Empirical Assessment of the Impact of Automatic Static Analysis on Code Quality

Empirical Assessment of the Impact of Automatic Static Analysis on Code Quality

Antonio Vetro’
Politecnico dii Torino
Corso Duca degli Abruzzi, 24
10129 Torino - ITALY
0039 011 5647169

antonio.vetro@polito.it

ABSTRACT

Background: Automatic static analysis (ASA) is performed on source code with different goals: improve important characteristics of code (such as maintainability), check a standard compliance or detect possible defects; therefore a substantial impact of ASA on software quality is expected. However, many problems related to their usage, especially the high number of false positives and the absence of evidence on their impact on code quality, could easily hinder the potential benefits of such tools.

Aims: Assess the impact of ASA issues (detections of ASA tools) on code quality by means of empirical analyses and controlled experiments on different software contexts.

Method: Goal Question Metric approach is used. Two main goals are defined: 1) Assess the precision of ASA issues (percentage of them actually related to real bugs) from the view point of a Java programmer; 2) Assess the impact of code refactoring based on ASA issues on ISO 9126 quality characteristics from the view point of a Java programmer.

Concerning the first goal, two strategies are defined in order to understand which issues are related to real defects: the first one is an experiment using source code and bug information coming from open source and possibly industrial projects, whilst the second strategy is a case study to be conducted during Object Oriented Programming Courses at the author’s university. At the end of the two experiments, those issues that were observed as linked to real defects will be included in the input set of defects prediction models: the accuracy of the models with/without ASA issues will be measured.

The second goal will be achieved through a controlled experiment in which the impact of the ASA issues will be evaluated independently for each ASA issue on the different ISO 9126 quality characteristics.

Results: Some preliminary results are already obtained from an empirical analysis on university Java projects: we observed that just a very limited set of issues have high precision and therefore can be considered as good defect predictors; conversely we identified those issues characterized by a such low precision that they can be considered as bad defects predictors in the context we studied.

Conclusions: Effective use of bug finding tools promise to speed up the process of source code verification. However, many problems related to their usage could hinder the potential benefits of such tools. The PHD plan aims at the comprehension of the impact of code refactoring from ASA issues on code quality, viewed both as defectiveness and a set of different characteristics. We expect to obtain, for each perspective of software quality, a reduced set of ASA issues whose impact in the quality characteristic is empirically proven. A practical application of this study is the adoption of the triage/taxonomy we want to develop by developers that try to find defects before testing or improve certain characteristics of their code (e.g. : maintainability, efficiency, etc.) . The PHD started on January 1st , 2010.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Code inspections and walk through

General Terms

Measurement, Experimentation, Verification.

Keywords

Bug finding tools, defects prediction, software quality.

1. INTRODUCTION

Software quality is crucial in many fields and quality assurance is a critical activity [1][2]. It is possible to adopt several techniques to improve quality: testing, code inspections, formal specification and verification. Although effectiveness and importance of these activities and methodologies is historically proved, there are important limitations, such as low easiness of use of formal specifications, necessity of having the system (or part of it) entirely built and working for testing, and low scalability of code inspections.

Given that the longer the delay of a fault insert-remove is [2], the higher the cost of removing that defect is and that testing and code inspections need a working code base, for this activities there is a consistent delay injection, that means costs and gap in reliability. ASA tools promise to speed up the verification process: they evaluate software in the abstract, without running it or considering a specific input. Such tools look for violations of recommended programming practice, conventions or standards (e.g.: MISRA-C), bug and design patterns, and they are able to automatically list all violations (*issues*, that are supposedly defects of the program that ought to be removed), statically analyzing source code or intermediate code (at compile time). Find bugs tools compensate the disadvantages of other techniques we listed above, because they’re very easy to use (it is just a matter of running the main and check the output), they are scalable (they can analyze thousands of lines of code in few minutes), and they could be used even in a non working code base. Despite the potential benefits listed above, several limitations were observed in literature and in the state of the practice. The most common problems are: the high number of false positives the ASA tools generate [12][15], the reduced subset of possible bugs that can be automatically detected [11][12] (for instance, defects related to requirements are not automatically detectable, unless requirements could be expressed in a formal way), the dubious efficiency of the default issues prioritization [7][17], the questionable economical benefits brought by their usage [9][11].

As a consequence companies and universities still need large empirical evidence of the efficiency of ASA tool.

The goal of the PHD is to contribute to the effort of the scientific community towards the assessment of the benefits deriving from usage of ASA tools and techniques. Our aim is to evaluate the impact of such technique on software quality.

In the next sections the following terminology will be used :

- ASA : Automatic Static Analysis
- ASA issue : rule/bug pattern/smell detectable by ASA tools
- detection : the single instance of the issues signalled on the source code. The relation ASA issue – detection is (1: N) .

2. RESEARCH GOALS, QUESTIONS, ISSUES

The first step of the PhD plan is the collection of existing empirical studies on static analysis and the recognition of the state of the practice (tools and methodologies). Activities related to this step are:

- Collect empirical studies in literature in the scope of ASA .
- Identify tools for ASA, and tools supporting refactoring.
- Identify university, open source or industrial projects suitable for the study. Necessary conditions: available source code and related process (defects) and product (structural metrics) data
- Identify in literature models for prediction of fault prone modules
- Identify in literature efficient inspection techniques

The next activities are the empirical experiments driven by the following goal: assessing the relationship between issues and software quality. We define software quality in two ways:

- software quality meant as software defectiveness (defects density)
- software quality meant as the set of 6 different characteristics, i.e. as described by the ISO-IEC standard 9126 (see Figure 1).

These two definitions are not exhaustive, and we consider them as approximations of software quality. For instance, the first definition has several issues, because it doesn't take into account the severity of bugs or their impact on usage scenarios (e.g. a bug in code which is rarely executed may not affect system quality as much as a bug in a very commonly used feature).

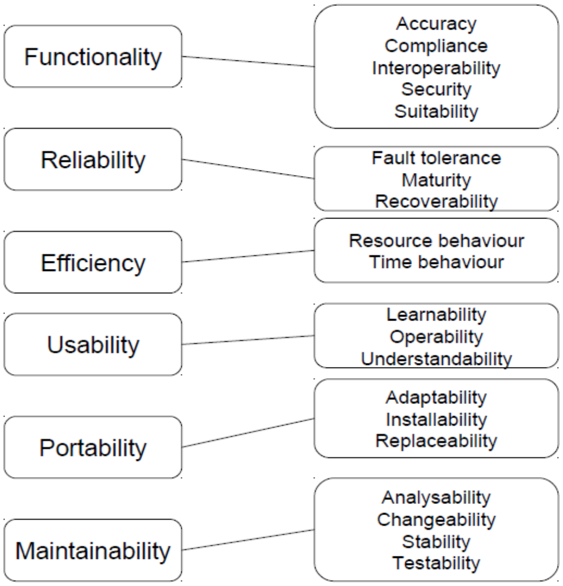

Figure 1. ISO-IEC 9126 Quality Model

We summarize the GQM in Table 1 and Table 2, then the detailed description of the GQM and of the experiments follow. At the end we give a graphical representation of the PHD plan objectives in Figure 2.

GOAL 1	
Purpose	Assess
Issue	the precision of
Object (process)	ASA issues
View point	from the view point of a Java programmer
RQ1.1	*Which ASA issues are related to real defects ?*
Metric M1.1	*Issue precision (detections related to defects/detections)*
RQ1.2	*Can ASA issues raise the accuracy of models to predict fault prone modules?*
Metric M1.2	*Precision, recall and F-measure of fault predictions*
RQ1.3	*Can ASA issues increase the generality of such models, not restricting them to specific context ?*
Metric M1.3	*Precision, recall and F-measure of fault predictions*

Table 1: Goal 1

GOAL 2	
Purpose	Assess
Issue	the impact of
Object (process)	code refactoring based to ASA issues on ISO 9126 quality characteristics
View point	from the view point of a Java programmer
RQ2.1	*What quality properties do ASA issues impact ?*
Metric M2.1	*To be defined: for each experiment different metrics will determine the impact of the issue on the different quality properties.*
RQ2.2	*Under what conditions ASA issues should be eliminated through refactoring ?*
Metric M2.2	*To be defined: for each experiment different metrics will determine the impact of the issue on the different quality properties.*

Table 2: Goal 2

3. RESEARCH APPROACH

The empirical analysis will be performed on different kinds of projects: small projects (mainly students projects, few hundreds of NCSS), and medium-large size projects, both open source and proprietary. For the first investigation, it is necessary that projects have both source code and bug database available.

<u>GOAL 1 : Assess the precision of ASA issues (percentage of them actually related to real bugs) from the view point of a Java programmer</u>

RQ1.1. Which ASA issues are related to real defects ?

Metric M1.1: issue precision (detections related to defects/detections)

We will perform two experiments to answer this research question computing the precision for each issue.

<u>Experiment 1.1</u>

Given a project's software repository (Java language will be preferred) and a bug database:

- analyze past defects, trace them to a source module, verify issues signalled on them. If yes the issue is assumed to have a positive effect
- analyze past changes, verify if a change has provoked the deletion of issue previously detected, if yes the issue is assumed to have a positive effect
- analyze code, find issues signalled, verify by manual inspection if defects are associated to these parts of code, if no the issue is assumed to have no effect

If the software project is small or no bug database is available, we will inspect manually the code or a portion of it (following the most efficient inspections found in the state of the art) and we will identify manually defects. Then we will run on the code one or more ASA tools, tracing:

- manual inspection time and inspection time of ASA issues (useful to evaluate the possible advantage in term of time spent in performing automatic inspection instead of manual inspection)
- defects identified by both manual inspection and ASA tool
- detections of ASA issues that are actually defects and are not identified by manual inspection
- detections of ASA issues that are not related to defects

The precision of issues will be available for each issue in both cases, at the end of the experiment: such ratio will be transformed in response 1/0 (issue is related to real defect: yes/no) using a threshold (e.g: *if* precision>50%, *then* output =1 *else* output=0) and different thresholds will be used to evaluate their effect on results through a sensitivity analysis. For each threshold *t,* issue *i* and precision p_i, the following null hypothesis will be tested :

H0 : $p_i < t$.

When hypothesis is rejected the issue is considered to be related to real defects .

<u>Experiment 1.2</u>

Softeng Research Group teaches 2 Object Oriented (OOP) Courses, in which Java language and Object Oriented Programming principles are explained, and where students develop small Java programs for the exam. The exam procedure is the following one:

1. Teachers define the project and provide the students with a textual description and a set of wrapper classes.
2. 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.
3. A tool on the server, PoliGrader, manages the delivery process and runs a suite of black box acceptance tests (JUnit classes). 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 tests coverage on it.
4. Results of test execution and their source code are sent back to the students.
5. Students improve the lab version at home, creating a new version of the program, called "home" version, that must pass all acceptance tests. This new version is submitted back to the server.
6. 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, quantifying the changes made to the lab version in order to pass all tests.
7. All information (marks, source code, tests, changes) is available to teachers in order to finally evaluate the students.

As a consequence of this process, for each student two versions of the same project are available: the lab version, that probably contains defects (revealed by tests failures), and the home version, that passes all tests and it's functionally correct.

We want to perform 2 experiments on the top of two consecutive sessions of OOP courses, each one including 2 parallel courses. In both sessions, only students of one course will be introduced to the ASA tool FindBugs [33], teaching them how to use it during classes and laboratories (students cannot change course inside the same session). The students will be required to minimize the number of FindBugs issues signalled in the Java projects. FindBugs is choosed because widely used in literature and since we already conducted research experiments with it [29] . In the first experiment, all FindBugs issues (more than 350 in current version 1.3.9) will be activated, while in the second experiment we will activate only those issues whose precision was empirically proved in previous empirical studies from the state of the art and in our works. To sum up, there are 2 exam sessions (the second one occurs after the first one), each of one having 2 courses :

- Session 1
  - Course A1 : no FindBugs
  - Course B1 : FindBugs – all issues activated
- Session 2
  - Course A2 : no FindBugs
  - Course B2 : FindBugs – only most precise issues activated

Students of session 1 will be different from students of session 2, and the same for projects: however, the difficulty level of the assignment and number and type of functionalities will be the same to make the two sessions comparable. Instead courses belonging to the same session will have the same projects requirements.

The following metrics will be collected from projects developed at the exam :

- FindBugs issues in both versions (lab, home)
- Test failures in lab versions
- FindBugs issues related to portion of code that is activated by tests failures
- Changes between lab and home versions
- Changes done expressly to delete FindBugs issues (students will be asked to provide this information)

Since we expect a large number of projects and issues, the relationship between defects (test failures) and issues will be investigated following the same procedure described in experiment 1, and a manual validation of a representative sample will be also performed.

Statistical analysis of data collected will permit to test the following null hypotheses:

- HA0 : External quality of projects of session 1 is different from external quality of projects of session 2
- HB0 : External quality of projects A1 is higher than external quality of projects B1
- HC0 : External quality of projects A2 is higher than external quality of projects B2
- HD0 : External quality of projects B1 is higher than external quality of projects B2

The external quality is measured as the percentage of tests failed: the lower is the percentage of failed tests, the higher is the external quality of a project.

The aim of HA0 is to test whether the Java programming capabilities of students of the two sessions are comparable (given that the 2 projects have the same difficulty level and the same number and type of functionalities to implement).

Possibly rejecting HB0 and HC0 will be instead the statistical proof that FindBugs issues are precise predictors of defects, therefore developing/refactoring a small project driven by FindBugs issues likely leads to a higher external quality of the code.

Additionally, rejecting fourth null hypothesis (HD0) will be a statistical proof that triage of issues is necessary to make usage of FindBugs more efficient.

Finally, further analysis of data let we investigate side aspects related to the main goal:

- number and type of FindBugs issues in home versions: these issues will be considered to be not related to functionality since home versions are functionally correct;
- correlation among FindBugs issues
- differences between code changes not related to FindBugs issues and changes made purposely to delete FindBugs issues.

D1.1. Deliverable

Activities related to the two experiments will produce a reduced set of ASA issues, actually linked to defects in the software projects we studied. We expect that a small portion of all issues are actually linked to defects.

RQ1.2. Can ASA issues raise the accuracy of models to predict fault prone modules?

RQ1.3. Can ASA issues increase the generality of such models, not restricting them to specific context ?

Metric M1.2: precision, recall and F-measure of fault predictions

Models to predict fault prone modules will be built to answer RQ1.2 and RQ1.3 ; the possibility to import existing defects prediction models from literature will be evaluated after state of the art activities. The dependent variable of model(s) is the likelihood the module is defect-prone: such probability will be transformed in response 1/0 (module defect prone: yes/no) using a threshold (e.g: *if* probability>50%, *then* output =1 *else* output=0). Different thresholds will be used and a sensitivity analysis will evaluate their effect on results.

Each model will be ran with two different sets of the inputs: the first one will be composed exclusively by traditional independent variables (source code metrics, past defects, etc.), whilst the second set of inputs will be obtained adding the information about ASA issues signalled on source code (number, type, etc...) to the previous variables, and considering only the reduced set of ASA issues identified in RQ1.1. Then the comparison between the accuracy of the two versions will indicate whether the ASA issues could improve the prediction power of such models or not. Summarizing, the procedure will be the following one:

- Define in detail what is meant by module in the project (function, file, class, other)
- Identify most fault prone modules by analyzing past defects
- Verify if, over time, the subset of fault prone modules remains the same or changes
- Build model(s). Input variables per each module: source code metrics, effort, other relevant information if available. Output variable: module is / is not fault prone.
- Verify models on past data: evaluate accuracy in predicting fault prone modules.
- Choose best model
- Enhance model integrating most precise ASA issues found in RQ1.1
- Verify model on past data evaluating its accuracy
- Compare precision of models with and without ASA issues
- Verify whether accuracy of both models types holds in different contexts (application type, open source/off the shelf, size,...)

<u>D2 - Deliverable:</u>

- Empirical assessment of the impact of ASA issues on the prediction power of defects prediction models
- A prediction model for faulty modules, using as input source code metrics, defects, effort, and in case ASA issues

<u>GOAL 2 : Assess the impact of code refactoring based to ASA issues on ISO 9126 quality characteristics from the view point of a Java programmer.</u>

RQ2.1. What quality properties do ASA issues impact ?

RQ2.2 Under what conditions they should be eliminated through refactoring ?

Metrics to be defined: for each experiment different metrics will determine the impact of the issue on the different quality properties.

The goal is to evaluate the impact of ASA issues on quality characteristics referring to ISO-IEC 9126 model. No author has, until now and up to our knowledge, proposed a comprehensive taxonomy of ASA issue – code quality property impacted based on empirical experimentation.

We identify three steps in the empirical experimentation. Given a set of ASA issues I and the set of quality characteristics C, the procedure is the following one:

1. *for each* $i \in I$ *and each* $c \in C$ *do*
    - determine manually by expert judgement whether issue *i* could impact characteristic *c*. In this context any of the following is considered as expert: a professor, a research assistant or a PHD student, that teaches in a University course the programming language used in the experiment; a programmer from industry that programs frequently with the programming language used in the experiment (e.g: Java programming for at least 3 days a week in the last 3 years). In the case multiple experts disagree, the impact is considered as not present. An impact matrix IM will store experts judgements, while a second matrix (validation matrix VM) will trace empirical validations of the judgements. Matrices are initialized in the following way:
    - *for each* $i \in I$ *and each* $c \in C$ *do*
        - *if* issue *i* is thought by experts to have impact on characteristic *c* *then* IM_{ic} =1 *else* IM_{ic} =0
    - *for* each $i \in I$ and each $c \in C$, *do*
        - *if* IM_{ic}=0 *then* VM_{ic} = 0 else VM_{ic} = -1 ;
2. Experimentation*:*

    for each $i \in I$ *and each* $c \in C$ *where* VM_{ic}= -1 *do*

    - plan one or more experiments where the impact on *c* of having/not having issue *i* can be measured. We provide the reader with an example of experiment; let's consider the FindBugs issue Inefficient_Integer_Constructor (IBC) that is signalled when the `new Integer()` constructor is used instead of `Integer.valueOf()`. It is expected that issue IBC impacts characteristic Efficiency. A possible experiment to verify the impact is :
        - write a piece of code where issue IBC is signalled;
        - refactor the previous piece of code obtaining a second version, functionally identical, but where the issue IBC is no longer signalled;
        - prepare the environment in which to run both versions of the code: the environment should be as much isolated as possible (e.g: disable network, disable routine tasks, no other programs running...), in order to minimize the possibility that executions of the code are interrupted/delayed by other programs/routines;
        - run a very high number of times (to eliminate random effects - e.g.: 1000000) the two codes (independently and sequentially), measuring their total execution time;
        - compare the two measurements by statistical analysis: if code without IBC performs better, its impact on Efficiency is demonstrated and confirmed.
    - *if* impact of *i* on *c* is empirically demonstrated, *then* VM_{ic} =1 *else* VM_{ic} =0.

Having validated the taxonomy, it is possible to define a set of heuristics to improve a specific property of source code, using the most effective related refactoring.

D3 - Deliverable:

- A validated taxonomy of the relationship ASA issue – quality property
- A prioritization of ASA issues and a set of heuristics for refactoring code, in function of the software quality property impacted

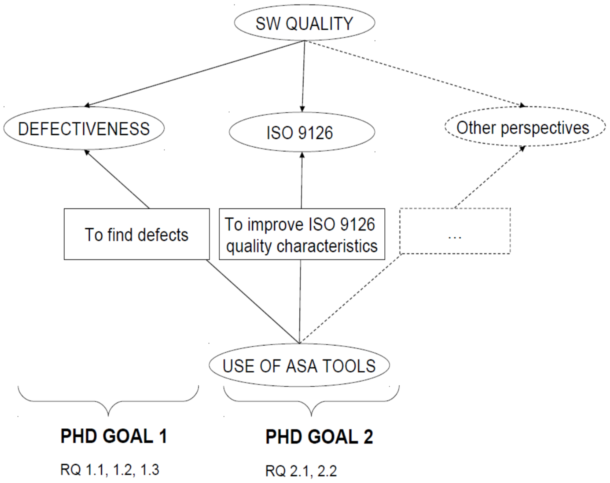

Figure 2. PHD Plan Overview

4. INITIAL RESULTS

In our previous work [29] we faced Research Question 1.1 for the issues of the ASA tool FindBugs v1.3.8, signalled on 85 Java assignments from the 2009 OOP course. We considered issues grouped according to two dimensions: category (Bad Practice, Correctness, Style, Performance, and Malicious Code are the categories with at least one issue signalled in our code base) and priority (Low, Medium, High). For each issue group (combination of category and priority) we computed its precision. The number of detections related to defects was determined through the Spatial + Temporal technique, previously presented in literature in [5]. The technique is the following one: 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, as explained in Goal 1 - Experiment 2, defects fixed are revealed when a test that in lab version fails instead in home version succeeds. The possibility that a disappearing issue was not related to the disappearing defect is the noise of this metric, that is filtered out by adding 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. In practice, the combination of temporal and spatial coincidence is interpreted as a change intended to remove the issue, that is linked to the defect.

Defining precision p of the issue group g as p_g , we decided that an issue was related to real defects if the null hypothesis that $p_g < 30\%$ was rejected. Such a low threshold was justified by the exploratory nature of the work and it compensated for the large precision variability in each group. The analysis of precision measures obtained demonstrated that only 2 out of 15 groups of issues could be considered as related to actual defects in our code base. Moreover, one group of issues had a precision that was practically negligible.

In a subsequent and more detailed experiment, still not published, we repeated the study enlarging the code base (we analyzed 301 Java Projects) and computing precisions at single issues level (instead of groups). We observed again that few issues were related to real defects. The precision threshold used in this study was stricter (50%), but the sensitivity analysis demonstrated that results held for precision > 21% (that is a very low threshold). As a consequence we can consider results stable.

The findings of this second study are :

- The 20% of issues made the 80% of total detections, and just 5 issues represented about half of total detections.
- 80% of detections were related to 5 categories of problems: objects and references, violation of naming conventions, no effect of fields, variables or methods .
- Distribution of issues among Java classes was inhomogeneous: detections were concentrated in a few classes.
- The analysis of issues precisions demonstrated that only 4 issues could be considered as reliable predictors of real defects in the context the research was conducted .
- The same analysis let us identify 16 issues whose precision was practically negligible: they were responsible of about the 45% of issues detections.

We will try to extend prior work repeating the study both in industrial and open source systems, and facing also research question 1.2.

5. THREATS TO VALIDITY

Here we identify external and internal threats for each planned experiment .

Goal 1- Experiment 1.1 In the case the fault database is available, the major construct thread is the fact that the relation between issues and defects is determined with statistical techniques, checking whether issues are signaled in lines that changed due to a faults located on them. In order to control it, the manual inspection of links faults-issues should include a significant number of samples, and one or more reviewer should validate it. On the contrary, if the fault database is not available, manually find a significant number of defects by means of manual inspection could be a long and error prone task (internal threat): having one or more reviewers will permit to control the threat also in this case.

Goal 1 – Experiment 1.2. An important 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 experiment, differently to many others in the literature, eliminates the effect of developer style on the results, because a large pool of developers is used for the same projects. A further threat is a construct one: it is concerning the identification of defects. In this second experiment, no bug database is available: we make the assumption that all changes between lab and home version are done in order to fix a defect; actually, 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). 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:

- the home version is the last version of the project, actually no maintenance has to be done subsequently;
- students are discouraged in doing many changes, because the mark suggested by PoliGrader decreases with the quantity of changes made (see details in [34]).

Goal 2 – Experiments 2.x. We observe two important internal threats related to the experiments that should verify the impact of an issue on a code quality characteristic. The first one is the difficulty to minimize the noise introduced by different confounding factors in the experiments results (e.g.: measure the performance of two different pieces of code inside an operating system could be highly affected by the presence of other running processes). The second threat is: for characteristics such as maintainability and usability, it could be hard to find non-subjective metrics: for instance, if the maintainability is measured computing 2 maintenance tasks performed by 2 different teams, the programmers capability could be a deviant factor. This threat could be controlled repeating the experiment with different teams. We plan to control these threats increasing the number of samples and executing the experiments in different machines / operating systems.

6. RELATED WORK

In this section we want to report the work already done in literature in the scope of the PHD plan. The present section is a partial result of the step 1 of the PHD plan, that is the recognition of state of the art, that will continue in the following months. The related work we report here mainly refers to Goal 1 – RQ1.1 .

The first research we cite is the one of Boogerd and Moonen [5]: they tried to empirically assess the relation between violations of coding standard rules and actual faults. They analyzed the relationship between defects in an industrial software project and MISRA-C 2004 rules violations, using history of project (code versions) and problem report database. They introduced the concepts of temporal and spatial coincidence: if a fault disappears from one version to another one, and a violation too, probably that violation was related to the fault that disappeared; this is temporal coincidence. However, temporal coincidence is affected by noise: some violations can disappear as a consequence of a change in a portion of code not directly related to it or to another defect. Authors decided to assess the impact of this noise by means of spatial coincidence: using information from Software Configuration Management System and from problems reports database, they were able to individualize only the lines changed related to a fault that disappeared from two consecutive versions, and to count as effective only that violations that were in those lines of code. These violations were considered as true positive. The experiment of Boogerd and Moonen showed that a reduced set of rules (12 over 72) performed significantly better (true positive rate between 23% and 100 %), while about one third of the rules (25 over 72) had zero positive rate: so, taken together with Adams' law [13] , it means that it is better to don't apply the smell. They repeated the experiment with another system [28], however, the set of efficient rules found in [5] have a few intersections with the one found in [28].

Boogerd and Moonen tried to assess the value of ASA issues for fault detection by means of statistical inference, guessing that a violation in a defect fixing changed line is really related to that issue. On the contrary, Pugh et al. [6] tried to understand the efficiency of the static analysis tool FindBugs by manually checking the issues signalled on projects: experiments showed higher true positive ratios. The authors classified issues in 4 categories, based on their impact on code. They got that, in JDK 1.6.0-b105, almost 50% of medium/high priority issues of category correctness had impact, and 10% had a serious impact. 160/379 were trivial, while 5 issues were due to bad analysis of FindBugs. A similar experiment with the same category of issues was performed at Google, with similar high percentages of true positive issues, while a further experiment conducted on Glassifish v2 showed that 50 defects over 58 disappeared due to small edits designed to specifically address the issue raised by FindBugs.

High percentage of true positive found by FindBugs were also found by Cole et al. [7], asserting that the rate of false warnings reported by FindBugs is generally lower than 50% but they didn't prove it. Almossawi, Lim and Sinha [8] provided a third party evaluation of Coverity Prevent and reported an overall evaluation that 13% of the warnings seemed be infeasible and 64% seemed very likely to result in faults.

Wagner et al. [9] evaluated FindBugs and PMD on two software projects. They found out that very few of defects (post-release and actually correlated to documented failures in deployed software) were identified by these tools. On a total of 91 defect removals, comparing two successive versions of software, they found only 4 that corresponded to remedying a problem that caused a failure in the deployed software.

Robert OCallahan told in his blog[1] about his experience with Klocwork and Coverity, and noted that many of the defects found did not seem to be significant. Konstantin Boundnik also blogged[2] about his experience with Klocwork and Coverity, asserting that false positive rates were respectively 10 and 15%. In order to contrast the false positive ratio, Hovemeyer et al. [10] introduced the use of Java annotations by the programmer in order to explicitly inform FindBugs about which values must not be null and which on the contrary may be null. They performed experiments on both production software and students projects, obtaining false positive rates of 20% on production software and identifying 50% to 80% of issues dealing with null pointer exceptions at run time in students projects.

Zheng and al. [11] tried to go deeper in the topic of ASA tools efficiency, answering to the question: for which kind of defects are ASA tools effective? They performed a study on static analysis faults, tests and customer-reported failures for three

[1] See http://weblogs.mozillazine.orgrocarchives200609static analysis and scary head.html

[2] See http://weblogs.java.netblogcos archive200610static analyzer.html

large-scale industrial software systems developed at Nortel Networks. They found that automated static analysis is effective at identifying low level bugs like assignment and checking faults, and that they're complimentary to the other fault detection techniques. They also asserted that automatic static analysis defect removal yield is similar to the one of inspections and that the faults signalled by automatic static analysis tools are good fault predictors; however, the hypothesis that quality with automatic static analysis is higher than without was rejected. Another study attempted to understand the added value of ASA tools with respect to the traditional fault detection techniques: Wagner et al. [12] compared bug finding tools with reviews and tests. They found that bugs signalled by smell tools are a subset of those found by reviews, and different from dynamic testing. The authors analyzed 5 projects and 5 categories of defects (with respect of their effect on system behaviour). They manually determined the ratio of false positive issues, and results were : 47% for FindBugs, 31% for PMD, 96% for QJPro. This result suggest another consideration: are bug finding tools able to predict the same defects ? So, are they equivalent ?

A study conducted by Rutar et al. [13] classified issues of 5 different ASA tools (JLint, Bandera, ESC/Java, FindBugs, PMD), and ran them on different projects. They were able to cross-check their issue reports and warnings, and they discovered that tools generally find non-overlapping issues. Finally, they proposed a meta tool for combining the results of different smell tools, in order to cover all the bugs categories.

A similar conclusion was found by Wagner et al. [14] : authors compared FindBugs and PMD, running them on 2 industrial projects. They demonstrated that the 2 tools have a small intersection of issues, and they suggested to use them in combination. They also studied the economical efficiency of the usage of ASA tools: estimating direct and indirect costs, they asserted that, in order to be cost efficient, the tools must individualize 3-4 potential field-defects. Economical considerations were also done by Zheng et al. [11]: the cost of automatic static analysis per detected fault is of the same order of magnitude as the cost of inspections per fault detected. Coming back to Wagner's research, he also studied the technical efficiency of the tools : even if the number of issues could be a good predictor for faulty modules (again, as in [11]), the true positive ratio of smell tools issues in his experiment is very low. Analysing a sample of 72 real bugs from the bug repository of the projects, Wagner et al. found out that in the first project none of the bugs was revealed by the static analysis tools, while in the other one just 16% of them was found. They motivated this results saying that the majority of issues had its roots in the semantic. The same last finding is present in Zhenmin Li et al. [15]. The authors tried to identify which bugs categories are present in Mozzilla and Apache software, automatically analyzing Bugzilla repositories by means of natural language text classification. Results of their study showed that the majority of bugs were semantic, thus they could be revealed only by dynamic testing. Yet another finding concerned the presence of a lot of simple memory related bugs such as NULL pointer references (12.2 - 16%), that should have been detected by static analysis tools : this indicates that the tools have not been used with their full capacity.

This last issue (how ASA tools are used) was investigated by Ayewah et al. [16], too. The authors experienced in Google that the more the usage of the tool appears within the code production work flow, the higher the efficiency of the tool is: when they were able to incorporate FindBugs in the review process, more than 200 users suppressed thousand of warnings signalled by FindBugs in 6 months. However, they observed that the situation out of Google is quite different. Authors tried to understand how FindBugs is used by means of a survey with tool's users: they discovered that the 81% of users have no exact policies in their companies on how to run FindBugs, and in the 76% of the cases, running FindBugs was not required by the process. However, the majority of users reviews at least high priority issues (thus it seems that these are the most e effective issues), but generally they don't have policies for handling issues designated as not real bugs, or low priority warnings.

Management of issues priorities is also discussed by Kim and Ernst [17]. They combined, in their experiments on 3 open source projects and 3 bug finding tools, source code changes and bugs information from bug repository, individualizing which lines contain fix changes (if the changed line contains a bug disappeared from a version to the following one, the change is a fix change), checking the ratio of ASA issues in those lines. The experiment showed that very low percentages of warnings were removed by a fix change, and yet lower percentages considering only high priority warnings (6%, 9% and 9%). So they proposed an algorithm based on the history of removed warnings in order to modify priority of issues, and repeating the experiment with the new obtained priorities, they were able to have higher percentages of high priority issues removed by fix changes (up to 17%, 25% and 67%).

Even if the first static analysis tool appeared more then 30 years ago (Lint, by Stephen Johnson of Bell Labs [18]) the research field is wide and literature doesn't fit all open issues and contexts. For instance, studies on ASA tools in university context is one of them, and this is the field in which I and my supervisor conducted the first experiments. We found two examples of researches conducted in University in a similar scope. Hristova et al. [19] noticed that students of Java introductory courses usually do common errors: they created a small taxonomy of them and they built an educational tool able to capture these common errors, providing additional information to the error messages of the Java compiler. Starting from the same consideration, Truong et al. [20] built a whole framework to check common students' errors and help them in writing better code. The underlying idea that we liked is that such common errors can be automatically detected, but in our opinion there's no need to build new tools or frameworks: several ASA tools already exist, they can be customized and used in an educational context. The achievement of the goals of this PHD plan will give the possibility to better customize the tools and to activate only issues with higher precision. A further study in the university environment was done by Jaime Spacco, Jaymie Strecker, David Hovemeyer, and William Pugh in the context of the Marmoset Project [24] [25] [26] [27]. Marmoset is a platform that allows students to submit versions of their Java projects to a central server, which automatically tests them and records the results. It also collects code snapshots of the projects: each time a student saves the work, it is automatically committed to a CVS repository. They declared correlations between warnings signalled by FindBugs and tests outcomes, and between three types of FindBugs issues (ClassCast, StackOverow, Null Pointer) and Java Exceptions, but they didn't do any deep statistical analysis. Precision and recall can be obtained by their tables: precisions are, with respect to issue type, 23%, 71%, 45% , and recall values are 28%, 44%, 18%.

Furthermore, we would like to cite some study that investigated the correlation between smells and code quality. Code smell is any symptom in the source code of a program that possibly indicates a deeper problem: something that is not working properly or a problem that could be resolved in others way, simplifying the code, making it faster or decreasing the resource usage. From the point of view of a programmer charged with performing refactoring, code smells are heuristics to indicate when to refactor, and what specific refactoring techniques to use. Thus, a code smell is a driver for refactoring [3], and refactoring improves code maintainability and flexibility. The debate about automatic detection of code smell is still open in literature, and it will be investigated. We found a work by Li and Shatnavi [21] in which they investigated the relationship between the class error probability and bad smells based on three versions of the Eclipse project. Their result showed that classes which are infected with the code smells Shotgun Surgery, God Class or God Methods have a higher class error probability than not infected classes.

Deligiannis et al. investigated the impact of God Classes, based on Riels Definition [22] on the maintainability of object oriented design [23]. They built two functional identical software systems, one containing the god class smell (Design B) and the other one without (Design A). An identical maintenance task was assigned to two student groups (Group A and Group B), whereas one group had to perform the task on Design A and the other on Design B. After the study, a questionnaire was performed, and analysis of answers questionnaire showed that the students of group A had fewer problems with the understandability of the design as well as the modification of the design. Furthermore, a quality analysis was performed on both the delivered solutions, showing differences towards the 2 solutions.

Finally, in several empirical studies (we cite [30], [31] and [32]), static analysis issues are used in conjunction with code and history metrics as factors in defects prediction models: since while we're writing the PHD in step 1, we still have to explore this area of the field.

7. AKNOWLEDGEMENTS

I would like to thank my supervisors Maurizio Morisio and Marco Torchiano for their precious advices, and the reviewers of IDOESE 2010 for their observations that let me improve my PHD plan.

8. BIBLIOGRAPHY

[1] B. W. Boehm. Software process management: lessons learned from history. In ICSE '87: Proceedings of the 9th international conference on Software Engineering, pages 296{298, Los Alamitos, CA, USA, 1987. IEEE Computer Society Press.

[2] Barry Boehm and Victor R. Basili. Software defect reduction top 10 list. Computer, 34(1):135-137, 2001.

[3] M. Fowler, K. Beck, J. Brant, and D. Roberts. Refactoring: improving the design of existing code.

[4] Edward N. Adams. Optimizing preventive service of software products. IBM Journal of Research and Development, 28(1):214, 1984.

[5] C. Boogerd and L. Moonen. Assessing the value of coding standards:An empirical study. Software Maintenance, 2008. ICSM 2008. IEEE International Conference on, pages 277-286, 28 2008-Oct. 4 2008.

[6] Nathaniel Ayewah, William Pugh, J. David Morgenthaler, John Penix,and YuQian Zhou. Evaluating static analysis defect warnings on production software. In PASTE '07: Proceedings of the 7th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering, pages 18, New York, NY, USA, 2007. ACM.

[7] Brian Cole, Daniel Hakim, David Hovemeyer, Reuven Lazarus, William Pugh, and Kristin Stephens. Improving your software using static analysis to find bugs. In OOPSLA '06: Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications, pages 673-674, New York, NY, USA, 2006. ACM.

[8] Ali Almossawi, Kelvin Lim, and Tanmay Sinha. Analysis Tool Evaluation: Coverity Prevent. Technical report, Carnegie Mellon University,May 2006.

[9] Stefan Wagner, Florian Deissenboeck, Michael Aichner, Johann Wimmer, and Markus Schwalb. An evaluation of two bug pattern tools for java. In *ICST '08: Proceedings of the 2008 International Conference on Software Testing, Verification, and Validation*, pages 248–257, Washington, DC, USA, 2008. IEEE Computer Society.

[10] David Hovemeyer, Jaime Spacco, and Bill Pugh. Evaluating and tuning a static analysis to find null pointer bugs. Lisbon, Portugal, September 5-6, 2005. ACM.

[11] J. Zheng, L. Williams, N. Nagappan, W. Snipes, J. P. Hudepohl, and M. A. Vouk. On the value of static analysis for fault detection insoftware. Software Engineering, IEEE Transactions on, 32(4):240-253,2006.

[12] Stefan Wagner, Jan Jrjens, Claudia Koller, Peter Trischberger, and Technische Universitt Mnchen. Comparing bug finding tools with reviews and tests. 2008.

[13] Nick Rutar, Christian B. Almazan, and Jerey S. Foster. A comparison of bug finding tools for java. In ISSRE '04: Proceedings of the 15th International Symposium on Software Reliability Engineering (ISSRE'04), pages 245-256, Washington, DC, USA, 2004. IEEE ComputerSociety.

[14] S. Wagner, F. Deissenboeck, M. Aichner, J. Wimmer, and M. Schwalb. An evaluation of two bug pattern tools for java. In Software Testing, Verification, and Validation, 2008 1st International Conference on, pages 248-257, 2008.

[15] Zhenmin Li, Lin Tan, Xuanhui Wang, Shan Lu, Yuanyuan Zhou, and Chengxiang Zhai. Have things changed now? An empirical study of bug characteristics in modern open source software. In ASID '06: Proceedings of the 1st workshop on Architectural and system support for improving software dependability, October 2006.

[16] Nathaniel Ayewah, David Hovemeyer, J. David Morgenthaler, JohnPenix, and William Pugh. Using static analysis to find bugs. IEEE Software, 25(5):22-29, 2008.

[17] Sunghun Kim and Michael D. Ernst. Which warnings should I fix first ? In ESEC-FSE '07: Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering, pages 45-54, NewYork, NY, USA, 2007. ACM.

[18] S. C. Johnson. Lint, a c program checker. In COMP. SCI. TECH.REP, pages 78-1273, 1978.

[19] Maria Hristova, Ananya Misra, Megan Rutter, and Rebecca Mercuri. Identifying and correcting java programming errors for introductory computer science students. In SIGCSE '03: Proceedings of the 34th SIGCSE technical symposium on Computer science education, pages 156, New York, NY, USA, 2003. ACM.

[20] Nghi Truong, Paul Roe, and Peter Bancroft. Static analysis of students' java programs. In ACE '04: Proceedings of the sixth conference on Australasian computing education, pages 317-325, Darlinghurst, Australia, Australia, 2004. Australian Computer Society, Inc.

[21] Wei Li and Raed Shatnawi. An empirical study of the bad smells and class error probability in the post-release object-oriented system evolution. J. Syst. Softw., 80(7):1120-1128, 2007.

[22] Arthur J. Riel. Object-Oriented Design Heuristics. Addison-WesleyLongman Publishing Co., Inc., Boston, MA, USA, 1996.

[23] Ignatios Deligiannis, Martin Shepperd, Manos Roumeliotis, and Ioannis Stamelos. An empirical investigation of an object-oriented design heuristic for maintainability. J. Syst. Softw., 65(2):127-139, 2003.

[24] Jaime Spacco, David Hovemeyer, and William Pugh. An eclipse-based course project snapshot and submission system. In 3rd Eclipse Technology Exchange Workshop (eTX), Vancouver, BC, October 24, 2004.

[25] Jaime Spacco, Jaymie Strecker, David Hovemeyer, and William Pugh. Software repository mining with Marmoset: An automated programming project snapshot and testing system. In Proceedings of the Mining Software Repositories Workshop (MSR 2005), St. Louis, Missouri, USA, May 2005.

[26] Jaime Spacco, David Hovemeyer, William Pugh, Je Hollingsworth,Nelson Padua-Perez, and Fawzi Emad. Experiences with marmoset: Designing and using an advanced submission and testing system for programming courses. In ITiCSE '06: Proceedings of the 11th annual conference on Innovation and technology in computer science education. ACM Press, 2006.

[27] Jaime Spacco, David Hovemeyer, Bill Pugh, J Hollingsworth, Nelson Padua-Perez, and Fawzi Emad. Experiences with marmoset. Technical report, 2006.

[28] Cathal Boogerd and Leon Moonen. Evaluating the relation between coding standard violations and faults within and across software versions. In *MSR '09: Proceedings of the 2009 6th IEEE International Working Conference on Mining Software Repositories*, pages 41–50, Washington, DC, USA, 2009. IEEE Computer Society.

[29] A. Vetro', M. Torchiano, and M. Morisio. Assessing the precision of findbugs by mining java projects developed at a university. In IEEE CS Press, editor, *Proceedings of MSR 2010*, pages 110–113, 2010.

[30] Sarah Heckman and Laurie Williams. A model building process for identifying actionable static analysis alerts. In *ICST '09: Proceedings of the 2009 International Conference on Software Testing Verification and Validation*, pages 161–170, Washington, DC, USA, 2009. IEEE Computer Society.

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

[32] Joseph R. Ruthruff, John Penix, J.David Morgenthaler, Sebastian Elbaum, and Gregg Rothermel. Predicting accurate and actionable static analysis warnings: an experimental approach. In *ICSE '08: Proceedings of the 30th international conference on Software engineering*, pages 341–350, New York, NY, USA, 2008. ACM.

[33] HOVEMEYER, D., AND PUGH, W. Finding bugs is easy. In *OOPSLA '04: Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications* (New York, NY, USA, 2004), ACM, pp.132–136.

[34] M.Torchiano and M.Morisio. A fully automatic approach to the assessment of programming assignments. *INTERNATIONAL JOURNAL OF ENGINEERING EDUCATION*, 24 (4)(0):814–829, 2009.