Why Don't Software Developers Use Static Analysis Tools to Find Bugs?

Brittany Johnson, Yoonki Song, and Emerson Murphy-Hill
North Carolina State University
Raleigh, NC, U.S.A.
bijohnso,ysong2@ncsu.edu,emerson@csc.ncsu.edu

Robert Bowdidge
Google
Mountain View, CA, U.S.A.
bowdidge@google.com

***Abstract*—Using static analysis tools for automating code inspections can be beneficial for software engineers. Such tools can make finding bugs, or software defects, faster and cheaper than manual inspections. Despite the benefits of using static analysis tools to find bugs, research suggests that these tools are underused. In this paper, we investigate why developers are not widely using static analysis tools and how current tools could potentially be improved. We conducted interviews with 20 developers and found that although all of our participants felt that use is beneficial, false positives and the way in which the warnings are presented, among other things, are barriers to use. We discuss several implications of these results, such as the need for an interactive mechanism to help developers fix defects.**

I. Introduction

Software quality is becoming more important with the increasing reliance on software systems. There are different ways to ensure quality in software, including code reviews and rigorous testing. Software defects, or bugs, can cost companies significant amounts of money, especially when they lead to software failure [1], [2].

Static analysis tools provide a means for analyzing code without having to run the code, helping ensure higher quality software throughout the development process. There are a variety of ways to perform automatic static analyses [3], including at the developers request, continuously while creating the software in a development environment, and just before the software is committed to a version control system. The tool may allow the developer to configure what kinds of bugs it finds, and sometimes even define new bug patterns. Some automated static analysis software, such as the software integrated into IntelliJ IDEA [4], provide *quick fixes*. A quick fix is a suggested solution for a defect that is automatically applied to a developer's code. To help explain the "state of the art" of static analysis tools, let us look at FindBugs [5] as a concrete example of how these tools work [6]. FindBugs runs as a plug-in for the Eclipse [7] and NetBeans [8] integrated development environments (IDEs). It can also be run from the command line or as a separate tool on its own. When run in the IDE, FindBugs has its own perspective where the defects are listed and organized. Each defect is assigned a severity, signifying how important the defect is; either *high*, *medium* or *low*, each represented by red, yellow and green bug markers respectively. FindBugs offers a select few quick fixes.

There are many situations where a developer may consider using a static analysis tool to find defects in their code. Let us consider a developer, Susie. Susie is a software developer at a small company. She wants to make sure that she is following the company's standards while maintaining quality code. She needs a way of checking her code in her IDE, before submitting it to the general code repository, without worrying about any outside dependencies that she has no control over. Susie decides that her best bet is to install a static analysis tool. She decides to install FindBugs because she likes the quality of the results and the fact that bugs can be found as she types; at first, she is very happy with her decision and feels productive when using it.

The above scenario is an interpretation of an experience one of our participants recalled during their interview. Static analysis tools use well-defined programming rules to find defects early in the development process, when they are cheap to fix [6]. For example, there are static analysis tools that can alert developers to synchronization issues which can lead to unsafe thread interactions. Developers have been able to eliminate many defects that were previously overlooked at large companies [9] using the warnings produced by static analysis tools.

Despite the benefits of using static analysis tools to find bugs, consistent usage of these tools is not very frequent [6]. Remember Susie, who adopted a static analysis tool to improve the quality of her code? After using the tool for a while, dealing with the interface became a burden; finding the warnings was not easy and when she did, she had a hard time interpreting the feedback. Inspecting her code without using the tool involved more work, but she prefered to do it this way to avoid the time and confusion involved with using the tool. There have been studies to investigate ways of improving static analysis tools. However, none look at what the tools do or can do for a developer, what features developers use, what could be improved *and* why [10], [11]. Our research aims to understand why software developers are not using static analysis tools and how current tools could be improved to increase usage based on developer feedback. For our study, we intend to focus on static analysis tools used to finds bugs. This includes tools like FindBugs, Lint [12] , IntelliJ [4] (which includes built-in static analyzers), and PMD [13]. FindBugs will be referenced the most as it is the tool we chose to use during our interviews.

In the following sections of this paper, we will first discuss some related work (Section II) and the methods used in our study (Section III). Section IV presents the results and threats to the validity of our study. In Section V we discuss implications for static analysis tools and finish with a discussion of future work (Section VI) and take away points (Section VII).

II. Related Work

There have been many studies on static analysis tools, many of which focus on their correctness and functionality [6], [10], [14], [15]. Unlike existing work, our work focuses on developers' perception on using static analysis tools, including interacting with the interface of the tool, and what may have caused their perceptions. Perception plays an important role in when considering human and computer interactions [16] and can be influenced by a number of things, such as the subjective preferences of the user.

Ayewah and Pugh conducted a study where they claimed that static analysis tools should help engineers find bugs as early as possible in the development cycle, when they are cheap to fix [17]. They interviewed 12 FindBugs users by phone and conducted a controlled study with 12 students to see how they use FindBugs and handle defects that are labeled "not a bug". Their work is similar to ours in that they are interested in how developers use static analysis tools. Our work builds on this work by recruiting various tool users for interactive, participatory interviews.

Khoo et al. examined and focused on the interface of static analysis tools and how the interface could be improved [11]. They developed a user interface toolkit called *Path Projection* that uses program visualizations to help developers walk through the error reports produced by static analysis tools. *Path Projection* was designed to improve and simplify the process of triaging bug reports, or labeling bugs as a false or true positives, by utilizing checklists to systematically label bugs. This study is similar to our work in that they look at improving the static analysis tool user experience. Our study builds on this study by investigating not only improving the user experience, but also finding out why these improvements need to be made from the developers who use them.

Heckman and Williams conducted research in an attempt to develop a benchmark, FAULTBENCH, that would help developers compare and evaluate static analysis alert prioritization and classification techniques [18]. The overall goal of their research was to make using static analysis tools easier and more useful to developers. Our work is related in that we are also looking for ways to improve current static analysis tools for developers. Layman et al. recruited 18 participants to investigate factors that developers may consider when deciding whether to address a defect when notified of it [19]. This study is related to our work in that a similar methodology is used and they are also interested in learning more about how developers use these tools and how it can be made easier. Our work builds on these works by focusing on various aspects of using static analysis tools, including how users interact with the tools.

III. Methodology

For this study, we conducted interviews with software developers. Each semi-structured interview lasted approximately 40-60 minutes and, with the participant's consent, was recorded. By conducting "semi-structured" interviews, we aimed to achieve the flexibility needed to get as much detailed information as possible [24]. We prepared a script of questions for the interview, but would add or omit questions on the fly depending on how detailed a participant was in their responses. We created and modified the script as we conducted trial interviews; any changes made to the script was based on the responses we got from our 4 trial participants [25].

Upon completion, we manually transcribed each session. We performed qualitative analysis [1] on the transcripts by "coding" the transcriptions. This process is discussed in detail in Section III-F.

A. Participants

We conducted this study with a group of 20 participants. Although this seems like a small sample, we followed a similar methodology to that of Layman et. al.'s study that only had 18 participants [19]. Participants were recruited using an electronic recruitment flyer that was sent out to our industry contacts to then be sent to developers within their company. Sixteen of our participants are professional developers at a large company and 4 are graduate students at North Carolina State University with previous industry experience. Participants' years of development experience ranged from 3 to 25 years. We did not explicitly ask participants about their experience building static analysis tools, however, based on conversations approximately 2 participants had tool building experience. We interviewed two participants remotely, one by phone and one by video chat, due to location differences. Each participant filled out a short questionnaire used to collect demographic information.

Table I shows the statistics and background information gathered from the questionnaire and interviews. The first column lists the participants' pseudonyms, given for confidentiality purposes. The second and third columns show the open-source tools and closed-source tools that they have used to find bugs. If a space has a "-", it indicates no response from the participant.

B. Research Questions

For this research, we want to learn:

- **RQ1**: What reasons do developers have for using or not using static analysis tools to find bugs?
- **RQ2**: How well do current static analysis tools fit into the workflows of developers? We define a workflow as the steps a developer takes when writing, inspecting and modifying their code.
- **RQ3**: What improvements do developers want to see being made to static analysis tools?

[1] All study materials including interview scripts and coding categories are available at http://www4.ncsu.edu/~bijohnso/ffsat.html

TABLE I
DESCRIPTIVE STATISTICS REPORTED BY PARTICIPANTS.

Participant	Open-source Tools	Closed-source Tools	Local
Abby	FindBugs	IntelliJ	Yes
Adam	CheckStyle, FindBugs, PMD	IntelliJ	Yes
Andy	FindBugs, Lint	Jtest [20]	Yes
Chris	CheckStyle, FindBugs, Lint	Coverity	Yes
Cody	Dehydra	-	Yes
Frank	-	-	Yes
Gordon	Lint, CheckStyle, FindBugs	-	Yes
Jake	FindBugs, Lint	FlexLint, Klocwork Insight [21], Visual Studio [22]	Yes
James	Lint, CheckStyle, FindBugs	Visual Studio	Yes
Jason	Lint, FindBugs	-	Yes
John	CheckStyle, Copy/Paste Detector(CPD), FindBugs, Lint, PMD	CodePro [23]	Yes
Jordan	CheckStyle, FindBugs, PMD	Jtest	Yes
Josh	FindBugs, Lint	Coverity	No
Lee	CheckStyle, FindBugs, Lint	Visual Studio	Yes
Matt	Lint	FlexLint, PyCharm	Yes
Mike	cpplint, Lint	-	Yes
Phil	-	-	Yes
Ray	CheckStyle, FindBugs	-	Yes
Ryan	FindBugs, Lint	Coverity	Yes
Steve	CheckStyle, CPD, FindBugs, Lint	IntelliJ	Yes
Tony	CPD, FindBugs, Lint, Splint,cpplint, PMD, Checkstyle	Coverity	No

We ask these questions because answers to these questions will give toolsmiths and researchers areas for future work and improvement in the area of static analysis tools. Research has shown that the way a tool interrupts a developer's workflow is important therefore we wanted to specifically investigate this aspect of tool usage [26], [27]. The interviews focused on developers' experiences with finding defects using static analysis tools. Learning developers' relevant experiences and observing how they use static analysis tools to find bugs may shed some light on why these tools may be underused. The interviews were organized into into three main parts: Questions and Short Responses (Section III-C), Interactive Interview (Section III-D), and Participatory Design (Section III-E).

C. Part I: Questions and Short Responses

During part 1, Question and Short Response, we asked developers questions related to their general usage, understanding, and opinion of static analysis tools in order to answer RQ1. Some of the questions asked include:

- Can you tell us about your first experience with a static analysis tool?
- Can you remember anything that stood out about this experience as easy or difficult?
- Have you ever used a static analysis tool in a team setting? Was it beneficial and why?
- Have you ever consciously avoided using a static analysis tool? Why or why not?
- What in your opinion are the critical characteristics of a good static analysis tool?

D. Part II: Interactive Interview

The second part is what we call the Interactive Interview. The goal behind the Interactive Interview is to be able to observe developers actually using a static analysis tool. This allowed us to get more detailed information as to how developers are using their tools. We aim to use the information obtained during this portion to address RQ2. We asked our participants to explain what they are doing out loud [28] so we could get a better understanding of their workflow and thought process. Practice interviews before this study revealed that using the interactive interview portion produced more detailed information regarding when and how developers use their static analysis tools [25].

Some of the questions asked during this portion include:

- Now that you have run your tool and gotten your feedback, what is your next move(s)?
- Do you configure the settings of your tool from default? If so, how?
- Does this static analysis tool aid in assessing what to do about a warning?
- Do you feel that "quick fixes" or code suggestions would be helpful if they were available?[2]

For confidentiality reasons, not all of our participants could use their own workstation for this part of the interview. For those who could not, we provided 6 open source projects in Java, such as log4j [29] and Ant [30], and asked each participant to run FindBugs on one of them. We chose FindBugs because it is one of the most popular and mature static analysis tools for Eclipse. Due to technical difficulties, our remote interviews were not able to fully experience the "interactive" portion. Each was given a scenario of static analysis tool usage and asked to, first, explain their thought process in walking through that particular scenario. We then asked the same questions as we would have asked if they had been local.

[2]Participants were only asked about quick fixes and code suggestions being useful when they mentioned, either during the Question and Answer or Interactive Interview, that they either a) find quick fixes useful, b) felt that the tool should be more helpful or c) did not understand how to fix the defect we presented them with.

E. Part III: Participatory Design

We intended the last part of the interview to get the participants to make design suggestions for improving static analysis tools. We utilized a concept called participatory design [31], which involves getting stakeholders (in this case, our participants) involved in the design process by allowing them to show what they want instead of saying it. In order to promote creativity, each participant was given a blank sheet of paper and asked to show us what they wanted their tool to look like and how it should work [25]. Participants were not required to draw something, but 6 of them did. The rest of our participants gave verbal descriptions of tool features they desired.

F. Coding Interview Responses

After completing the interviews, we manually transcribed each interview. Then, the transcriptions were *coded*. Coding is a process that is meant to make referencing transcriptions quicker and easier [32]. We used Gordon's basic steps to code our interviews and use the codings to help organize the Results (Section IV). Before coding an interview, "coding categories" need to be defined. These should be general enough for relevant information to be grouped together but detailed enough that a concrete example only falls under one category. Because of this, it is possible to have "emergent" categories that may need to be defined after reading the transcriptions. We developed and used the following coding categories: *Tool Output*, which includes anything related to the output produced by the tool (for example, false positives); *Supporting Teamwork*, which includes anything about using static analysis tools in a team or collaborative setting; *User Input and Customizability*, which highlights points made about the customizability of the static analysis tools (for example, modifying rule sets); *Result Understandability*,which includes anything said about the ability or inability to understand or interpret the results produced by a static analysis tool; *Workflows*, which is defined as anything related to the steps a developer takes when writing, inspecting and modifying their software (for example, tool integration); and *Tool Design*, which includes the proposed tool design ideas from our participants. Examples of each of these categories from the transcriptions are as follows:

Tool Output

> Jason: *"...like I mentioned with FlexLint it gives you so many warnings and sifting through them is so, arduous that whenever I just look at it I'm like ehhh forget this."*

User Input/Customizability

> Andy: *"...it's like is this list prioritized by you know what's important to me? No. You know? And there may be a default listing that should be prioritized because like this one's inefficient."*

Supporting Teamwork

> John: *"The only reason I like the batch results is to communicate, broadcast to the team a sense of progress or lack of progress."*

Result Understandability

> Matt: *so now I wanna know why raising a string exception is bad. Like what should I be doing instead? Since it thinks it's a problem. And so none of these really help me.*

Workflows

> Mike: *"Clang is my favorite. Its built into the compiler. You don't have to invoke anything special."*

Tool Design

> Chris: *"I dont mind the idea of the actual source code itself having some plasticity ... lets say the fourth line there was some error here... having the 5th line drop down and having the content expand with maybe all sorts of annotations about my code."*

The next step in Gordon's methodology is to assign "category symbols" to each category for easier indexing and processing of information. Gordon then suggests finding and classifying the relevant information in the transcriptions using the category symbols. In our codings, each coding category had its own color as a "symbol"; if a portion of a participant's transcription fell into one of the categories, the text would be highlighted the same color as its respective category. A participant's coded interview could contain multiple categories or even multiple data items for one category. To ensure consistency, one person was responsible for coming up with the coding categories and "symbols" and going through the transcriptions to apply them. The last step is to check the reliability of the codings. For our study, once the codings were complete, it was passed off to the other contributors to look over. If there were any discrepancies they were discussed and resolved as a group. This includes items that could fall into more than one category; in this situation, either a new, more specific, category or a "sub-category" was created for the item. The purpose of the categories are to organize the data in a relevant and useful manner; they are not meant to directly correlate with the research questions.

IV. Results

In this section, we will discuss the results we obtained. We answered our research questions by linking the questions to coding categories and interview parts. After analyzing the results, we believe the following to be true:

- Our first research question (RQ1) can be answered by observing the results that have been categorized under "Tool Output," "Supporting Teamwork," "User Input and Customizability," "Result Understandability" and "Developer Workflows"; the information collected in these categories could be reasons why developers are or are not using static analysis tools.
- Our second research question (RQ2) can be answered by observing the results that have been categorized under "Developer Workflows."
- Our third research question (RQ3) can be answered by observing the results that have been coded under "Tool Design"; most of these results are from the Participatory Design portion.

In each category, we expected there to be negative and postive remarks about current tools, both of which are equally important in answering our research questions; anything positive could be a reason for use while anything negative could be a reason to discontinue use. For each coding category, we separated the relevant statements into positive statements and negative statements; if something good is said about a static analysis tool it's considered a *positive* comment and vice versa for a *negative* comment. In Figure 1, we can see that the majority of our participants have had problems with tool output, customizability and workflow integration, and all but one of our participants have had problems with understanding results. Tool design is not included because this category was defined to capture the developers' ideas for improving static analysis tools. Their reasons for wanting the features are captured in the other categories.

A. RQ1: Reasons for Use and Underuse

Our interviews revealed that there are a variety of reasons developers may have for choosing to use a static analysis tool to find bugs in their code. One of the obvious reasons is because too much time and effort is involved in manually searching for bugs. Five out of our 20 participants feel that because static analysis tools can automatically find bugs, they are worth using. During his interactive interview, Jason told us "*anything that will automate a mundane task is great.*" In other words, one reason for using static analysis tools is that they automates the process of finding bugs.

Another reason developers might use a static analysis tool is if it is already available in the development environment and ready to be used. For 3 of our participants, this was the case. Development environments such as IntelliJ and PyCharm come with built-in static analyzers, which requires little extra effort on the developer's part. Two of our participants, Matt and Adam, use PyCharm and IntelliJ regularly and like the fact that static analysis is already integrated. For 7 of our participants, a good reason to use static analysis tools is to support team development efforts. According to Josh and Andy, static analysis tools do this by raising awareness of the potential problems, or "dumb mistakes," in the code earlier in the development process. For Cody and Ray, static analysis tools are useful for communicating and enforcing coding standards and styles on development teams. Some developers enjoy using the static analysis tools they use to find bugs because of the level of customizability. Three of our participants fit into this category. According to James, the customizability of a tool can play a large part in the volume and quality of output developers get.

Although some of our participants could find reasons to use static analysis tools to find bugs, most of our participants brought up conflicting concerns that could make the decision to adopt and use a static analysis tool less obvious.

Tool Ouput. Tool output was a popular dicussion topic. Out of the 20 people we interviewed, 14 people expressed the negative impacts of poorly presented output. Static analysis tools are known to produce false positives and these false positives can "outweigh" the true positives in volume [33]. Another known fact is that, especially with larger projects, the number of warnings produced by a tool can be high, sometimes in the thousands [9]. Some of our participants felt, however, that false positives and large volumes of warnings would be less burdensome if the way the output is presented was more user-friendly and intuitive. Cody, who likes using Dehydra, finds himself frustrated at times because the results are dumped onto his screen with no distinct structure causing him to spend a lot of time trying to figure out what needs to be done. Jason wishes that his tool's output would be a "slice" that shows what the problem is and what else could be affected in order to more quickly assess what is or is not important. This "slice" should be taken from the entire project, using call hierarchies, to show which parts are affected by each defect. During his Interactive Interview he commented on a previous experience with FindBugs. He had a large list of warnings to scroll through but without there being any context to the problems it just seemed like "a bunch of junk to sift through," which made him not want to bother using it. It may be worth investigating how valuable an output like this would be.

Collaboration. In industry, software development is often a team effort. For 9 of our participants, lack of or weak support for teamwork or collaboration is one reason that teams, as well as individual developers, may not adopt or regularly use static analysis tools. According to John, although static analysis tools are useful for trying to enforce coding standards, there is no easy way to share the settings with other people on the team so it ends up being a cumbersome manual process and causing confusion when the standards need to be changed. Many of our participants mentioned the desire for a way to easily communicate and collaborate when using their static analysis tool, especially in a team setting. Although static analysis tools can be beneficial in team settings, current tools are not collaborative enough for some developers. Newer versions of FindBugs offer a cloud storage feature that can be used store, share and discuss warning evaluations [34]. Although a feature like this does make it easier to communicate and share warning evaluations between developers, to add a comment to a bug or current evaluation a web browser is needed. This takes the devloper out of context and out of the development environment which could demotivate some individuals from checking them when they should.

Customizability. For 17 of our participants, customizability is important however many tools are not trivial to configure and do not accomodate the customizations that developers want. False positives and large volumes of warnings are well-known downsides to use static analysis tool to find bugs, however Frank told us he believes that the way you configure your tool plays a large part in the output you get. John stated during his interview that "*many tools are so hard to configure, they prevent you from doing anything.*" Sometimes it is difficult just to get to the menu where the options for configuring a particular feature are, which participants Matt and Josh agree with. One of our participants, Jake, found himself in an interesting situation during his Interactive Interview where he could not figure out how to customize his

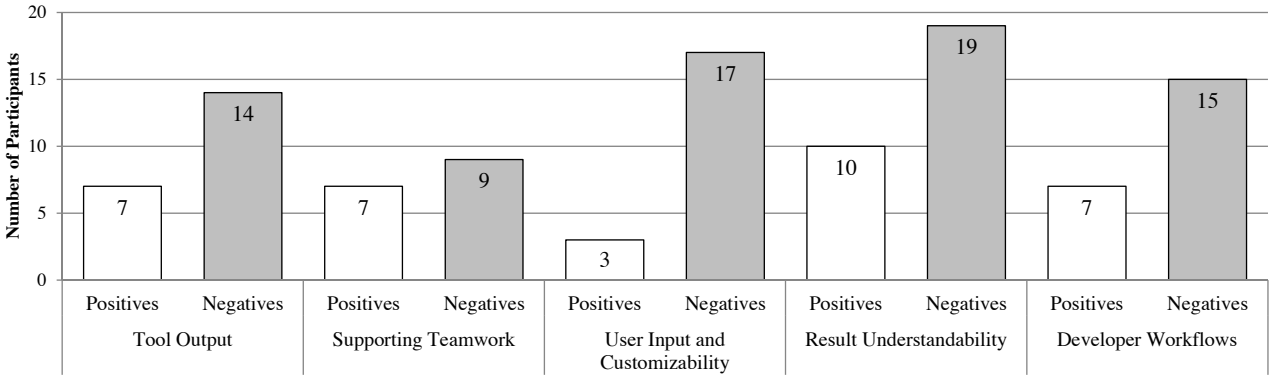

Fig. 1. The number of participants in each category expressing the good and the bad about static analysis tools they have used.

tool and wound up having to search the web to find out where the tool's preferences were. A common problem expressed by most of the participants is the inability to temporarily ignore or suppress certain warnings. Although some static analysis tools allow developers to turn off certain filters, not all developers are comfortable with turning warnings completely off. Matt, for example, is afraid that he may not remember to turn it back on. The notion of dismissing or ignoring static analysis warnings may be too coarse; as Jordan noted, he would prefer that static analysis tools offered a way of recording his judgement about that warning. More sophisticated judgements may include things like "this warning isn't a problem now, but may be in the future if the following conditions are met…".

Result Understandability. The main objective when using a tool like FindBugs is to learn what defects are in the code so that problems can be removed. A developer not being able to understand what the tool is telling her, according to our participants, is a definite barrier to use. Nineteen of our 20 participants, felt that many static analysis tools do not present their results in a way that gives enough information for them to assess what the problem is, why it is a problem and what they should be doing differently. James told us during his interview that "*it's one thing to give an error message, it's another thing to give a useful error message.*" When talking about the Eclipse Python plug-ins, he also stated, "*I find that the information they provide is not very useful, so I tend to ignore them.*" A few participants felt that it would be helpful to have links to more details or examples in the error reports. In some situations more information is needed to understand exactly what the problem is and why it is a problem; understanding why a defect is a problem can help the developer better assess whether the error is a false positive and try to avoid repeating the same problem. Ryan told us during his Interactive Interview that a start would be using "real words," or a more natural language, to explain the problem.

The most frequently mentioned difficulty when using static analysis tools is lack of or ineffectively implemented quick fixes. Most of our participants expressed interest in having their tool provide code suggestions or quick fixes that assist them when attempting to fix a bug; Abby proclaimed "*if you can tell me it's an error, you should be able to tell me how to fix it.*" Jordan strongly agrees; he loves tools that have quick fixes and hates tools that do not. According to our interviews, these fixes do not have to be automatic; some prefer that code suggestion previews be used or possibly using examples to get a better understanding of how to fix the problem. Some participants expressed interest in but skepticism toward integrating quick fixes into static analysis tools. For example, during Jordan's Interactive Interview, he noted that sometimes when using multiple tools, they may have conflicting quick fixes or solutions. In Frank's past experiences with automated code changes, he has had to do manual refactorings because something was done wrong; because of this, he prefers to use find and replace to make his own changes. Another participant, Adam, was concerned with knowing whether the semantics of his code would be preserved after applying a quick fix. Most static analysis tools, if they offer quick fixes, leave it to the developer to figure out exactly what has been done after it has been done. Almost all of our participants agree that effectively designed quick fixes can help them to better understand the problems its tool is telling them about, leading to a better sense of productivity for the developer.

B. RQ2: Workflow Integration

The most common topic during the interviews was "tool/environment integration." Sometimes a developer's process includes running a static analysis tool, but more often it is not part of a developer's workflow to stop and run a tool in the middle of working on some code or a specific task; she usually prefers finding a "stopping point" in her code to run the tool [19]. Analysis of our interviews reveal that while this is true, there are many different ways that developers may want their tool to fit into their development workflow. For example, some developers prefer that the tool run in the background; it is easier for them to figure out what is wrong if they are in the process of doing it and do not have to think about invoking the tool. On the other hand, some developers do not use IDEs, so if they are to use a static analysis tool, compiler integration is very important. Nineteen of the 20 developers we interviewed expressed the importance of workflow integration to them and how these needs have or should be met.

For some of our participants, there are features of static analysis tools they have used that helped the tool better integrate into their workflow leading to increased usage of the tool. In fact, John feels that static analysis tools can be used to help organize your workflow, based on the results it produces. For example, if you are running a static analysis tool on some code for the first time, it can be a good indicator of the kinds of bugs the tool finds and that may be present; this can give an idea as to how detailed of an analysis the tool does, possibly giving you a better idea of when it would be best for you to run it. Of all the tools Adam has used in the past, he much prefers to use IntelliJ and its built-in static analysis to find bugs; they are tightly integrated making it seem more "real time". For these participants, as well as a few others, integration with the development environment plays a major role in their decision to use or continue using a static analysis tool. Common standalone static analysis tools like FindBugs and PMD have the ability to integrate with IDEs like Eclipse and NetBeans which becomes especially important when you are using more than one static analysis tool at a time, as we learned from discussing a past experience of Steve's where he was using 3 different static analysis tools. Jordan and Chris like how FindBugs, PMD and CheckStyle fit into their development processes; for Jordan, it is an integral part of his workflow. For the majority of our participants, however, current static analysis tools are not doing enough to effectively integrate into their development process.

One of the biggest demotivational forces on a developer when it comes to using a static analysis tool to find bugs is when it is what Tony calls a "disjoint process." Many of our participants, especially those who do not use IDEs, do not like when they have to go out of their coding environment to use a tool or view the results produced by the tool. For example, Frank, Lee, James and Andy commented on how "painful" it was during their Interactive Interview to have to switch perspectives in FindBugs to explore the complete listing of bugs. According to Lee, having to open another perspective to know what is going on is a guarantee that unmotivated people will not do it. For Frank, although it is nice that the results are hidden so that you are not overwhelmed, having to go back and forth and drill down to see the bugs requires extra effort and is disruptive to his workflow. Other tools our participants had similar complaints about was Coverity and Lint for C/C++ projects. For Ryan and Tony, the biggest downside to using Coverity is that it is not capable of being integrated into their coding environment, leading to a lot of clicking back and forth between their editor and the static analysis tool. Phil does not like using Lint because of the fact that he has to "go out of his way" to do so.

Some of our participants made it clear, however, that even if the tool is integrated with their development *environment*, it is still possible that the tool does not integrate well into their development *process*. For example, one of our participants, Mike, does not use IDEs so using a tool that integrates well with an IDE does not fit well into his development process; he likes using Clang because it can be tied into his compiler which does not require a "development environment". According to Gordon, one of the key problems with static analysis tools is that at times they can prevent him from being productive. One way this can happen is when the tool slows the developer down by taking a long time to run, which was a common complaint amongst our participants. From Jason's experience, he believes that "*if it disrupts your flow, you're not gonna use it.*" Jason's statement rings true among other participants as well, like Steve who has used various tools in his past but does not like to use FindBugs because, even though it is IDE integrable, it runs slow. IntelliJ, which contains built in static analyzers, utilizes *idle time* when reporting bugs in an attempt to prevent the problem of interrupting the developer's workflow but for Matt, it can still be bothersome. Jason believes that the problem with current static analysis tools is that they are not capable of running well on larger code bases, leading to a break in his "development flow" as he waits for the tool to catch up.

In terms of workflow, participants valued using static analysis both to fix bugs once they are introduced into the program, but also later in the development process. From a workflow standpoint, it is valuable to fix potential bugs when they are entered into a program because the necessary context to understand the bug is already in the developers' working memory. In contrast, fixing bugs later is difficult because a developer must recall the context to analyze the corresponding static analysis warning. This contrast is similar to the difference between "floss refactoring" and "root canal refactoring," where the former involves restructuring code as it is being worked with and the latter involves refactoring by finding the "worst code" and dealing with that first [35]. Root canal refactoring is a discouraged practice and its analog in static analysis – finding the most severe static analysis warnings in a whole codebase and dealing with those first – may also be a wasteful practice. Research has shown that many static analysis warnings in working systems do not actually manifest as program failures [9].

C. RQ3: Tool Design

Our main goal in this research is to improve static analysis tools for developers. The best way to do this is to find out how developers want their tool to be designed. Most of the proposed designs are for warning notification and manipulation or quick fix display. Participants made some other interesting proposals which will also be presented.

Quick Fix Design. Ten of our participants made a suggestion related to the way in which a quick fix should be displayed. Most of our participants wanted to be able to preview the fix and how it is going to change their code before they apply it. Abby and Tony recommended splitting the code editor to show a diff of the code, using highlighting to show what code has changed or been added to their code. On one side there would be the code now and on the other the code once the fix is applied. Some felt that you should be able to see the fix before applying it, but then also manually apply it so that you know the fix is being applied without introducing any

new problems. One participant, Mike, prefers not to have quick fixes at all because he feels the error messages are enough to assess what to do about an error.

One interesting quick fix design idea, which came from Ryan during his Interactive Interview, was to have what he called a "*three option dialog box*" available when applying a quick fix. This dialog box would pop up upon a click to fix the bug and there would be three choices: apply the entire fix (default option), do not apply the fix or step by step apply the solution allowing the developer to decide which parts of the solution they would like to keep. Static analysis tools like FindBugs and IntelliJ offer some quick fixes. However they do not give a full context preview of the changes that will be made, leaving it to the developer to manually ensure that the fix was applied correctly and to their liking.

Warning Notification and Manipulation Design. All 20 of our participants told us when and how they want to be notified of errors in their code. The theme in this category is "fast." Developers want tools that provide faster feedback in an efficient way that does not disrupt their workflows. For some of our participants, this meant running the tool in the background of the IDE so that feedback occurs as soon as a problem is detected. For other participants, this meant running the tool at build time or compile time. In this way, the results are presented when the developer is at a "stopping point." [19]. Overall, our participants find that current static analysis tools are not fast enough when providing them with feedback; this quickness should be accompanied with discretion as the developer does not want the tool to break their thought process.

Our participants also thought it would be beneficial to have the ability to easily make "judgements" about defects, such as setting it aside to view later, save these judgements and share them with other developers. Many of our participants suggested that static analysis tools should allow developers to ignore specific defects and move them to their own list for later viewing, a form of *temporary suppression*. Most tools, if they allow the developer to ignore specific warnings, only allow the developer to turn off or suppress a bug category for particular line of code using a comment-like annotation, which Gordon told us makes the code "smell". Developers would like to have the option to ignore each individual defect in case they either do not want to fix it and do not want to be bothered by it again or do not want to be bothered with it at that particular time but would like to come back to it later.

Other Design Ideas. Our participants also came up with creative design ideas. One participant, Chris, suggested giving the editor "plasticity". When he is given a warning and would like to get more information, the tool should move the code surrounding the warning to embed this information into the editor. A couple of our participants thought it would be useful to have visual output, possibly a pie-style diagram of the project and the bugs in it, instead of standard list and tree outputs to make it easier to go back and forth between warnings and code. During Frank's Participatory Design session, he suggested a potential solution; a parts-to-a-whole corpus view of the project as a *"heat map"*. The heat map would use colors to show where the errors are and how severe the problems are. It would start with an overall "view" of the project and as you drill down you can see the condition at each level to see where the most attention is needed. This is similar to the concept behind Khoo's toolkit *Path Projection* in that the toolkit is meant to visualize output that is usually, if not always, textual and difficult to understand [11].

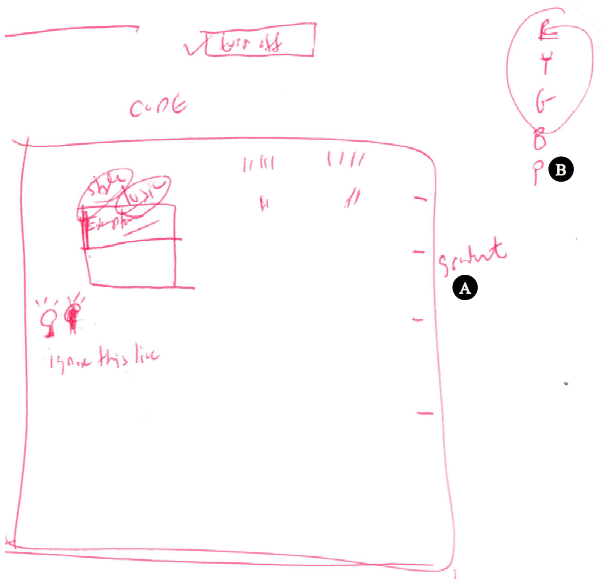

Fig. 2. One of our participant, Matt's, Participatory Design drawing; (A) shows where Matt wants the gradient colors and (B) shows the way his current tool represents severity.

An interesting suggestion made by a couple of our participants is to represent the severity of the defects using gradients of one color instead of multiple different colors; the darker the color the more important or urgent the bug is. Figure 2 depicts a drawing one of our participants, Matt, drew during his Participatory Design; he labeled the side of the editor "gradient" (A) where he would like to see his severity representation. In the top right corner, Matt also lists the colors that his current tool uses (B); for example, "R" means red. The idea behind this is not new; other studies have focused their attention on using colors for error representation [36], [37].

D. Threats to Validity

There are several threats to the validity of our study; here we categorize each threat as a threat to external, internal, or construct validity.

External. One limitation to the generalizability of our study is the sample size. Although we obtained valuable information from the 20 interviews, due to time constraints (and busy developers) they may not be representative of the larger population that use static analysis tools. Although we would have liked more participants, having a large number of interviews to transcribe and code could lead to less accurate analysis. The study conducted by Layman et al. [19], which we discussed earlier as utilizing a similar methodology, had a participant pool of similar size (18). Another possible threat

is that we only interviewed developers who have used static analysis tools. In some cases it may be that static analysis tools are not being used for other reasons, such as lack of awareness. It should also be noted that some of our participants had experience building static analysis tools, giving them somewhat of a biased opinion of the usage of these tools.

Internal. Another threat to the validity of this study is the way in which we conducted remote interviews. We did not thoroughly prepare for what we would do if the technology we wanted to use did not work or was not available. Therefore, the Interactive Interview and Participatory Design in remote interviews had to be conducted differently than local interviews. Despite this, there was still value in the results obtained from our remote participants; they could still give useful insights from their previous experiences. Only 2 of the interviews fell into this category, so this helps limit the impact of this threat.

Construct. The objective for using the Interactive Interview was to get more accurate information on how developers use their tools. One limitation here is that some developers were not as familiar with the code or environment they had to use in our interviews as they would be with their own code in their own development environment. This could have caused some developers to take different actions than they would if they were in their own environment. Ideally it would have been better to have been able to observe our participants working in their own environment; however, for confidentiality reasons, we were not allowed to view participants' own proprietary code. In an effort to compensate for this threat, the open source projects and tool we chose are well-known open source projects. Another threat to the validity our work is that we did not originally consider is that we may have said things in our consent form or session script that would give unintended "hints" to our participants concerning our research expectations. One example of this is us outlining our research goals in the introductions we gave prior to beginning each session. This could have led to what is called "hypothesis guessing" where participants respond to questions based on what they think the researcher wants to hear [38]. In retrospect, we helped alleviate this threat in our interviews by asking our participants experience questions.

V. Discussion

A. Implications

Our interviews have several implications for current and future static analysis tools. Current static analysis tools may not give enough information for developers to assess what to do about the warnings produced and very seldom offer a fix to what it claims is an issue. If static analysis tools offered quick fixes, giving a potential solution and applying it to the problem may help developers assess warnings more quickly and ultimately save time and effort. Our results indicate that FindBugs, for example, would be more useful if it had more informative messages and offered quick fixes. At the same time, quick fixes do not appear to be a universally applicable mechanism to help developers resolve static analysis warnings because many static analysis warnings do not have a small set of solutions. For example, FindBugs warns developers when two method names in the same class differ only by capitalization; no quick fix for this problem is likely to satisfy a developer. Instead, interactive quick fixes that enable easy access to refactoring and code modification tools may be able to semi-automatically help developers resolve static analysis warnings. On the negative side, quick fixes could also cause developers to be hasty in fixing their code, which could potentially lead to more problems, such as the introduction of new defects. There are also challenges related to implementing usable interactive quick fixes. We have not yet investigated what theses challenges are or how to address them as they are out of scope for this particular study.

Developers like tools like IntelliJ and FindBugs because they have the ability to run without the developer telling it to, however, there is still the issue of giving the developer information they find useful. One way to allow developers to focus on making judgments about defects is to treat each warning like a Mylyn task [39], where the program elements that are explored when making a judgement, such as the assignments to a variable when judging a null-pointer warning, automatically populate a warning's task-context. In this way, extraneous warnings and program elements not related to a warning under investigation can be automatically elided, reducing distractions. Like Mylyn task contexts, such "judgement contexts" could also be saved and passed around between developers, enabling knowledge about static analysis issues to be more easily shared.

Developers may prefer a tool where the usage is tied into their normal workflow. For example, if a developer has to commit their code to a repository so many times a day, they may be more likely to use a static analysis tool if it can be run each time they go to commit their code; this way they do not have to go out of their way to use the tool. Developers may want also features such as the ability to modify existing warnings or rule sets or choosing how and when their tool runs. Most static analysis tools for finding bugs today offer some type of customization to the bugs it finds; for example, in FindBugs and IntelliJ it is fairly simple to turn off or suppress warnings for any of the categories of bugs that the tool finds. If a tool is to be customizable, it should be customizable in a way that is simple and useful to the developer. FindBugs allows developers to turn off certain bug detectors but only on a project level. Turning off a detector for a specific class or file has to be done manually in the code at each line you want ignored. Configurations that require this much effort may cause the developer to discontinue configuring the tool, which could eventually lead to the developer discontinuing use of the tool.

VI. Future Work

The results from our study suggest that there are ways to make static analysis tools more useful to developers. In the future it may be necessary to perform a follow-up study that focuses on the adoption of static analysis tools to give a more holistic view of what factors developers consider

when choosing to use a static analysis tool. We have begun to implement a static analysis tool prototype based on the results we obtained in this study. One of the main features we plan to focus on are defect remediations, as this seemed to be one of the most frequently mentioned requests made by our interviewees. More specifically, we are interested in implementing interactive quick fixes, giving the developer more enhanced control over the "automatic fix," beyond what would normally be offered in a one-shot quick fix. We also plan to conduct a user study to evaluate our prototype with software developers with a range of experience.

VII. Conclusion

In this paper, we investigated why developers do not widely use static analysis and how current tools could be improved to increase usage. We conducted a user study involving 20 software developers who have an average of about 10 years of experience with using static analysis tools to find bugs. We also discussed the implications of our results.

Our results confirmed that false positives and developer overload play a part in developers' dissatisfaction with current static analysis tools. Each of the factors presented in this paper should also be considered when implementing a tool that will lead to higher usage of static analysis tools for improving software code quality and maintaining coding standards. Future static analysis tools could improve adoption by software developers by enhancing support for team development while using static analysis tools, improving integration of the tool into developers' processes, having intuitive defect presentation and detailed explanation of defects with automatic fixes where appropriate, and including easy and useful configuration options for the tool.

Acknowledgements

We would like to thank Nat Ayewah and our participants for their contributions. This material is based upon work supported by the National Science Foundation under Grant No. 1217700 and a Google Faculty Award.

References

[1] L. C. Briand, W. M. Thomas, and C. J. Hetmanski, "Modeling and managing risk early in software development," in *Proc. ICSE*, 1993, pp. 55–65.
[2] N. Nagappan and T. Ball, "Static analysis tools as early indicators of pre-release defect density," in *Proc. ICSE*, 2005, pp. 580–586.
[3] M. Gegick and L. Williams, "Towards the use of automated static analysis alerts for early identification of vulnerability- and attack-prone components," in *Proc. ICIMP*, 2007, pp. 18–23.
[4] "IntelliJ IDEA," http://www.jetbrains.com/idea/.
[5] "FindBugs," http://findbugs.sourceforge.net.
[6] N. Ayewah, D. Hovemeyer, J. D. Morgenthaler, J. Penix, and W. Pugh, "Using Static Analysis to Find Bugs," *IEEE Softw.*, vol. 25, no. 5, pp. 22–29, 2008.
[7] "Eclipse," http://www.eclipse.org/.
[8] "NetBeans," http://www.netbeans.org/.
[9] N. Ayewah and W. Pugh, "The Google FindBugs Fixit," in *Proc. ISSTA*, 2010, pp. 241–252.
[10] A. Bessey, D. Engler, K. Block, B. Chelf, A. Chou, B. Fulton, S. Hallem, C. Henri-Gros, A. Kamsky, and S. McPeak, "A Few Billion Lines of Code Later: Using Static Analysis to Find Bugs in the Real World," *Commun. ACM*, vol. 53, no. 2, pp. 66–75, 2010.
[11] Y. P. Khoo, J. S. Foster, M. Hicks, and V. Sazawal, "Path projection for user-centered static analysis tools," in *Proc. PASTE*, 2008, pp. 57–63.
[12] S. C. Johnson, "Lint, a C Program Checker," Bell Laboratories, Tech. Rep., 1978.
[13] "PMD," http://pmd.sourceforge.net/.
[14] B. Chess and J. West, *Secure Programming with Static Analysis*. Addison-Wesley Professional, 2007.
[15] K. Vorobyov and P. Krishna, "Comparing Model Checking and Static Program Analysis: A Case Study in Error Detection Approaches," in *Proc. SSV*, 2010, pp. 1–7.
[16] M. Dastani, "The role of visual perception in data visualization," *Journal of Visual Languages and Computing*, vol. 13, no. 6, pp. 601–622, 2002.
[17] N. Ayewah and W. Pugh, "A report on a survey and study of static analysis users," in *Proc. DEFECTS*, 2008, pp. 1–5.
[18] S. Heckman and L. Williams, "On Establishing a Benchmark for Evaluating Static Analysis Alert Prioritization and Classification Techniques," in *Proc. ESEM*, 2008, pp. 41–50.
[19] L. Layman, L. Williams, and R. St. Amant, "Toward reducing fault fix time: Understanding developer behavior for the design of automated fault detection tools," in *Proc. ESEM*, 2007, pp. 176–185.
[20] "Jtest," http://www.parasoft.com/jsp/products/jtest.jsp.
[21] "Klocwork Insight," http://www.klocwork.com/products/insight.
[22] "Microsoft Visual Studio," http://www.microsoft.com/visualstudio/.
[23] "Google CodePro AnalytiX," http://code.google.com/javadevtools/codepro.
[24] S. Hove and B. Anda, "Experiences from Conducting Semi-structured Interviews in Empirical Software Engineering Research," in *Proc. METRICS*, 2005, pp. 1–10.
[25] B. Johnson, "A Study on Improving Static Analysis Tools: Why are we not using them?" in *Proc. ICSE, Student Research Competition*, 2012.
[26] T. Robertson, S. Prabhakararao, M. Burnett, C. Cook, J. Ruthruff, L. Beckwith, and A. Phalgune, "Impact of interruption style on end-user debugging," in *Proc. CHI*, 2004, pp. 287–294.
[27] J. Gluck, A. Bunt, and J. McGrenere, "Impact of interruption style on end-user debugging," in *Proc. CHI*, 2007, pp. 41–50.
[28] C. H. Lewis, "Using the "Thinking Aloud" Method In Cognitive Interface Design," IBM, Tech. Rep. RC-9265, 1982.
[29] "log4j," http://logging.apache.org/log4j/.
[30] "ANT," http://ant.apache.org/.
[31] C. Spinuzzi, "The Methodology of Participatory Design," *Technical Commun.*, vol. 52, no. 2, pp. 163–174, 2005.
[32] R. Gordon, "Coding interview responses," in *Basic Interviewing Skills*. Waveland Pr Inc., 1998, pp. 183–199.
[33] H. Shen, J. Fang, and J. Zhao, "EFindBugs: Effective error ranking for findbugs," in *Proc. ICST*, 2011, pp. 299–308.
[34] "FindBugs Cloud Storage," http://findbugs.sourceforge.net/findbugs2.html#cloud.
[35] E. Murphy-Hill and A. P. Black, "Refactoring Tools: Fitness for Purpose," *IEEE Softw.*, vol. 25, no. 5, pp. 38–44, 2008.
[36] B. Oberg and D. Notkin, "Error reporting with graduated color," *IEEE Softw.*, vol. 9, no. 6, pp. 33–38, 1992.
[37] E. Murphy-Hill and A. P. Black, "An Interactive Ambient Visualization for Code Smells," in *Proc. SoftVis*, 2010, pp. 5–14.
[38] "Threats to Construct Validity," http://www.socialresearchmethods.net/kb/consthre.php.
[39] M. Kersten and G. C. Murphy, "Mylar: a degree-of-interest model for IDEs," in *Proc. AOSD*, 2005, pp. 159–168.