STATIC CODE ANALYSIS IN JAVA
Bachelor Thesis

Fabian van den Broek

Supervisor: dr.ir. Erik Poll

Radboud University, Nijmegen

Abstract. Software contains bugs and bugs cost money. A good way to find some bugs quickly is the use of static code analysis. There are no exact numbers on the use of static code analyzers in the industry, but in our experience to few software developers actually make use of them. This thesis describes a survey that was conducted to find out why only so few developers in the Java community use static code analysis. Furthermore, a case study is performed to investigate the advantage of using a static code analyzer for Java, namely FindBugs, on a real world application.

1 Introduction

Software inevitably contains bugs and bugs can cause software freezing and malfunctioning and decreased security. The consequences of bugs range from waste of capital to patients' deaths. Many techniques to deal with bugs have been developed, such as testing, debugging and expert code reviews. A lot of bugs are the result of common coding mistakes. For example, the eponymous buffer overflow problem in C is a common mistake and in Java many mistakes are made with Object comparison using '=='.

It is commonly known that the earlier bugs are found, the less it costs to fix them. Static code analyzers are tools designed to find many types of common made mistakes. Using static code analysis throughout development results in finding certain bugs at an early stage. Other uses of static code analysis are guiding expert code reviews [1] and in programming education [2]. Static code analyzers have existed for a couple of decades and have steadily improved. Nowadays, they excel at spotting common problems and tracing variables throughout source code. The use of static code analysis generally reduces the number of bugs [3], improves the security of an application [4], and improves overall code quality [5].

However, little information is known about how often static code analyzers are used during software development. In my experience, software developers judge static code analysis useful, yet they do not use them. An explanation for this may be found in a commonly heard complaint of developers about static code analysis. Namely, static code analysis produce many false positives. However, it remains unclear whether this conception is true.

This thesis investigates the use of static code analyzers among Java developers and the number of usefull warnings produced by the static code analyzer FindBugs. More specifically, the following research questions were addressed concerning the use of

static code analyzers: what percentage of the developers is aware of the existence of static code analysis, why do developers choose not to use static code analysis, how often are static code analyzers used during software development, and how do developers experience static code analysis.

In section 2, a short introduction to static code analysis is given. Section 3 describes the survey that was conducted in the Java community to discover the attitudes of Java developers towards static code analysis. Next, section 4 describes a case study of the number of false positives produced by the SCA tool FindBugs. Finally, conclusions are provided in section 5 and some recommendations for future work in section 6.

1.1 Related work

There has already been some research into static code analysis lately. In [3], Rutar et al. compared five static code analyzers by running them over several Java projects and comparing the output and the run times. They conclude that there is quite a bit of difference between the bugs these tools find and suggest the use of a meta tool combining multiple static code analyzers.

FindBugs is an open-source static code analyzer created by the University of Maryland. On their website they are now performing a survey of their own, to find out how FindBugs is used [6].

In [7,8] Kim and Ernst have tried to improve the priorization that static code analyzers use to present their reported bugs. They measured the time certain bug categories remain within a software project in order to find out which type of bugs are more important for developers.

2 Static code analysis

Static code analysis is the analysis of computer software without actually running the software. Usually this is done automated by another program. This article is about using this analysis technique to automatically search for bugs and improve source code. There are numerous tools around, both commercially and free, which do this.

In the late seventies the Lint tool became popular. It was a static code analyzer for the C programming language which looks though the source code for certain code patterns of often made mistakes. This tool was named after the undesirable bits that gather in sheep wool because of static electricity. From here on static code analysis has seen many tools for many languages, each with different success.

Static code analysis can be used for more then finding bugs. The most widely used application of static code analysis is most likely type checking, which is found in nearly every compiler of every strongly typed language. Type checking is also performed without actually running the program. This article however will only refer to the automated scanning of bugs within code when mentioning static code analysis.

Most static code analyzers nowadays read the code and construct an abstract model from this. In this abstract model they can then look for known bug patterns. There are also several static code analyzers that complement these checks with some form of data flow analysis [9]. Using data flow analysis these tools can track possible values for

variables at different points in the program and follow possibly ‘tainted’ input through a program.

Static code analysis is neither sound nor complete. That is to say no static code analyzer is able to find only bugs (and no false positives) nor is it able to find all bugs present in a program. This is in fact an example of the more general result known as Rice’s theorem, which informally states that no program is able to verify for which programs a certain property holds. The problem of static analysis in fact is undecidable, but like all undecidable problems it is still possible to find useful approximations.

Because these tools are not exact, they will have false positives (a bug report that does not signify an actual bug) and false negatives (undetected bugs). The rates of false positives and negatives is a good measure to compare static code analyzers [3]. Other important criteria can be the overall usability and the ability to modify the behavior of these tools (extending or reducing the bug detectors) or the number of bug detectors that these tools offer.

2.1 When to use static analysis?

There are basically two moments or situations were a static source code analyzer can be useful. The first is during testing or when auditing source code, the second is during development and before testing. In the first situation static code analysis is often used to get some idea of the quality of the source code or as a guide for a code reviewer. In the second situation programmers employ static code analysis after compilation and before testing. Both approaches will demand different things from a static code analyzer. For usage during an audit users will typically not mind a larger false positive rate if this means that more bugs are found (lower false negative rate). Most programmers who use these tools will mostly be glad getting rid of a few bugs and not receiving a large amount of false positives.

Most static analysis tools can be tailored to be use in both situations, however there are good reasons to promote the use of static code analyzers during development and before testing. The most important reason is that when a static code analyzer is used in this way it will typically keep the time between the creation of the bug and finding them smaller. This is useful because the fixing of bugs costs more, the later they are discovered.

Another reason is that using a static code analyzer during the implementation of a program will likely lead to less bug reports per run then only using these tools during testing. Keeping the number of bug reports manageable will make the number of false positives also less painful.

2.2 Static code analysis in Java

This paper will only look at static code analysis for the Java language. The difference between scanners for different languages is not in the methods of scanning, but in the types of bugs they look for. Buffer overflows is a typical example of a kind of bug any C static analyzer will look for, but analyzers for the Java language will not be interested in. This however does not mean that all conclusions on static code analysis are generally applicable for all tools. The bugs from a specific language can be easier to find then

those of another. Similarly the results of this paper can not simply be applied to all static code analyzers. The survey of section 3 is aimed at the Java community, and it is unclear to what measure the results will differ when the same questions are directed to programmers who use different languages. Also in section 4 a Java static code analyzer is used on a Java project and these results

There are two sorts of static code analyzers for Java; those that scan the source code and those that scan the byte code. Both approaches have their benefits. Scanning the source code means you scan the actual code written by the programmer. Compilers tend to optimize code and a bug found in the resulting byte code might not be easily translatable to the defect in the source code. Also you do not have to compile the source code before you can start checking it with static code analysis. But scanning the byte code is often a lot faster then scanning through the source code. On large projects it may be vital to keep the performance hits on static code analysis low.

3 Attitude towards static analysis tools

In this section we try to determine the attitude of Java programmers towards static analysis of Java code.
The remainder of this section will look closer at the survey, how it was set up, its results and some discussion on those results. A copy of the actual survey can be found in appendix A.

3.1 Setup

To answer various aspects on the use of static code analyzers several smaller questions were asked. A precondition for using static code analyzers is awareness of their existence. Consequently, the first question on the survey is to find what percentage of the participants is aware of the existence of static code analyzers. The next question investigates why people who are aware of the tools decide not to use them. This provides insights into the appearance of static code analyzers and prejudices around them. To discover the popularity of particular static code analyzers, the next question is which static code analyzers are used. Finally, the last question investigates what problems are encountered when using static code analyzers. This question provides insights into what aspects of static code analyzers need to be improved in order to attain a higher popularity.

The first question was whether or not people were aware of the existence of static code analysis for Java. If they were not, it ended the survey for them. The group that continued was divided into two subgroups; participants who are using or have used static code analyzers and participants who have never used static code analyzers. Both subgroups were presented different questions. The participants who used static code analyzers were asked three questions: (1) which tools they have used, (2) why they started using the tools, and (3) what problems they encountered when using these tools. The participants who never used static code analyzers were asked what their reasons were

for not using the tools.

To get a good population, various sources were used to find participants. There are several Java communities both on the Internet and in real life. Two of these communities were used to find subjects for the questionnaire. OWASP-NL [10] is a Dutch application security community. They held a meeting sponsored by Fortify[11], with a talk on secure programming with static analysis. This made this an audience with a relatively high knowledge on static code analysis, and around twenty people here participated in the survey.

The spring conference of the NL-JUG, the Dutch Java user group, was the second place the survey was held. This J-SPRING[12] is a yearly conference where eleven thousand Java specialists were present. Around thirty people filled in a questionnaire here. I used my personal Linked-IN network to find contacts that used Java, also through the NL-JUG group. This resulted in the rest of the participants filling in the questionnaire online.

3.2 Results

This survey was conducted from the 25th of March until the 30th of May 2008, both online and via paper forms. In the end there where 101 responses in the survey. Of course this is not a representative group for the Java community, because of the manner in which the survey was spread, but perhaps some conclusions can still be drawn from the results. These results can be found in table 3.1.

We will first look at the spread of static code analyzers through the Java community. How many actually use these tools and how often do they use them? And which tools do they use? Then we will look at the reasons users, who never used such tools, may have for not using these tools and finally we will look at the problems users encountered while using static code analyzers.

Usage One of the main reasons for not using static code analyzers, may be unfamiliarity with such tools. Therefor the first question the survey inquired about was the awareness of the existence of static code analyzers by the participant. Of the 101 participants, around 80 percent stated that they were at least aware of the existence of static code analyzers for Java.

The rest of the survey was only taken by the eighty people who were aware static source code analyzers exist for Java. Asked about their use of static code analyzers, revealed that nearly forty percent had never used such tools (though they were aware of their existence). Thirty percent had used a static code analyzer at some point, but did not continue using them on software projects. Another thirty percent do still use such tools, from which only six percent state to use them as often as possible. This is of course a shockingly small number of people, which immediately justifies to question the effectiveness of static source code analysis. We will come to the reasons for not using any static analysis shortly, but over thirty percent of this group has had reasons to

Table 3.1. Results of the survey. The actual result can be obtained in a .csv file from www.science.ru.nl/infstud/ fabianbr/basc/results.csv

Are you aware that static source code analyzers (SSCA's) exist for Java?	
Yes	80
No	21

Have you ever used a SSCA to examine some Java source code?	
No, Never	28
Yes, I have tried out a tool for a bit, but I don't regularly use one.	27
Yes, I use one or more SSCA's on occasion to improve source code	15
Yes, I use SSCA's on as many coding projects as I can.	8

What was your reason for never trying out a SSCA?	
I do not believe they will really find any bugs.	1
The bugs these tools find are not the ones I am looking for	6
It's too much of a hassle to install these tools and learn to use them.	13
There is never enough time in a software project to begin using these tools.	15
Other, There is not enough budget to begin using these tools.	2
Other, Never considered using such a tool.	3
Other	4

Which SSCA's have you used?	
Fortify SCA	12
ParaSoft JTest	12
Coverity Prevent	3
FindBugs	37
PMD	22
Eclipse Phoenix / Eclipse TPTP	9
CheckStyle	9
Jlint	9
Other, ESC/JAVA2	1
Other, JSR308 checkers	1
Other, GOT	1
Other, FxCop	1

Why did you start using a SSCA?	
I was required to use these tools by the company I worked for.	5
The use of such tools was encouraged by the company I worked for.	12
Out of personal interest.	32
Other, education	7
Other	2

What were / are the main problems you encountered while using SSCA's?	
Installation problems (eg frequent Eclipse crashes with SSCA plug-in).	4
Too steep learning curve in using these tools.	9
The tool falsely marks too much code as a bug.	7
After using a SSCA, there are still too many bugs left.	6
The use of a SSCA uses up a lot of time with bugs that are not my main concern.	11
The tool reports so many bugs that I do not know where to start.	15
The tool does not display the bugs ordered by severity.	21
The tool does not display the bugs categorized (eg. security or performance bugs).	9
It is hard or impossible to remove certain bug patterns from the list of bugs the tool checks for.	9
It is hard or impossible to extend the list of bugs the tool checks for with self created bug patterns.	10
On projects there is too little time to invest in learning / using SSCA's.	21
It is not company policy.	9
Other, performance issues.	3
Other	4

stop using these tools (figure 3.1).

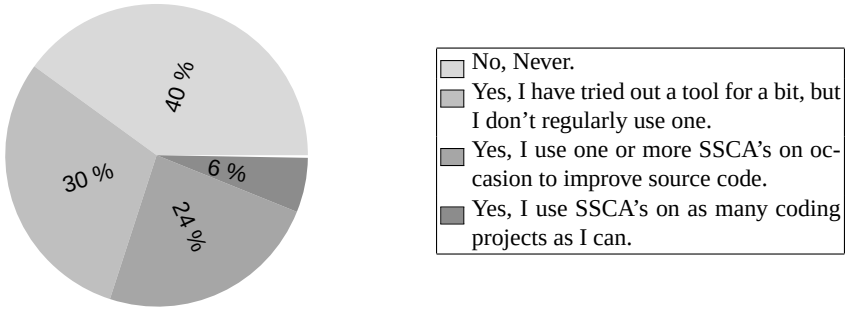

Fig. 3.1. Results of the question on the usage of static code analysis.

Those participants who have used static code analyzers were asked which ones they have used. The results can be seen in table 3.2. Please note that the column with percentages indicates the percentage of answers that contained a specific tool. Scince this was a question where more then one answer was possible, the total of this column exceeds a hundred percent.

Table 3.2. Results of the question "Which SSCA's have you used?" (multiple answers were possible)

Tool	Count	Perc. of users
FindBugs	37	71%
PMD	22	42%
Fortify SCA	12	23%
ParaSoft JTest	12	23%
Eclipse Phoenix / Eclipse TPTP	9	17%
JLint	9	17%
CheckStyle	9	17%
Coverity Prevent	3	8%
ESC/Java2	1	2%
JSR308 checkers	1	2%
GOT	1	2%

The first result here is that in this question most participants checked multiple tools. Nearly every participant who has used a static code analyzer at some time, has tried several ones. The popularity of FindBugs is another striking result here, nearly three quart of the participants who once used a static code analyzer used FindBugs. The best scor-

ing commercial tools are ParaSoft JTest and Fortify SCA. ESC/Java2, JSR308 checkers and GOT where not pre-defined choices in the survey, but filled in on blank spots, so they might have done better if they where pre-printed answers. However ESC/Java2 and JSR308 checkers are both tools that require extra user input, by requiring the programmer to annotate their code, and which were not a direct part of this research. Furthermore JSR308 tools have only existed for a couple of months now.

If we look further into the reasons the participants stated for starting to use static code analyzers we find that personal interest is the decisive factor for nearly sixty percent of the test group. Some twenty percent started to use static code analyzers on stimulation by the company they worked for and just under ten percent uses them because it is company policy to use them. There was also a group of around twelve percent who started using these tools for some form of education (figure 3.2).

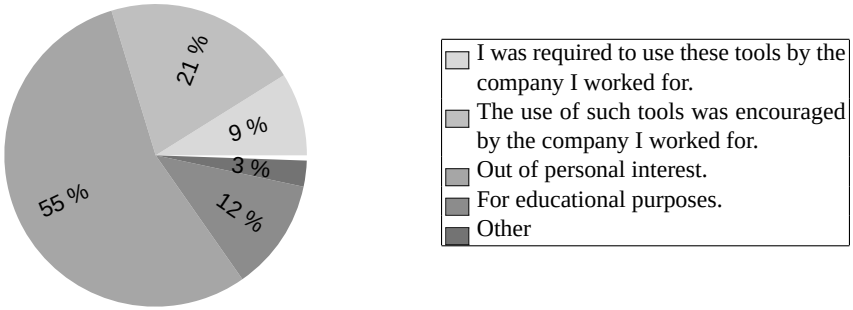

Fig. 3.2. Why did you start using static source code analysers?

Prejudice One third of the Java community stating they knew of the existence of static code analyzers has never used one. It is of course interesting to see why this group never wanted to use these tools. They might have good reasons for this, but perhaps they also have some misconceptions about static code analysis that leads to them never trying one of these tools.

This group was asked what their reasons were for never trying out a static code analyzer. The answers can be divided into three categories:

- *Correctness*, the assumptions that these tools will not find much (relevant) bugs; the first two reasons from figure 3.3.
- *Usability*, the assumption that learning to use these tools is to problematic; the third reason from figure 3.3.
- *Procedural*, lack of time or money for using these tools; the fourth and fifth reason from figure 3.3.

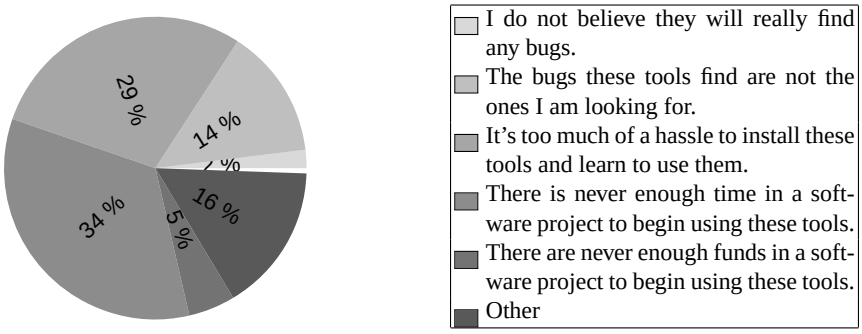

Fig. 3.3. Why did you never use a static code anlyzer.

The "other" option in this question spans all these categories and some answers there fall outside this categorization all together. It should be noted that lack of funds was not a pre-printed option with this question, but it was filled in by five percent of the participants non the less.

Only one person declared to think that these tools wouldn't find any bugs. So in general it would seem that people do believe these tools to work. The main reasons for never trying out a static code analyzer are not really correctness related. Nearly forty percent of the reasons state that there simply is not enough time or money in an organization to familiarize oneself with static code analysis. Also thirty percent thinks that it is too much trouble to learn how to use these tools. So even though these people do believe that static code analyzers would find bugs for them in their code (they did not check the correctness related reasons) they never used one. Over sixty percent of the reasons for not using a static code analyzer come down to somebody not being convinced that the cost in time, money and trouble actually weighs up to the improvement on their source code.

Experiences Now to look at the experiences by people who have actually used these tools. The problems that users experience can roughly be divided into five categories

- *Practical*, for instance problems during installation, or performance hits.
- *Correctness*, the actual effectiveness of static code analysis at finding bugs.
- *Usability*, the usability of static code analyzers.
- *Adaptability*, problems with adding new or removing existing bug-patterns in static code analyzers
- *Procedural*, all problems not directly related to these tools, but to their environment, such as project boundaries.

These five categories are chosen rather arbitrarily, especially the categories practical and adaptability are actually sub-domains of the usability category. However this

division leaves room for a more detailed view on the results, so we can better determine the big issues users encounter when using static code analyzers.

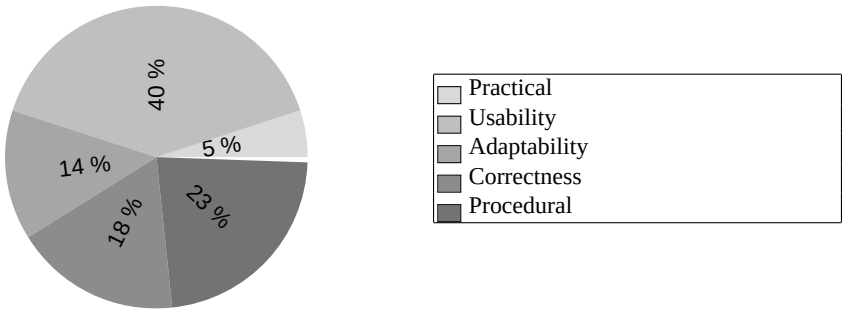

Fig. 3.4. The problems users encountered while using static code analyzers divided by category.

The results can be seen in table 3.3, where the last column shows the percentage of users who encountered these problems. Since participants could check multiple answers here, the total again exceeds a hundred percent. In figure 3.4 the relative percentages of the categories are shown.

The practical issues are problems such as unstable programs or performance issues, only five percent of the checked problems fall into this category. The correctness category represents the actual effectiveness of static code analysis at finding bugs. Almost twenty percent of the complaints fall into this category, mostly concentrating on the tools falsely marking too much code as a bug and the tools finding bugs that are not the users concern. The adaptability of these tools is a problem for about 15 percent of the experiences. These problems are evenly divided amongst the difficulty to add new and deactivate existing check rules. Forty percent of the responses fall in the usability category. Since the adaptability can easily be seen as a usability issue, usability scores as the most encountered problems with the use of these tools. The usability issues here, center specifically around the display of possible bug detections. Procedural problems are stated by over twenty percent of the participants. Among these is the shared number one problem which 21 percent of the participants encountered; the lack of time to invest in using or learning to use static code analyzers.

3.3 Discussion

As was stated earlier it is hard to draw conclusions from this data, because of the way the survey was spread through the Java community. Still now that the result have been presented, we might be able to arrive at some conclusions.

Table 3.3. The main problems users encountered while using static code analyzers.

Category	Problem	Count	Percentage
Practical	Installation problems (eg frequent Eclipse crashes with SSCA plug-in).	4	8%
Usability	Too steep learning curve in using these tools.	9	18%
Correctness	The tool falsely marks too much code as a bug.	7	14%
Correctness	After using a SSCA, there are still too many bugs left.	6	12%
Correctness	The use of a SSCA uses up a lot of time with bugs that are not my main concern.	11	22%
Usability	The tool reports so many bugs that I do not know where to start.	15	30%
Usability	The tool does not display the bugs ordered by severity.	21	42%
Usability	The tool does not display the bugs categorized (eg. security or performance bugs).	9	18%
Adaptability	It is hard or impossible to remove certain bug patterns from the list of bugs the tool checks for.	9	18%
Adaptability	It is hard or impossible to extend the list of bugs the tool checks for with self created bug patterns.	10	20%
Procedural	On projects there is too little time to invest in learning / using SSCA's.	21	42%
Procedural	It is not company policy.	9	18%
Practical	Other, performance issues.	3	6%
Other	Other	4	8%

Usage If the twenty percent who have never heard of static code analyzers is a large group, is hard to say. As mentioned before, because of the way the survey was spread, this number can not be seen as representative of the entire Java community. A part of the questionnaire was taken at a meeting of the OWASP-NL[10], a Dutch application security community, where one might expect more people to know about static analysis. This expectation indeed shows in the results. Figure 3.5 shows only the results of the question on awareness of static code analysis taken at the OWASP meeting. When we exclude the results from the OWASP meeting from the total results we get the percentages shown in figure 3.6.

So the percentage of programmers unaware of static code analyzers becomes nearly a fourth, when we exclude results obtained from a meeting with security minded persons. This indicates that the actual number of people unaware of static code analysis is probably larger then the twenty percent concluded from the entire test group. More research will be necessary to find out whether this is an accurate estimate. However if around twenty percent of the Java community is indeed not aware that automated tools exist to improve ones code, then some profit can still be gained by informing this group.

When we look at which tools the participants have used, there is no real difference between the participants from the OWASP meeting and the rest of the test group. Because it was an OWASP meeting sponsored by Fortify, this could have had an influence on the results, but the scores here were largely the same as from the overall test group.

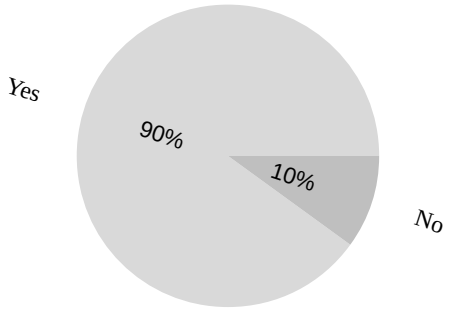

Fig. 3.5. Results enquiry on awareness of SCAs at OWASP-NL.

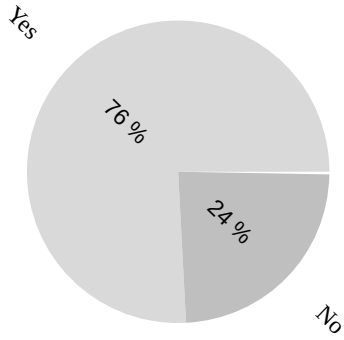

Fig. 3.6. Results enquiry on awareness of SCAs, with OWASP-NL results excluded.

If we only look at the answers from the participants who stated they still use static code analyzers (as seen in table 3.4), FindBugs is still the most used tool, but the difference with PMD has shrunken. The relatively low rankings of the commercial tools here is prominent, because you would expect them to score much better in this category.

The answers from the participants who stated that they no longer use static code analyzers are summarized in table 3.5. These are the programmers who decided to try out a static source code analyzer, but for some reason stopped using them. Again FindBugs is firmly on top. Apparently about half of the participants who have ever used FindBugs, for some reason stopped using it on other projects. This does not really say anything on the quality of FindBugs seeing that over seventy percent of the participants checked FindBugs as a tool they used, it is no wonder it also scores highly in subsets of these answers. There does not seem to be a strong relation between the choice of tools and

Table 3.4. Top results from programmers who still use SCAs

Tool	Count
FindBugs	20
PMD	14
CheckStyle	8
Eclipse Phoenix / Eclipse TPTP	7
Fortify SCA	7

Table 3.5. Top results from programmers who stopped using SCAs

Tool	Count
FindBugs	17
PMD	8
ParaSoft JTest	7
JLint	6
Fortify SCA	5

the decision to stop using them. Furthermore if we look at the reasons these participants stated for using these tools, we find that nearly seventy percent of them started using these tools either for educational purposes or out of personal interest, both of which are reasons that can lead to short term use of static code analysis.

It is striking that nearly all users of CheckStyle, are frequent users of static code analysis. Perhaps the ease of use and up-front clarity of this tool attributes to this. Fur-

thermore this is a tool that can be used to enforce a companies coding guidelines, which might also explain its popularity with frequent users of static code analysis.

When looking at the reasons for starting to use static code analysis it is clear the main reason is personal interest. About a third starts using them because they are in some way stimulated to do so by the company they work for. Assuming the lion's share of the participants work for a company they have to write Java code for, thirty percent is a low number.

Twelve percent stated they came to use static code analysis for educational purposes. This also seems a very low number because static code analysis can be a valuable tool in programming education [2].

Prejudice In this section we will concentrate on the participants who never used static code analyzers. Looking more closely at the reasons participants had to never start using static code analyzers, we see that around ten percent opted that these tools do not find the kind of bugs they are looking for. Please recall that there are basically two ways of using a static code analyzer; during the development process, for instance checking for bug patterns whenever a piece of code is compiled, and after the development process as an aid in testing, bug fixing or auditing. If a user complains he is not finding the bugs he is looking for, then he will most likely be using a static code analyzer at the very end of the development phase. After all, during development you will probably be glad with every bug these tools prevent you from creating.

It is of course true that static code analyzers are designed to find generic flaws, and though some can be extended with additional bug patterns to look for, for specific application logic this might not be very efficient. However if you would use these tools during the implementation of the source code, then this would prevent some generic flaws and (when extended) possibly enforce certain company coding standards, which would almost certainly lead to more reliable and better maintainable code [5]. This in turn could decrease the troubles at the end of the production process. So this reason for not using static code analysis is questionable when used by any developers.

As we found in the results, nearly seventy percent of the reasons for not starting to use static code analyzers comes down to a cost-benefit estimation that does not favor static code analyzers. Costs in these estimates can be monetary, but also time and effort. When used during development (some tools even have the option to add their check to the ANT-build tasks) the performance hit for most tools is very acceptable [3]. The acquisition costs of these tools should not be a problem, considering the many free alternatives. From own experience the training costs for developers to learn how to use such tools does not seem very high. Most of these tools have plug-ins for Eclipse, which should benefit their usability. Although anybody who has ever installed too many eclipse plug-ins knows the performance issues this can bring along with them, it seems premature to reject them on low usability without ever using them. It might be that most people expect these tools to be too complicated in their use. So it could be a good idea for creators of static code analyzers to advertise their products ease of use.

Experiences Now we will look further at the actual problems some participants have experienced while using static code analyzers, starting with the practical ones. As can be found in table 3.3, this category represents problems like unstable tools and impact on performance. Only a small percentage state these experiences, mostly without a clear link with the exact tools they have used.

When we look at the correctness category, the ability of static code analyzers to actually find bugs, we see that around eighteen percent of all problems fall into this category (figure 3.4). The first problem in this category is the false positives. Most tools are being designed to minimize this number of false positives, simply because of its annoyance. Conversely this also causes them to have a higher false negative rate, which is the second problem in this category. However nearly all participants either checked both these problems, or none of them. So around fourteen percent of all participants still find the current rates of false positives and false negatives troublesome.

The third problem in the correctness category, is interesting; the use of static code analyzers takes up a lot of time with bugs that are not my main concern. So these participants claim that static code analyzers do find bugs in their code, but they cannot be bothered with these types of bugs. It is hard to state who is correct here. Sure the bug indicated by a static code analyzer might not be the problem you are looking for, but it may very well be the problem you are looking for next week. On the other hand if a developer spends a lot of time fixing bugs found by the static code analyzer, which happen to be less important then the bugs it doesn't find, this might ruin the time a developer can spend on the important bugs. Again this problem seems less valid when using a static code analyzer during the implementation phase. When using a tool in this way, it should use up relatively little time, while preventing some bugs that do not need fixing afterwards.

Almost fifteen percent of the participants found the adaptability of the static code analyzer lacking. Either the addition of new bug patterns, or the deletion of an existing bug pattern, was too difficult or could not be done. The ability to deactivate a certain bug check, that time and again has falsely marked a code pattern as a bug seems imperative for static code analyzers to enjoy user acceptance. But also the addition of new rules (e.g. prohibiting the use of certain libraries) can make these tools much more valuable for a company. Most current versions of static code analyzers have these features, and it is unclear from the responses which specific tools lack them.

The usability issues center around the display of possible bugs. Even with this narrow definition, it is by far the biggest category of problems and in fact the lack of ordering by severity of the possible bugs is the shared number one problem the participants experience with static code analyzers. When these tools are used on a large amount of code for the first time, a lot of possible bugs will probably be detected. Problems arise as to which bugs to concentrate on, when there are thousands of possible bugs. Most static code analyzers do have some ordering on severity of these bugs, but this ordering is rather ad-hoc [7].

This is a very difficult problem to solve, since static code analyzers have no understanding of the programs it scans. It can for instance report that invalidated input is queried directly to the database somewhere, but is this more severe then a String comparison using '==' somewhere else in the code? A static code analyzer does not know whether the program will be used as an high end, Internet connected database containing very sensitive information, or as a simple personal recipe database on a stand alone computer. In the first case the lack of input validation is probably worse, but the String comparison might make essential functionality like logging in, impossible, or it might not be a bug at all.

The problem only increases when a static code analyzer encounters the same bug patterns in multiple class files. Which of those is most important to fix? First flagging bug patterns in the class files where the most execution paths go through, might be an option, but such an analysis is not scalable when the number of class files increase.

Better solutions to this problem seem to either give a static code analyzer more understanding of the program it is checking, or by maintaining a history on which bugs were solved in the past. In order to increase the understanding a static code analyzer has on the code it checks, we have to give it more information about the code. In the case of Java this can be done via annotations. This technique is used in ESC/Java2 [13] and will be integrated into FindBugs [6]. The downside here is that it demands more input from the user and thus becomes more work intensive. A history-based warning priorization has already been proposed [7,8], and seems to improve the precision of static code analysis. This approach means that the source code revisions are reviewed by scanning through the code in a versioning system and measuring the average time a certain bug category remains in the code. The faster a certain bug category is fixed the less time it will remain in the code and the more important these types of bugs will be for the developers. However this approach will need a change history of a software project in order to be succesfull.

Analogously it is also hard to map certain bug patterns in a bug category (eg security or performance). For some, like using 'new Integer()', instead of 'Integer.valueOf()', this is a clear call - the 'Integer.valueOf()' statement is much more efficient, both computational and memory wise - but for others like the String comparison this is impossible.

This is a problem that will probably not be addressed by static code analyzers in the near future. Again, using static code analyzers frequently during development is the best remedy, because this will keep the number of reported bugs maintainable.

Finally the procedural category, which besides coming in second in the overall problems list, also contains the shared number one problem with using static code analyzers; lack of time on software projects. This is quite similar to the problems with lack of time or funds to even begin with using static analysis tools from the previous section. It should be researched whether the use of these tools outweighs the costs.

4 A case study

As we have seen in the previous section one of the reasons for not using a static code analyzer can be that these tools find different bugs than the ones developers care about. It is of course possible for a static code analyzer to identify erroneous code within source code that does not lead to erroneous behavior of the program. This is not exactly the same as a false positive, since the bug pattern was found correctly and the code is indeed faulty, but perhaps only under circumstances that will never occur. This section describes a practical experiment to see how effective these tools are at finding the actual bugs that developers care about. A static code analyzer will be run against several releases of a software project that has been developed without the use of static analysis. This project has had several bug fixes. If we compare the reports the static code analyzer creates then we can see how many of the reported bugs are resolved over time and thus how effective the static code analyzer is at finding the bugs developers care about.

There has been more work in practical tests like this lately. In [14] Ayewah et al look at FindBugs reports that are automatically generated from FindBugs runs over any modified code in the Google code base. Using a hashing technique they follow every bug through all the commits and can thus see how all the reported bugs are handled. They conclude that FindBugs indeed finds a lot of what they call trivial bugs, which are defective code patterns that do not cause faulty behavior by the program. This research however was done on a code base that was created while using static analyzers.

Sunghun and Ernst do not expect the production software they examine to have used static code analysis during creation [7,8]. However, it might still be the case that this production software did use static code analysis. They want to find a historical prioritization for bug reports by looking through the revision history of two software projects and measuring how long specific code defects remain in the code. They therefore assume that more critical problems are fixed quickly. A similar assumption is made in this research, where it is assumed that bugs which are important to developers are removed from the code.

Wagner et al [15] evaluated FindBugs and PMD on two software projects, in a research that resembles this research the most. Their aim is to find how static analysis tools can best be deployed in the quality assurance process for software. They find that very few of the reported bugs by these tools actually correlate with documented bugs in the deployed software.

This research will look at a single project where no static analysis was used during implementation. By comparing the bug reports of several versions from this project we can see how many code defects identified by the static analysis tool were fixed during future releases and thereby judge the effectiveness of a static analysis tool at finding “real world” bugs.

4.1 Setup

As a static analysis tool, FindBugs was used for this research. It is an open-source static analysis tool for Java that is widely used as we could see in the previous section.

FindBugs examines the Java byte code for known bug patterns and performs data flow analysis [16]. Version 1.3.4 of FindBugs, the most current release at the time, was used in this test.

The tested project is a commercial product developed in the Java Enterprise Environment (J2EE). For confidentiality reasons this project will be referred to as project X. It is a typical web-based application with a database back-end, consisting of around 130.000 lines of code (LOC) divided over 500 classes in 15 packages. The project was developed from 2005 until recent, over several releases without ever using a static analysis tool during development.

Six tagged versions of project X were retrieved from the repository. Each version was about three months apart from the previous version. Because FindBugs works on class files, these projects had to be build first. Then FindBugs reports were generated on all of them. These XML reports were then compared with each other.

To check whether the various software versions actually contained the same reported bugs, I manually compared all high priority bugs and a random sample of medium priority bugs. This random sample was around two percent.

For every reported bug that remains in the project it is possible to argue that they were not fixed simply because developers were not informed of them, not because they were not interesting for developers. Therefor a developer on project X was asked to classify each reported bug of the newest version of project X into one of the following categories:

- *False positive*, the reported bug is not a bug at all.
- *Nice to fix*, the reported bug signifies a code defect that will not result in faulty program behavior. However given enough time these can still be interesting to fix, for instance because they might prevent future bugs or because fixing them improves the maintainability of the code.
- *Important fix*, the reported bug will lead to faulty program behavior and should be fixed.
- *Needs research*, the reported bug needs more research in order to classify it in any of these categories.

In this way we can measure whether if the reported bugs are indeed bugs that developers care about.

Every single bug that FindBugs finds will be called a reported bug. When it is verified that a reported bug is an actual code defect (so not a false positive) we will call it a source code bug. The bugs that are actually encountered inside programs will be called actual bugs. Whenever a reported bug dissapears in a newer release, we assume that this reported bug was fixed and thus that this was an actual bug. It is assumed that during the use of the program X, several actual bugs were encountered and fixed in subsequent releases. The question therefore is whether the use of FindBugs would have prevented some of these actual bugs by accurately reporting source code bugs.

Note that in this research we only look at all reported bugs, and see how many of those are actual bugs. Thus every actual bug is a source code bug and a reported bug. This research states nothing about actual bugs that are not reported bugs. Also there can be more reasons than a bug fix for the dissappearance of a reported bug, such as the removal of an entire piece of code. However, for all the high priority reported bugs

and the sample of medium reported bugs this was not the case and the removal of said reoported bugs was an actual bugfix.

4.2 Results

Table 4.6 shows the results FindBugs found in the six versions of project X that were checked out of the version control system. This was the first time static analysis was applied to the code in this project.

Table 4.6. FindBugs reports from six different versions of project X

Version tag:	**v1.0.0**	**v1.1.0**	**v1.1.7**	**v1.2.0**	**v2.0.0**	**v2.0.7**
Total Lines of Code:	124724	128015	128401	128624	130505	130788
Number of Classes:	506	517	519	519	526	526
Total number of bugs	**1030**	**1045**	**1040**	**1035**	**1067**	**1061**
High priority bugs	*53*	*56*	*53*	*50*	*60*	*57*
Bad practice	11	11	9	8	10	9
Correctness	3	4	4	2	7	5
Malicious code vulnerability	14	14	14	14	15	16
Dodgy	25	27	26	26	28	27
Medium priority bugs	*977*	*989*	*987*	*985*	*1007*	*1004*
Bad practice	229	230	231	231	235	235
Correctness	16	19	17	17	19	18
Malicious code vulnerability	62	62	62	63	66	66
Performance	557	567	566	566	574	573
Dodgy	113	111	111	108	113	112

The bug categories that are seen in table 4.6 are defined by FindBugs. These should be interpreted as follows:

- *Bad practice* are issues that involve clear violations of recommended and standard coding practice. For example, failing to close file or database resources.
- *Correctness*; issues involving code that is probably incorrect in some way, for instance dereferencing a sure null pointer. FindBugs strives at a very low false positive rate here.
- *Malicious code vulnerability* is a possible security vulnerability for malicious code, such as not making fields final when they could be.
- *Performance* contains potential performance issues, such as unnecessary object creation.
- *Dodgy*; code that seems odd, such as switch fall throughs. FindBugs allows more false positives here.

When we look at the number of reported bugs per thousand lines of code we see that this number remains at around eigth for all releases. Release v1.0.0 has the most bugs per thousand lines of code and version 1.2.0 the least.

Assuming that most truly important actual bugs were resolved after some versions of this project, it seems that most reported bugs generated by FindBugs are indeed not the kind of bugs developers are looking for. So although these results say little about the percentage of source code bugs in the reported bugs, the percentage of actual bugs in the reported bugs seems very low.

We do see a decline in the number of reported bugs between some versions. After checking all the change logs and release files it seemed that versions 1.1.0 and 2.0.0 introduced a lot of new functionality into the system, while versions 1.1.7, 1.2.0 and 2.0.7 mainly contained bug fixes when compared to the previous releases. This also shows when we look at the increase of the total lines of code and the total number of class files. In fact in this data there is a link between the increase in class files and the increase in reported bugs; the more class files are added to the project the more bugs are reported by FindBugs. If few or no new classes are added then the number of reported bugs drops.

Of course the number of reported bugs that are not resolved is striking. Over a thousand reported bugs are found throughout the releases. Of course there is no guarantee that these bugs are continuously the same. All the high priority but only two percent of the medium priority bugs were tracked through the source code. It is possible that some of the bugs in a category were resolved and new ones were inserted between versions. however, it seems unlikely that this happened for a large percentage of the reported bugs since the number of reported bugs per category between versions are often so close together.

Some of the bugs reported by FindBugs were indeed actual bugs. Table 4.6 shows the results of the FindBugs test per priority level and bug category. About three fourth of the number of bugs that are removed from the system are located in the High priority category.

If we assume that the decline in reported bugs is caused by the fixing of actual bugs, then it also shows that these types of bugs are specifically removed from certain categories. In categories like bad practice and correctness we can sometimes see drops in the number of reported bugs, while none of the reported bugs in categories like malicious code vulnerability ever get fixed. Looking more closely at the high priority reported bugs gives us the results shown in table 4.7.

Source code bugs involving dropped or ignored exceptions often lead to actual bugs that developers have to fix. The same can be said for null pointer dereference errors.

The category of J2EE errors consists of four accounts of the storing of non serializable Objects into a HttpSession. This means that if the current session has to be stored to disk then these non serializable objects will be lost. These however caused no actual bugs and thus were never resolved.

The mutable static field category complains on the fact that several fields in the code are not made ‘final’ even though they should be. This is a security issue and does not lead to wrong program behavior, but is does make a program vulnerable for attacks. These issues were never dealt with and actually their number only increased over time.

The dead local store issues are also not fixed very often, but these source code bugs can not really lead to actual bugs, they only obfuscate the code.

Table 4.7. Detailed high priority reported bugs.

Version tag:	**v1.0.0**	**v1.1.0**	**v1.1.7**	**v1.2.0**	**v2.00**	**v2.0.7**
Bad practice	*11*	*11*	*9*	*8*	*10*	*9*
Checking String equality using == or !=	1	1	1	0	1	1
Confusing method name	4	4	4	4	4	4
Dropped or ignored exception	2	2	0	0	1	0
J2EE error	4	4	4	4	4	4
Correctness	*3*	*4*	*4*	*2*	*7*	*5*
Masked field	0	0	0	0	1	1
Null Pointer dereference	1	3	2	0	5	2
Redundant comparison to null	2	1	2	2	1	2
Malicious code vulnerability	*14*	*14*	*14*	*14*	*15*	*16*
Mutable static field	14	14	14	14	15	16
Dodgy	*25*	*27*	*26*	*26*	*28*	*27*
Dead local store	24	26	25	25	27	26
Misuse of static fields	1	1	1	1	1	1

The case study until now assumed that if a source code bug causes a actual bug that it will then at some point be removed. If the number of reported bugs drops between releases, then this shows that the use of FindBugs on this project would have prevented those actual bugs. However the rate of removed reported bugs is very low in the results of tables 4.6 and 4.7. Of course it is still possible for these reported bugs to accurately identify source code and actual bugs that simply have not been found yet. In order to test this a developer of project X was asked to review all reported bugs generated on version 2.0.7 of project X. The results of this analysis can be found in table 4.8.

The indicator of ‘False positive’ of course shows a false detection. ‘Nice to fix’ shows a source code bug that should be resolved, for instance because it improves performance or the readability of the code, but that does not cause actual bugs. Possible actual bugs are indicated as ‘Important fix’ and the ‘Needs research’ category are the ones that require further study to find out the true extent of the problems.

It took a developer around two hours to sift through the reported bugs. It shows here that FindBugs does perform a lot better then the first test results might have led to believe. A little over a fourth of the reported bugs are false positives. There are still twelve potential actual bugs in the code, that were deemed important enough to be resolved immediately, and nearly three quart of the reported bugs indeed report source code bugs, but they are not of immediate threat to the program.

4.3 Discussion

These results are only from one software project made by one company. Naturally we will need more scans on different projects to reliably conclude anything about the usefulness of FindBugs, let alone static code analyzers in general. That being said, the results obtained here can still lead to some conclusions. Actually the results mostly coincide with results from [8] where the authors also found that the null pointer dereference bugs were the FindBugs reports that got resolved the fastest.

Table 4.8. Classification of bugreports on version 2.0.7 of project X by a developer.

Bug pattern	False positive	Nice to fix	Important fix	Needs research
Percentage of total bugs	29%	68%	1%	2%
Total	309	719	12	21
High priority bugs	*9*	*43*	*4*	*1*
Bad practice	8	0	1	0
Correctness	0	2	3	0
Malicious code vulnerability	1	15	0	0
Dodgy	0	26	0	1
Medium priority bugs	*300*	*676*	*8*	*20*
Bad practice	228	4	2	1
Correctness	0	3	5	10
Malicious code vulnerability	55	11	0	0
Performance	2	569	0	2
Dodgy	15	89	1	7

It seems rather logical that null pointer related bugs and mishandled exception bugs are resolved the fastest in this case study. These types of bugs tend to cause stack traces, either on your screen or in your log files, which are signs that are easily picked up during tests and reported as actual bugs to the developer.

It is also interesting to notice that the ‘redundant comparison to null’ report always increases when the reports of null pointer deference decreases. It seems likely that a developer looking to fix a null pointer dereference became over cautious and added unneeded null checks. Every case of an extra ‘redundant comparison to null’ in the high priority bugs has been manually verified to correspond with this intuition.

The fact that none of the malicious code vulnerability bugs get resolved can be explained because project X is a standard web-application. For web-applications the mutable static fields will not lead to actual bugs.

The priorization of the reported bugs by FindBugs seemed to work quite well. At least the number of false positives is significantly lower in the high priority category.

A piece of code can contain a source code bug, recognized with the code patterns of a static code analyzer. Even though it is indeed a code defect, this does not have to cause faulty behavior of the program; an actual bug. An excellent example of this is so called “dead code”; code that is unreachable or unused, e.g. a variable is instantiated, but never used. Static code analyzers are typically quite adept at finding these bugs, but developers will probably not deem them very important, since they do not cause faulty behavior of the program. Should a static code analyzer complain about these bugs then? That is a difficult question. On the one hand these reports can be very annoying for a developer who is not interested in finding these bugs. On the other these types of bugs can have an impact on the performance of the program (instantiating variables you do not need are wasted resources), but more importantly they can confront a developer with a more serious mistake if he assumed the variable would be used elsewhere. Another serious benefit of preventing these bugs is the increased readability and thus maintainability of the code. In the end it is up to the user who can tune the static code analyzer to suit his

needs.

Table 4.8 shows that the lion's share of reported bugs would be fixed if the developers were given enough time. This seems another argument for using a static code analyzer during development, since it often takes much less time to edit the code just after writing it and because these bugs were designated as 'Nice to fix' developers would probably fix most of them right away. Also around 530 of these reported bugs were from the likes of "it is more efficient to use Integer.valueOf(), then new Integer()" these kind of advices will probably be adopted automaticaly by developers once they are confronted with such reports a couple of times.

As an added note, it took the developer that estimated the reported bugs severity around thirty minutes to tweak FindBugs in such a way that it ignored 212 of the 309 false positives. This means that with a little effort the performance of FindBugs can greatly improve.

5 Conclusions

The objective of this thesis was to investigate the use of static code analyzers among Java developers and whether bugs found by static code analyzers are considered by developers to be important. A survey was performed among Java developers to investigate the use of static code analysis. Although the amount of participants was not a large enough sample of the entire Java community, some interesting conclusions can be drawn. Furthermore, a case study was performed to find out whether the static code analyzer FindBugs found bugs that were considered important.

The survey among Java developers showed that twenty percent was not aware of the existence of static code analyzers. One cause seems of this is that programming courses do not mention static code analysis. Static code analysis could be taught and used in programming courses because it will point out a lot of often made bugs to beginning developers.

In addition, the survey showed that static code analysis is only used extensively by six percent of the participants that knew static code analyzers existed. This number shows that static code analysis faces a serious popularity problem. Namely, the participants that knew but never used static code analyzers indicated that the costs in time, money, and effort do not weigh up to the improvement of source code. Moreover, the participants who used but stopped using static code analyzers reported that the main problem is the lack of funds or time to incorporate it into the project. Developers of static code analyzers should work on the ease of use and the promotion of the ease of use of their tools.

The participants who were using static code analyzers complained that these tools report an enormous amount of both important and irrelevant bugs in an unordered way. Research is currently being done to prioritize reported bugs better, allowing the developer to focus on the important bugs. In the meantime, the best remedy is to perform static code analysis frequently from the start resulting in a manageable number of re-

ported bugs. However, for the success of static code analyzers it is essential that it finds important bugs and presents them in an ordered way.

As confirmed by the survey, users experience that the quality of reported bugs should be improved. Our case study investigated the quality of the reported bugs by FindBugs in one commercial software project. Thirty percent of the bugs reported by FindBugs were considered to be false reports. However, FindBugs found 12 bugs that were considered as critical and should be resolved right away. Also, 68 percent of the bugs were classified as worthy to fix, meaning that the bug is not critical, but could become critical or would lead to better maintainable code. From these numbers, follows the conclusion that in this case study, FindBugs shows to be an effective tool to improve source code and to enforce certain coding standards.

6 Future work

There are several general recommendations I have for research on static code analysis. Namely, find more important bugs, and focus on the ease of using static code analyzers in software projects. However, to fulfil these recommendations, the opinions of developers on these subjects should be investigated. Namely, which bugs would developers like to be found? What bugs do developers consider insignificant? And, what aspect do developers think would increase user friendliness? In summary, a recommendation to investigate the needs of the developer.

Also follow a Java project over time, that does not use static code analysis, and measure the number of bug fixes that would have been avoided if static code analysis had indeed been used. Then you would have the actual percentage of bug fixes that could have been avoided using a static code analyzer. Together with a transcript of the hours that were spend fixing these bugs, a measurable economic argument for using static code analysis might follow.

References

1. Plsch, R., Gruber, H., Pomberger, G., Saft, M., Schiffer, S.: Tool support for expert-centred code assessments. In: Proceedings of the IEEE International Conference on Software Testing, Verification, and Validation (ICST 2008). Volume 9-11., Lillehammer, Norwegen, IEEE Computer Society Press (April 2008)
2. Truong, N., Roe, P., Bancroft, P.: Static analysis of students' java programs. In: ACE '04: Proceedings of the sixth conference on Australasian computing education, Darlinghurst, Australia, Australia, Australian Computer Society, Inc. (2004) 317–325
3. Rutar, N., Almazan, C.B., Foster, J.S.: A comparison of bug finding tools for java. In: ISSRE '04: Proceedings of the 15th International Symposium on Software Reliability Engineering, Washington, DC, USA, IEEE Computer Society (2004) 245–256
4. Huang, Y.W., Yu, F., Hang, C., Tsai, C.H., Lee, D.T., Kuo, S.Y.: Securing web application code by static analysis and runtime protection. In: WWW '04: Proceedings of the 13th international conference on World Wide Web, New York, NY, USA, ACM (2004) 40–52
5. Foster, J.S., Hicks, M.W., Pugh, W.: Improving software quality with static analysis. In: PASTE '07: Proceedings of the 7th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering, New York, NY, USA, ACM (2007) 83–84
6. FindBugs: `http://findbugs.sourceforge.net/`
7. Kim, S., Ernst, M.D.: 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, New York, NY, USA, ACM (2007) 45–54
8. Kim, S., Ernst, M.D.: Prioritizing warning categories by analyzing software history. msr **0** (2007) 27
9. Louridas, P.: Static code analysis. In: IEEE Software. Volume 23. (Jul/Aug 2006) 58–61
10. OWASP-NL: `http://www.owasp.org/index.php/Netherlands`
11. Fortify: `http://www.fortify.com/`
12. NLJUG, J-Spring 2008: `http://www.nljug.org/pages/events/content/jspring_2008/`
13. ESC/Java2: `http://kind.ucd.ie/products/opensource/ESCJava2/`
14. Ayewah, N., Pugh, W., Morgenthaler, J.D., Penix, J., Zhou, Y.: Using findbugs on production software. In: OOPSLA '07: Companion to the 22nd ACM SIGPLAN conference on Object oriented programming systems and applications companion, New York, NY, USA, ACM (2007) 805–806
15. Wagner, S., Deissenboeck, F., Wimmer, M.A.J., Schwalb, M.: An evaluation of bug pattern tools for java. unpublished (January 2007)
16. Hovemeyer, D., Pugh, W.: Finding bugs is easy. SIGPLAN Not. **39**(12) (2004) 92–106

A Questionnaire

Spring 2008 Static Source Code Analyzers for Java Radboud University of Nijmegen

Background This survey is a part of a bachelor thesis. The goal of this survey is to get a better view on the use and problems of static source code analyzers (SSCA's) for Java. Static source code analysis tools, also called "*static analysis tool*", "*source code scanners*", or "*source code analyzers*", are tools that can identify software failures (bugs) in source code, without actually compiling or executing the code. If you would like to receive the outcome of this survey please fill out your email address here:_____
Please note your e-mail address is not mandatory. You will receive only one e-mail with the results of this survey, after which the record of your e-mail address will be deleted. Your e-mail address will not be given to another party.

1. Are you aware that static source code analyzers (SSCA's) exist for Java?

- ☐ Yes.
- ☐ No, *thank you very much for taking this survey.*

2. Have you ever used a SSCA to examine some Java source code?

- ☐ No, never.
- ☐ Yes, I have tried out a tool for a bit, but I don't regularly use one. *please continue at question 4*
- ☐ Yes, I use one or more SSCA's on occasion to improve source code. *please continue at question 4*
- ☐ Yes, I use SSCA's on as many coding projects as I can. *please continue at question 4*

3. What was your reason for never trying out a SSCA? (*multiple answers possible*)

- ☐ I do not believe they will really find any bugs.
- ☐ The bugs these tools find are not the ones I am looking for.
- ☐ It's too much of a hassle to install these tools and learn to use them.
- ☐ There is never enough time in a software project to begin using these tools.
- ☐ _____

Thank you very much for taking this survey.

4. Which SSCA's have you used? (*multiple answers possible*)

- ☐ Fortify
- ☐ ParaSoft JTest
- ☐ Coverity Prevent
- ☐ FindBugs
- ☐ PMD
- ☐ Eclipse Phoenix / Eclipse TPTP
- ☐ Checkstyle
- ☐ JLint
- ☐ _____
- ☐ _____
- ☐ _____

5. Why did you start using a SSCA? (*multiple answers possible*)

- ☐ I was required to use these tools by the company I worked for.
- ☐ The use of such tools was encouraged by the company I worked for.
- ☐ Out of personal interest.
- ☐ _____

6. What were / are the main problems you encountered while using SSCA's? (*multiple answers possible*)

- ☐ Installation problems (eg frequent Eclipse crashes with SSCA plug-in).
- ☐ To steep learning curve in using these tools.
- ☐ The tool falsely marks too much code as a bug.
- ☐ After using a SSCA, there are still too many bugs left.
- ☐ The use of a SSCA uses up a lot of time with bugs that are not my main concern.
- ☐ The tool reports so many bugs that I do not know were to start.
- ☐ The tool does not display the bugs ordered by severity.
- ☐ The tool does not display the bugs categorized (eg. security or performance bugs).
- ☐ It is hard or impossible to remove certain bug patterns from the list of bugs the tool checks for.
- ☐ It is hard or impossible to extend the list of bugs the tool checks for with self created bug patterns.
- ☐ On projects there is too little time to invest in learning / using SSCA's.
- ☐ It is not company policy.
- ☐ _____
- ☐ _____

7. Do you have anything to add on the use of SSCA's?

Thank you very much for filling out this questionnaire.