1476589.1476628

[106] S. K. Card, G. G. Robertson, and J. D. Mackinlay, "The information visualizer, an information workspace," in *Proceedings of the SIGCHI conference on Human factors in computing systems: Reaching through technology*, ser. CHI '91. New York, NY, USA: ACM, 1991, pp. 181–186. [Online]. Available: http://doi.acm.org/-10.1145/108844.108874

[107] J. Nielsen, *Usability Engineering*. San Francisco, California: Morgan Kaufmann Publishers, October 1994.

[108] F. F. Nah, "A study on tolerable waiting time: how long are web users willing to wait?" *Behaviour & Information Technology*, vol. 23, no. 3, pp. 153–163, 2004. [Online]. Available: http://dx.doi.org/10.1080/01449290410001669914

[109] T. A. Proebsting, G. Townsend, P. Bridges, J. H. Hartman, T. Newsham, and S. A. Watterson, "Toba: Java for applications - a way ahead of time (wat) compiler," in *In Proceedings of the 3rd Conference on Object-Oriented Technologies and Systems*, 1997, pp. 41–53.

[110] G. Muller, F. Bellard, and C. Consel, "Harissa: a flexible and efficient java environment mixing bytecode and compiled code," in *In Proceedings of the 3rd Conference on Object-Oriented Technologies and Systems*. Usenix, 1996, pp. 1–20.

[111] C.-H. A. Hsieh, J. C. Gyllenhaal, and W.-m. W. Hwu, "Java bytecode to native code translation: the caffeine prototype and preliminary results," in *Proceedings of the 29th annual ACM/IEEE international symposium on Microarchitecture*, ser. MICRO 29. Washington, DC, USA: IEEE Computer Society, 1996, pp. 90–99. [Online]. Available: http://portal.acm.org/citation.cfm?id=243846.243864

[112] M. Weiss, F. Ferrière, B. Delsart, C. Fabre, F. Hirsch, E. Johnson, V. Joloboff, F. Roy, F. Siebert, and X. Spengler, "Turboj, a java bytecode-to-native compiler," in *Languages, Compilers, and Tools for Embedded Systems*, ser. Lecture Notes in Computer Science, F. Mueller and A. Bestavros, Eds. Springer Berlin Heidelberg, 1998, vol. 1474, pp. 119–130. [Online]. Available: http://dx.doi.org/10.1007/BFb0057785

[113] J. Aycock, “A brief history of just-in-time,” *ACM Comput. Surv.*, vol. 35, no. 2, pp. 97–113, 2003.

[114] G. J. Hansen, “Adaptive systems for the dynamic run-time optimization of programs.” Ph.D. dissertation, Carnegie Mellon University Pittsburgh, PA, Pittsburgh, PA, USA, 1974, aAI7420364.

[115] U. Hölzle and D. Ungar, “Reconciling responsiveness with performance in pure object-oriented languages,” *ACM Trans. Program. Lang. Syst.*, vol. 18, pp. 355–400, July 1996. [Online]. Available: http://doi.acm.org/10.1145/233561.233562

[116] C. Chambers and D. Ungar, “Making pure object-oriented languages practical,” *SIGPLAN Not.*, vol. 26, pp. 1–15, November 1991. [Online]. Available: http://-doi.acm.org/10.1145/118014.117955

[117] M. Paleczny, C. Vick, and C. Click, “The java hotspot(tm) server compiler,” in *In USENIX Java Virtual Machine Research and Technology Symposium*, 2001, pp. 1–12.

[118] M. D. Smith, “Overcoming the challenges to feedback-directed optimization (keynote talk),” *SIGPLAN Not.*, vol. 35, pp. 1–11, January 2000. [Online]. Available: http://doi.acm.org/10.1145/351403.351408

[119] M. K. Chen and K. Olukotun, “The jrpm system for dynamically parallelizing java programs,” *SIGARCH Comput. Archit. News*, vol. 31, pp. 434–446, May 2003. [Online]. Available: http://doi.acm.org/10.1145/871656.859668

[120] J. E. Moreira, S. P. Midkiff, M. Gupta, P. Wu, G. Almasi, and P. Artigas, “Ninja: Java for high performance numerical computing,” *Sci. Program.*, vol. 10, pp. 19–33, January 2002. [Online]. Available: http://portal.acm.org/-citation.cfm?id=1239945.1239951

[121] S. Uhrig, “The many java core processor (manjac),” in *HPCS*, 2010, p. 188.

[122] M. Schoeberl, “A time predictable java processor,” in *In Proceedings of the Design, Automation and Test in Europe Conference (DATE 2006*, 2006, pp. 800–805.

[123] ——, “A java processor architecture for embedded real-time systems,” *J. Syst. Archit.*, vol. 54, no. 1-2, pp. 265–286, Jan. 2008. [Online]. Available: http://-dx.doi.org/10.1016/j.sysarc.2007.06.001

[124] Y. Tan, L. W. Yiu, C. Yau, R. Li, and A. S. Fong, “A java processor with hardware-support object-oriented instructions,” *Microprocessors and Microsystems*, vol. 30, no. 8, pp. 469–479, 2006.

[125] K. Hoste, A. Georges, and L. Eeckhout, “Automated just-in-time compiler tuning,” in *Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization*, ser. CGO ’10. New York, NY, USA: ACM, 2010, pp. 62–72. [Online]. Available: http://doi.acm.org/10.1145/1772954.1772965

[126] X. Liu, “Exploiting object-based parallelism on multi-core multi-processor clusters,” in *Proceedings of the Eighth International Conference on Parallel and Distributed Computing, Applications and Technologies*, ser. PDCAT ’07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 259–266. [Online]. Available: http://dx.doi.org/-10.1109/PDCAT.2007.40

[127] B. W. Kernighan and R. Pike, *The Practice of Programming*. Addison-Wesley, 1999.

[128] J. Shirazi, *Java Performance Tuning*, 2nd ed. Sebastopol, CA, USA: O’Reilly & Associates, Inc., 2002.

[129] S. Wilson and J. Kesselman, *Java Platform Performance: Strategies and Tactics*. Boston, MA: Addison-Wesley, 2000.

[130] J. L. Bentley, *Writing Efficient Programs*, ser. Software Series. Englewood Cliffs, N.J.: Prentice-Hall, 1982.

[131] “Java anti-patterns,” 2010. [Online]. Available: http://www.odi.ch/prog/-design/newbies.php

[132] G. McCluskey, “Thirty ways to improve the performance of your javaâ„¢ programs,” 1999. [Online]. Available: ftp://ftp.glenmccl.com/pub/free/jperf.pdf

[133] P. Sestoft, 2005. [Online]. Available: www.dina.dk/~sestoft/papers/-performance.pdf

[134] A. Vetro', M. Torchiano, and M. Morisio, "Quantitative assessment of the impact of automatic static analysis issues on time efficiency," 2011. [Online]. Available: http://www.infq.it/doku.php/infq2011/papers

[135] K. Tian, Y. Jiang, E. Z. Zhang, and X. Shen, "An input-centric paradigm for program dynamic optimizations," *SIGPLAN Not.*, vol. 45, pp. 125–139, October 2010. [Online]. Available: http://doi.acm.org/10.1145/1932682.1869471

[136] D. Wampler, T. Clark, N. Ford, and B. Goetz, "Multiparadigm programming in industry: A discussion with neal ford and brian goetz," *Software, IEEE*, vol. 27, no. 5, pp. 61 –64, sept.-oct. 2010.

[137] M. Fowler, *Domain Specific Languages*, 1st ed. Addison-Wesley Professional, 2010.

[138] M. Eysholdt and H. Behrens, "Xtext: implement your language faster than the quick and dirty way," in *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*, ser. SPLASH '10. New York, NY, USA: ACM, 2010, pp. 307–309. [Online]. Available: http://-doi.acm.org/10.1145/1869542.1869625

[139] F. Seehusen and K. Stølen, "An evaluation of the graphical modeling framework (gmf) based on the development of the coras tool," in *Proceedings of the 4th international conference on Theory and practice of model transformations*, ser. ICMT'11. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 152–166. [Online]. Available: http://-dl.acm.org/citation.cfm?id=2022007.2022018

[140] C. Simonyi, M. Christerson, and S. Clifford, "Intentional software," *SIGPLAN Not.*, vol. 41, no. 10, pp. 451–464, Oct. 2006. [Online]. Available: http://-doi.acm.org/10.1145/1167515.1167511

[141] M. Volter, "From programming to modeling - and back again," *Software, IEEE*, vol. 28, no. 6, pp. 20 –25, nov.-dec. 2011.

[142] S. Dmitriev, "Language Oriented Programming: The Next Programming Paradigm," *onBoard*, vol. I, pp. 1–14, November 2004.

[143] M. Völter and E. Visser, "Language extension and composition with language workbenches," in *Proceedings of the ACM international conference companion*

on Object oriented programming systems languages and applications companion, ser. SPLASH '10. New York, NY, USA: ACM, 2010, pp. 301–304. [Online]. Available: http://-doi.acm.org/10.1145/1869542.1869623

[144] R.-H. Pfeiffer and A. Wasowski, "Taming the confusion of languages," in *ECMFA*, 2011, pp. 312–328.

[145] G. Procaccianti, A. Vetro, L. Ardito, and M. Morisio, "Profiling power consumption on desktop computer systems," in *Proceedings of the First international conference on Information and communication on technology for the fight against global warming*, ser. ICT-GLOW'11. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 110–123. [Online]. Available: http://dl.acm.org/citation.cfm?id=2035539.2035554

[146] A. Vetro', M. Morisio, L. Ardito, and G. Procaccianti, "Monitoring it power consumption in a research center: Seven facts," in *Proceedings of ENERGY 2011, The First International Conference on Smart Grids, Green Communications and IT Energy-aware Technologies*, 2011, pp. 64–69. [Online]. Available: http://porto.polito.it/-2388254/

[147] F. Khomh, M. Penta, Y. Guéhéneuc, and G. Antoniol, "An exploratory study of the impact of antipatterns on class change-and fault-proneness," *Empirical Software Engineering*, vol. I, pp. 1–33, 2012.

[148] V. Tiwari, S. Malik, and A. Wolfe, "Power analysis of embedded software: a first step towards software power minimization," *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, vol. 2, no. 4, pp. 437–445, 1994.

[149] H. Mehta, R. Owens, M. Irwin, R. Chen, and D. Ghosh, "Techniques for low energy software," in *Proceedings of the 1997 international symposium on Low power electronics and design*. ACM, 1997, pp. 72–75.

[150] T. Šimunic, L. Benini, G. De Micheli, and M. Hans, "Source code optimization and profiling of energy consumption in embedded systems," in *Proceedings of the 13th international symposium on System synthesis*. ÍEEE Computer Society, 2000, pp. 193–198.

[151] K.-D. Lange, “Identifying shades of green: The specpower benchmarks,” *Computer*, vol. 42, no. 3, pp. 95–97, Mar. 2009. [Online]. Available: http://dx.doi.org/-10.1109/MC.2009.84

[152] R. Jain, D. Molnar, and Z. Ramzan, “Towards understanding algorithmic factors affecting energy consumption: switching complexity, randomness, and preliminary experiments,” in *Proceedings of the 2005 joint workshop on Foundations of mobile computing*. ACM, 2005, pp. 70–79.

[153] N. Potlapally, S. Ravi, A. Raghunathan, and N. Jha, “A study of the energy consumption characteristics of cryptographic algorithms and security protocols,” *Mobile Computing, IEEE Transactions on*, vol. 5, no. 2, pp. 128–143, 2006.

[154] C. Bunse, H. Hopfner, E. Mansour, and S. Roychoudhury, “Exploring the energy consumption of data sorting algorithms in embedded and mobile environments,” in *Mobile Data Management: Systems, Services and Middleware, 2009. MDM’09. Tenth International Conference on*. IEEE, 2009, pp. 600–607.