Why and How Do Software Developers Use Static Analysis Tools?

Anonymous Author(s)

ABSTRACT

A growing number of companies use static analysis to automatically check their software for bugs and errors. Increasingly complex analyses are created and commercialized every year. An important–yet often overlooked in research–aspect of successful static analysis tools is its usability: how well the tool's interface supports the developer is key in how they understand the reported warnings and fix their code. To understand how to best design an analysis tool, we conducted a case study for the usage of static analysis tools in a software company. Through a survey of their developers and an analysis of one of their static analysis reports, we study how analysis tools are deployed in the company's systems, what the company's goals with those tools are, which strategies developers use to meet those goals, and how to best support them in doing so through interface design, tooling support, and company initiatives.

CCS CONCEPTS

• **Software and its engineering** → **Software testing and debugging**; • **Theory of computation** → **Program analysis**; • **Human-centered computing** → **Empirical studies in visualization**; *Collaborative and social computing systems and tools.*

KEYWORDS

Program analysis, Development tools, Integrated environments, Graphical environments, Usability

ACM Reference Format:

Anonymous Author(s). 2019. Why and How Do Software Developers Use Static Analysis Tools?. In *Proceedings of ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2019).* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Static analysis tools are increasingly being used in industry. From simple linters to more complex tools such as CodeSonar [13] and Fortify [24], static analysis has improved over time to detect more bugs faster and with better accuracy. However, as the underlying analyses get better, the analysis tools themselves have been improving at a slower rate. Usability issues such as poor explainability and slow updates still lead to misinterpreted warnings and tool abandonment [4, 6, 17, 18].

To understand how to best support developers when designing or setting up static analysis tools in industry, we conducted a study within a software company that we will refer to as Acme throughout this paper for anonymization reasons. Our study focuses on the company's objectives with analysis tools, the developers' goals and motivations when they use analysis tools, and how both aspects influence the way that developers interact with the tools. Through surveys and interviews, past studies have reported common usability issues in analysis tools such as waiting times or the difficulty of explaining warnings, and derive guidelines on how to improve the tools [3, 9, 17]. Unlike those studies, we focus on developer motivation and how it affects their use of the tools in the context of the company, to grasp a more realistic understanding of how analysis tools can be improved to suit their needs.

In this paper, we make the following contributions:

- We present the results of surveying 87 developers at Acme, focusing on how the company integrates static analysis in their systems, in which contexts developers interact with analysis tools, what their objectives and motivations are, which strategies they use to work through analysis warnings, and which analysis features they find most useful.
- We extract usage information from analysis reports over time from two large projects at Acme, to verify and complete the survey data on developer behaviour and strategies when using analysis tools.
- Following the results of the study, we advocate for the need to take developer goals and motivation into account when designing analysis tools, and the need for the company to properly support their usage. We also identify recommendations for building and using such analysis tools in industry.

2 STUDY ENVIRONMENT

To understand how developers interact with static analysis tools, we sent a survey across the main development teams at Acme, asking developers about their experience with static analysis tools. We were also given access to the reports of some analysis runs, including information on how developers handled the warnings, over several months for two projects at Acme. In this section, we present the composition of the survey, the analysis reports, and our methodologies for designing the survey and extracting the data.

Our study aims at answering the following research questions:

RQ1: How does Acme use static analysis tools?
RQ2: In which contexts do developers use analysis tools?
RQ3: How do developers fix analysis warnings?
RQ4: Which tool features are important to developers?

2.1 Developer Survey

2.1.1 Survey Design. The survey is composed of 40 questions (referred to as **Q1**–**Q40**) grouped into the following six categories. Unless specified otherwise, all questions are multiple choice questions with an "Others" free-text field.

(1) *Participant information:* We asked the developers about their background: how long they have worked as a developer (**Q1**) and which programming languages they work with (**Q2**).

ISSTA 2019, 15-19 July, 2019, Beijing, China

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM.
https://doi.org/10.1145/nnnnnnn.nnnnnnn

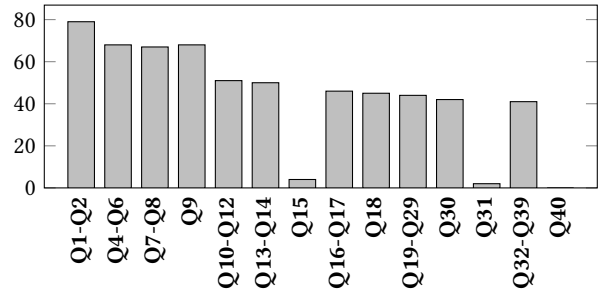


Figure 1: Number of responses per question.

(2) *General use of static analysis tools (**RQ1**–**RQ2**):* We asked the developers general questions on how analysis tools are used at Acme. This category includes questions on which analysis tools they use at the moment (**Q4**), when those tools are run in the project (**Q5**), who configures them (**Q6**), what kind of issues are detected (**Q7**), what kind of issues they would like the tools to report (**Q8**), if they fix the analysis warnings themselves (**Q9**), and who reviews the fixes (**Q29**).
(3) *Reporting warnings (**RQ1**–**RQ2**):* This category contains questions about the format in which the tools report warnings (**Q10**), which format the developers would prefer (**Q11**–**Q12**), how long developers take to fix a warning (**Q13**), and how long they typically wait before their fix is verified (**Q14**). In a free-text question, we asked developers to comment on the reporting systems of analysis tools (**Q15**).
(4) *Working context (**RQ3**–**RQ4**):* We asked developers which analysis tool they use the most (**Q16**), and focus on that tool for the rest of the category. We asked them how often (**Q18**), when (**Q19**), and where (**Q20**) they use it, how long they use it for (**Q21**–**Q22**), and why they use (and stop using) it (**Q23**–**Q24**). We also queried developers about which parts of the tool's interface they use the most, when they open (**Q26**), work with (**Q28**), and close (**Q27**) the tool, and if they are using the default layout of the interface (**Q25**).
(5) *Features of analysis tools (**RQ4**):* **Q30** and **Q31** (the latter, as a free-text question) asked developers to evaluate how important different tool features are.
(6) *Fixing analysis warnings:* This category reports on the ratio of warnings developers investigate (**Q32**), understand (**Q35**), and for which they seek help from colleagues (**Q38**). It details the strategies used by developers to choose which warnings to investigate (**Q33**–**Q34**), the reasons why certain warnings are difficult to understand (**Q36**–**Q37**), and why developers ask for help (**Q39**). Finally, (**Q40**) asks about final comments on analysis tools as a free-text question.

2.1.2 Participants, Pilot Survey, and Data Extraction. We ran a pilot survey on five developers, after which we compacted the survey so that it could be completed in approximately 20 minutes. Namely, we removed a category similar to **Q30**, about the presence of the tool features in the analysis tools that developers use.

We reached out to 120 developers at Acme (two thirds of the development force) and received 87 responses, with an exceptionally high response rate of 72.5%. From those participants, 53 developers fully completed the survey, yielding a drop rate of 39.1%. Figure 1 details the response rates. In the paper, when we report on percentages of participants, we take the number of responses to the corresponding question as the baseline, instead of the overall number of 87 participants. The percentages may also add up to more than 100%, because some questions allow the selection of multiple responses. Due to space limitations, we only report on the most popular or relevant responses. We have made the complete list of questions and anonymized responses available online [2].

In the survey, 46.8% of the participants have 2–5 years of experience as software developers, 25.3% have 5–10 years, 13.9% have 1–2 years, 10.1% have more than 10 years, and 3.8% have less than a year (**Q1**). While the large majority work with Java and Android (91.1%), Javascript (38%), C/C++ (10%), PHP (7.6%), Python (7.6%), and 12 other languages, each used by fewer than 2.5%, are also used (**Q2**). Due to Acme's policies on the usage of static analysis tools, all participants have experience with them. Our survey thus gathers information from a diverse group of developers.

All survey questions but three are multiple choice questions, for which we straightforwardly report the results. We either re-categorized responses provided in the "Others" fields in existing categories when possible (e.g., "15" was re-categorized in "> 10 years" for **Q1**), ignored when they did not answer the question (e.g., "Not applicable" for **Q24**), or kept in the "Other" category.

2.2 Analysis Reports

To complement the survey with respect to developer strategies for triaging and prioritizing analysis warnings (**RQ3**), we accessed the analysis reports of one of the main static analysis tools used at Acme, which we refer to as Dynamite in this paper. The tool is used by 51.2% of the survey participants. Dynamite is a *dedicated tool*, meaning that it is independent of the tools used by developers (e.g., code editors or project management systems). Dynamite has an elaborate Graphical User Interface (GUI) that provides developers with information such as warning categories (e.g., SQL injection), an estimate of their severity, general warning statistics, etc. Dynamite also allows developers to comment on the reported issues,, for example, marking them as false positives or fixed.

In this paper, we study the analysis reports of two projects, which we name Coyote and Road Runner. Coyote contains 8 sub-projects, varying from 56,000 to 1,550,000 LOC. Road Runner has 4 sub-projects, ranging from 270,000 to 6,650,000 LOC. We report on analysis scans from spring 2017 to December 2018, except for 6 sub-projects of Coyote, three of which have been using Dynamite since winter 2018, and three others, since winter 2017.

3 ANALYSIS TOOLS AT ACME

In the past few years, Acme has emphasized ensuring the quality and security of its software through the use of analysis tools. Individual developers and projects can use their own analysis tools independently, but global efforts across the company have recently resulted in the deployment of common analysis tools and platforms over most major projects. To address **RQ1**, we discuss which tools are used at Acme, how Acme integrates them in its development process, and the types of issues that they find.

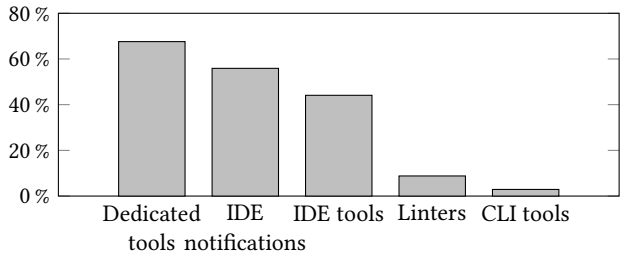


Figure 2: Types of analysis tools used at Acme. (Q4)

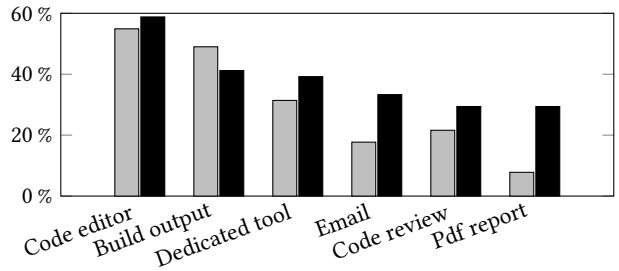


Figure 3: Reporting locations of current analysis tools (gray) and ideal reporting locations (black). (Q10–Q11)

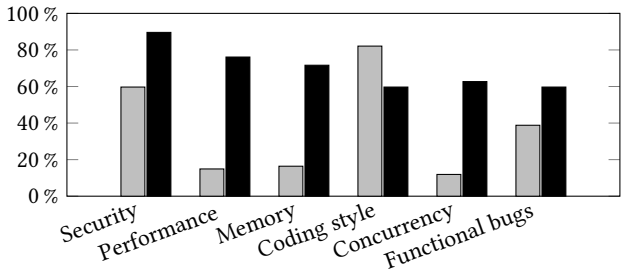


Figure 4: Warning types reported by the analysis tools (gray) and warning types the developers want (black). (Q7–Q8)

3.1 Analysis Tools

Acme developers report using a total of 17 different analysis tools that we group by usage types in Figure 2 (**Q4**). *IDE notifications* designate analyses run by the developers' Integrated Development Environment (IDE) (e.g., uninitialized variables). *IDE tools* refer to analysis tools integrated in the IDE (e.g., FindBugs). *Dedicated tools* provide interfaces that are separate from the developer's coding environment (e.g., Fortify). *CLI tools* provide a Command-Line Interface (CLI). We can see that Acme uses a wide variety of analysis tools. Among the survey participants, 67.6% use dedicated tools, which conforms with Acme's policy of using such tools in their projects. Participants also receive analysis information from IDE notifications (55.9%) and IDE tools (44.1%). Overall, 36.8% of the participants use only one analysis tool, and 48.5% of the participants use only one type of analysis tools.

3.2 Integration in the Development Process

With **Q5**, we observe that the use of analysis tools is spread over the software development lifecycle: 55.9% of the participants run their analysis tools at coding time, 52.9% during nightly builds, 29.4% at commit time, and 17.6% at major project milestones. We attribute this behaviour to the use of different types of analysis tools—IDE notifications run at coding time, while longer running tools are not usually able to do so.

Once the analysis tools have run, they display warnings in various places, as shown in grey in Figure 3 (**Q10**). Reports in the code editor, build output, and dedicated tools are expected from the tool types that are most used at Acme. When asked which reporting media they would like to use (**Q11**, black bars in Figure 3), participants confirmed wanting to use their current reporting platforms. However, alternative means were requested to a much higher extent: PDF reports were requested 3.75× more than currently used and email reports were requested 1.89× more, and the code review platform was requested 1.36× more. We attribute this higher demand to the ability of those media to aggregate results from multiple analysis tools in one place, which makes it easier to have an overview of the analysis results. Although we cannot confirm this claim with our current dataset, it is partially supported by the responses to **Q12** where 5.5× more developers indicated that they would prefer having the results of multiple analysis tools into one reporting place rather than in different ones (74.5% against 15.7%).

The survey responses show that 82.4% of the participants typically fix analysis warnings themselves (**Q9**). Once the warnings are fixed, they are reviewed by colleagues (79.5%), managers (15.9%), or dedicated teams (9.1%). Out of all fixes, 9.1% go unverified (**Q29**)

According to 47.1% of the participants, analysis tools used at Acme are configured by a dedicated team. However, 36.8% wrote that they configured some of their analysis tools themselves, and 16.2% that some of their tools run on default settings **Q6**. We see that a high number of developers set up their own tools themselves, which we attribute to the use of tools not proposed by the global company effort. This behaviour, along with the responses to **Q17** where 78.3% of the developers said that they use analysis tools because it "helps me code better" against only 30.4% because of "company policy", suggests that Acme generally encourages the use of analysis tools and spreads awareness among its developers about the importance of fixing bugs and security vulnerabilities.

3.3 Analysis Warnings

In their responses to **Q7**, participants indicated the vulnerability types that are reported by analysis tools in their projects the most (grey bars in Figure 4). The warnings that are reported the most are coding style-related (according to 82.1% of the participants), followed by security vulnerabilities (59.7%), and functional bugs (38.8%). We calculated the conditional probabilities of warnings being of a certain type given the tool type (Table 1). We see that for each tool type, the warning types most likely to be reported are security vulnerabilities, coding style issues, and functional bugs. This matches the distribution of warnings that code developers listed as most often reported. Dedicated tools and CLI tools are the

Table 1: Conditional probabilities of a warning being of a certain type (S = security, P = performance, M = memory, CS = coding style, C = concurrency, FB = functional bugs) given the tool type. (Q7)

	S	P	M	CS	C	FB
Dedicated tools	0.31	0.06	0.08	0.34	0.04	0.16
IDE notifications	0.26	0.06	0.04	0.4	0.06	0.19
IDE tools	0.27	0.05	0.06	0.31	0.09	0.21
Linters	0.27	0.09	0	0.36	0	0.27
CLI tools	0.50	0	0	0.25	0	0.25

most likely tools to report security vulnerabilities, and linters, IDE notifications and tools are more likely to report coding style issues.

When asking developers which types of warnings they would like analysis tools to report (**Q8**, black bars in Figure 4), we observe that the warning types are more distributed. While security vulnerabilities and functional bugs are still high (89.6% and 59.7%, respectively), the number of participants asking for other types of warnings (performance, memory, and concurrency) is higher than the number of participants getting access to such warnings by factors of 4.4× to 5.3×. On the other hand, participants wish to see less of the most frequently reported class of warnings: coding style.

3.4 Summary (RQ1)

Acme has a thorough approach to using static analysis tools: they use multiple analysis tools (**Q4**) at all stages of the software development process (**Q5**), regularly fix analysis warnings (**Q9**) and manually review the fixes (**Q29**). Acme put in a lot of effort in raising awareness about the use of analysis tools, which is acted upon by the developers (**Q17**).

Acme developers are more exposed to dedicated tools, and IDE tools and notifications. Warnings are reported in various places across the working tools (**Q10**), but developers indicate that they would prefer a common interface for all analysis warnings (**Q11**).

Analysis tools at Acme are more likely to find certain types of warnings: security vulnerabilities, coding style, and functional bugs (**Q7**). Developers would additionally like to receive more information about the performance, memory, and concurrency bugs in their code (**Q8**). Further analysis tools focusing on those warnings would make good additions to the analysis system.

Overall, Acme strives to include static analysis tools to bring their code to better standards and exposes its developers to large system of analysis tools, making a good case study for evaluating developer behaviour and motivation towards static analysis tools.

4 CONTEXTS FOR USING ANALYSIS TOOLS

We detail when and how Acme developers use analysis tools, and expand on the reasons that make them use those tools (**RQ2**). We focus on the tool types that developers use the most: IDE tools and notifications, and dedicated tools.

4.1 Developer Workflow

Although the usage of analysis tools is distributed evenly during the day (morning 11.4%, afternoon 13.6%, and evening 11.4%), the largest group of developers (22.7%) use analysis tools in their spare time (**Q19**). Analysis tools are typically used frequently in the work week: 75.6% of the participants say they use them multiple times in a week, 24.5% of which use them more than once a day (**Q18**). This usage pattern indicates that working with analysis tools is not a large task that requires a developer to block a part of their schedule, but instead a set of short tasks that can be interrupted and resumed later. This observation is further supported by responses to **Q13** and **Q22** where we see that the median time for one working session with an analysis tool is 10–30 minutes while the median time for fixing one warning is between an hour and a day. We can thus infer that in many cases, developers spread their treatment of an analysis warning over multiple working sessions.

Table 2: Conditional probabilities of the length of a working session given the tool type. (Q22)

	< 10 min	10–30 min	> 30 min	hrs
Dedicated tools	0.16	0.52	0.13	0.1
IDE notifications	0.31	0.35	0.12	0.12
IDE tools	0.40	0.44	0.04	0.04
Linters	0.75	0	0	0.25
CLI tools	0	1	0	0

Table 3: Conditional probabilities of a fixing a warning in a certain time given the tool type. (Q13)

	mins	< 1 hr	< 1 d	< 1 wk	< 1 Mo
Dedicated tools	0.15	0.15	0.38	0.21	0.06
IDE notifications	0.19	0.26	0.35	0.10	0.30
IDE tools	0.23	0.19	0.42	0.12	0
Linters	0.50	0.25	0	0	0
CLI tools	1	0	0	0	0

The length of a working session with IDE notifications, IDE tools, and dedicated tools mainly vary between a few minutes to 30 minutes (Table 2). While for IDE notifications and tools, the session length is evenly distributed between *< 10 min* and *10–30 min*, dedicated tools clearly lean towards the longer end of the spectrum. The typical fix times for one warning are shown in Table 3. We find the same trend over the time span of minutes to under a week: with IDE notifications and tools, a warning is fixed in around a shorter time (between an hour and a day) than with dedicated tools (around a day). This trend is explained by the fact that analyses running in the IDE must be able to yield results in a matter of seconds, which restricts them to fast and simple-to-compute checks. Their warnings are thus relatively easier to fix. We see that for the individual tool types, working sessions are typically shorter than the time to fix a warning, the only exception being CLI tools.

4.2 Developer Motivation

Having determined how developers work with analysis tools, we now explore the reasons why they interact with the analysis tools in such a way, and what their goals are when using them.

Table 4: Developer goals when opening analysis tools. (Q23)

	Strategy	% of Devs
O1	Fix all warnings	36.4%
O2	Fix warnings in a given time	31.8%
O3	Consult warning list	31.8%
O4	Fix a set number of warnings	9.1%
O5	Fix warnings up to a certain standard	4.6%

Table 5: Reasons for closing analysis tools. (Q24)

	Strategy	% of Devs
C1	Finished fixing everything	45.5%
C2	Professional obligation	25%
C3	Wait for the analysis tool to update	18.2%
C4	Office distraction	13.6%
C5	Never close the tool	13.6%
C6	Cannot fix an issue	9.1%

In Table 4, we see that developers open an analysis tool mostly to fix warnings with the variation of how many warnings they aim to fix (**Q23**). Conditional probabilities show that a relationship exists between tool types and fixing goals. When using dedicated tools, participants mostly aim at fixing as many warnings as possible in a given time ($Pr = 0.31$), which is a sensible strategy when dealing with complex warnings. With all other tools, participants mainly aim to fix all warnings when they open the tool. Consulting the list of warnings is a frequent reason why developers open the tools for all tool types ($0.24 \leq Pr \leq 0.28$). Table 5 details why developers close an analysis tool (**Q24**). The main reason is that they finished fixing all warnings, which we attribute to the use of lighter analysis tools that yield easier-to-fix warnings. The second cause is time limit. The third one is that they wait for a tool update (complex analyses can take minutes to hours to process an update). A minor reason for which developers close the tool is that they cannot fix a warning. This issue is likely encountered when dealing with complex warnings that are not properly explained by the tool.

Table 6 shows that, regardless of the reason why the tool was opened, a popular reason for closing it is that all warnings were fixed (C1). However, when developers open the tool with a certain limit in mind (time or number of warnings), fixing all warnings is not the main closing reason. Developers are also likely to close the tool due to professional obligations (e.g., a meeting) or waiting for a tool update. We also see that when developers open the tool with the intention to fix all warnings, they only manage to reach their goal 45% of the time. Otherwise, they are either stuck on a warning or waiting for a tool update. This suggests that when developers do not have time constraints, they have a fair chance to eventually run into warnings that they cannot fix in one working session.

4.3 Summary (RQ2)

Acme developers use analysis tools mostly in their spare time (**Q19**). For them, fixing warnings is a continuous task spread over short working sessions (**Q18**).

Table 6: Conditional probabilities of reasons for closing an analysis tool given the reason for opening it. The legends for Ox and Cx are found in Table 4 and Table 5. (Q23–Q24)

	C1	C2	C3	C4	C6
O1	0.45	0.05	0.15	0.05	0.15
O2	0.22	0.33	0.17	0.17	0
O3	0.35	0.25	0.15	0.10	0.05
O4	0.20	0.20	0.20	0.20	0

Table 7: Developer strategies to prioritize which warnings to address first. (Q34)

Strategy	% of Devs
Prioritize warnings affecting the developer's code	46.3%
Prioritize warnings with the most impact	43.9%
Prioritize warnings the developer can fix	31.7%
Follow the order of the warning list	31.7%

Depending on the type of analysis tools they use, developers have different working times with the tools. Warnings reported by IDE notifications and tools allow developers to spend less time fixing warnings, and to have shorter working sessions with the tools. More complex warnings produced by dedicated tools have longer fix times, and cause developers to work with the tools for a longer period of time (**Q13**, **Q22**).

In turn, this aspect generates different motivations for starting and stopping to use analysis tools. While most developers open analysis tools to fix warnings, they choose to fix different sets of warnings depending on their available time (**Q23**). This constraint introduces different interaction experiences with the tools, namely how to support developers in choosing which warnings to fix in the given time, which we explore in the following section.

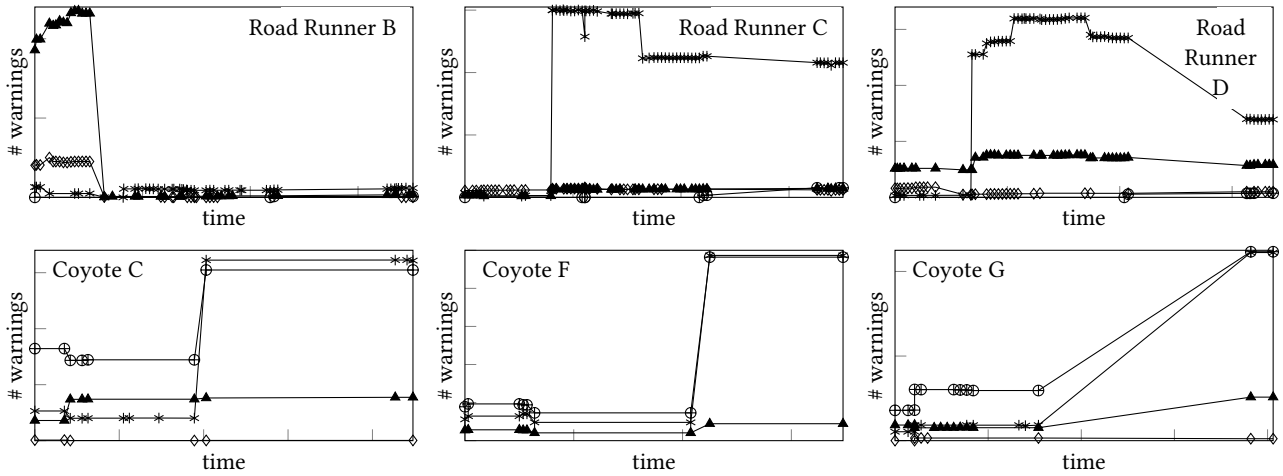
Apart from time constraints, other causes for developers stopping to use analysis tools are explainability issues for complex warnings, and the long time taken to update analysis results (**Q24**). Both issues should be considered when designing an analysis tool.

5 STRATEGIES FOR FIXING ANALYSIS WARNINGS

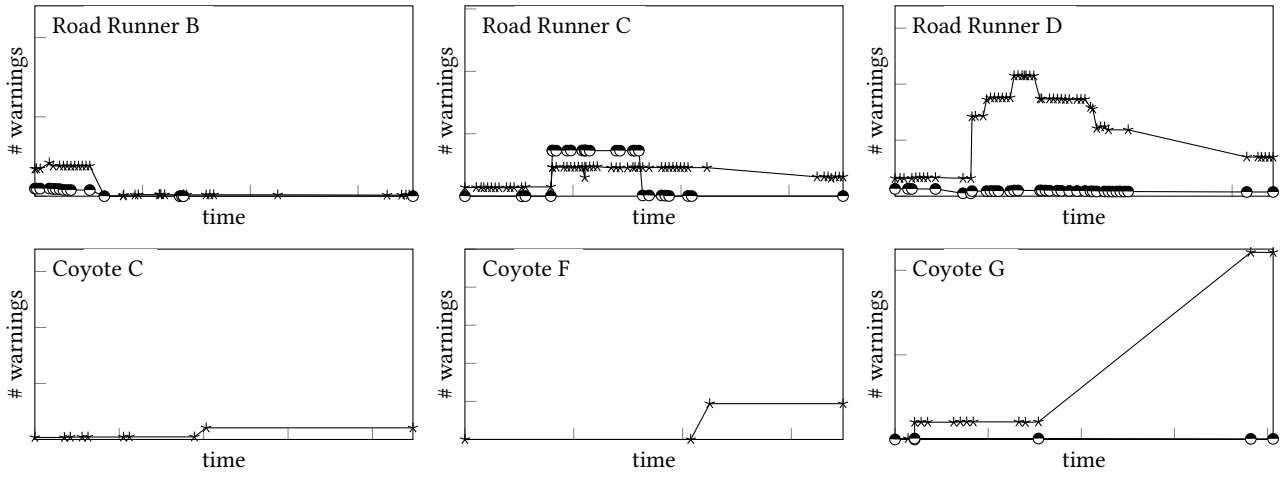
In this section, we explore developer behaviour when fixing warnings, in particular how they prioritize which ones to investigate first, what they do with warnings that they do not understand, and how they collaborate with their colleagues to fix them (**RQ3**).

5.1 Prioritizing Warnings

Before they start fixing warnings, developers must first choose which ones they should fix. To help them decide, analysis tools can provide them with additional information such as warning type (e.g., SQL injection), code location, or severity. Table 7 shows the four main strategies that the survey participants have to choose which warnings to investigate (**Q24**). The most popular is to prioritize the warnings by impact, which aligns with Acme's policies to remove all of the most severe warnings. A developer will also preferentially



(a) Severity (—◇— high, —▲— medium, —✳— low, —⊕— info).



(b) Labeled confidence (—✳— false positive, —⊖— true positive).

Figure 5: Number of warnings for three sub-projects of Road Runner, and three sub-projects of Coyote. For anonymization reasons, we do not disclose the axis labels. All axes are in linear scale, the y-axes all start at 0. The top / bottom pairs for each project have the same maximum value on the y-axis.

work on warnings that impact their own code or that they know how to fix, because they have the necessary knowledge to do so. The last strategy is to go from the top down in the warning list, which is a sensible methodology for simple lists of warnings that are all fixable within one working session. Such strategy is also useful for longer, more complex lists that the tool already sorts by importance, which is often the case in dedicated tools.

To gain a deeper understanding of which warnings are fixed first, we studied the analysis reports of two projects at Acme: Coyote and Road Runner. Figure 5 details the number of warnings found for six of their sub-projects, grouped by severity (top) and labeled confidence (bottom), over the time span of a few months for Coyote to nearly two years for Road Runner. Except for Coyote G, only a fraction of the warnings are labeled by developers as true or false positives. We also observe that the variations of the number of labeled warnings follows the variations of the general number of warnings, suggesting that developers actively handle new warnings but usually only look at a fraction of the total number of warnings, or do not often label warnings.

We see that developers tend to keep the number of warnings with a high severity at a minimum: the plot consistently remains close to 0, confirming our survey results (**Q24**: developers tend to fix warnings with the most impact first). This is supported by Table 8: the probability of a high severity warning to be in the *to verify* list is very low compared to other types of warnings, and the high severity warnings that remain are most often false positives. Confirmed true positives are handled similarly: they are kept to a minimum and eventually removed, (e.g., Road Runner C).

Most projects also have a small number of low severity warnings, which we attribute to the relative ease of fixing such warnings,

Table 8: Conditional probabilities that a warning is marked with a certain status, given its severity.

	False Positive	True Positive	To Verify
High	0.92	0.03	0.04
Medium	0.15	0.03	0.83
Low	0.05	0.01	0.93
Info	0	0	1

Table 9: Developer strategies to detect false positives. (Q33)

Strategy	% of Devs
Categories of issues are known false positives	43.9%
Code constructs are not handled by the analysis	39.0%
Warning witness is not executable	31.7%
Code locations are never executed	22.0%
Conditions along the warning are never true	12.2%

matching the developer strategy of fixing what they know they can fix. For example, *unchecked return value*, and *null pointer dereference*, likely to be classified with a low severity with a probability of 1, are simple to fix, and the fix can be easily verified.

In the longer running Road Runner projects, we also observe that the number of warnings regularly plummets, which (knowing Acme's release schedule) we suppose with fair confidence corresponds to compliance tests before major product releases or milestones. Outside of those times, the number of warnings decreases slowly due to continuous work done by developers on their spare time, as we have discussed in Section 4.1.

5.2 Detecting False Positives

To help developers decide whether a warning is a false positive or a real issue, analysis tools often provide them with additional metrics. For example, the main screen of Dynamite allows developers to filter analysis warnings by warning type (e.g., cross-site scripting), code location, severity, and other information the developer can edit (e.g., labeled confidence).

Table 9 shows five main strategies that participants use to evaluate if a warning is a false positive (**Q33**). The main strategy is looking at the warning type. This strategy can be a quick way of triaging through warnings, because certain warning types are more often labeled as false positives than others. For example, *memory leak* or *use of uninitialized variable* are very likely to be marked as false positives ($Pr = 0.8$ and $Pr = 0.71$ respectively). However, this strategy can be misleading: without investigating each warning in detail, true positives may be overlooked.

The second most popular strategy is related to misinterpretations of some code constructs by the analysis (for example, the behaviour of specific libraries or objects). Such constructs seem to be known by developers, and are used to differentiate true from false positives. Another 43.9% of the participants said that they recognize false positives, because the warning witness is wrong, meaning that the analysis' interpretation of the code's runtime behaviour is faulty. This situation requires from the developer to investigate the warning in detail, which is a very accurate, but time-consuming strategy. On average, participants investigate 65.1% of the warnings in detail (min = 20%, max = 100%, $\sigma = 26.1$) (**Q32**). The last strategy is to mark as false positives warnings that go through areas of the code that are never executed. While this strategy helps remove false positives, it is dangerous to keep vulnerable non-executed code in the codebase, as it could be exploited in the future.

Table 10: Reasons why warnings are difficult to understand (Q36)

Reason	% of Devs
Unfamiliar with the issue	48.8%
Explanation given by the analysis tool is unclear	48.8%
Span over too much of the code base	31.7%
The code base is unclear	4.9%

Table 11: Developer actions for warnings that they do not understand. (Q37)

Action	% of Devs	Behaviour type
Leave for later	56.1%	Neutral
Ask for help	51.2%	Good
Ignore	14.6%	Bad
Research and fix	7.3%	Good
Suppress	7.3%	Bad
Escalate	2.4%	Good

5.3 Understanding Warnings

To understand if a warning is a true positive, if they should prioritize it, and how they could fix it, developers have to gain an understanding of the warning. We have previously discussed that developers often use heuristics over easily accessible data to make a decision, because they cannot spend time investigating all warnings. Therefore, the ability of the analysis tool to explain the warning and showcase relevant data is key to supporting its users.

On average, participants understand 36.1% of the warnings they investigate (min = 0%, max = 80%, $\sigma = 32.6$) (**Q35**). To explain this low number, we asked participants for the reasons why warnings can be difficult to understand, which we detail in Table 10 (**Q36**). Three major reasons stand out. First on the list is that the warning is new to the developer, so they need to learn a lot: what the warning means, how it applies to their code, and how to safely fix it in the context of their code. Another reason is that the tool's explanation is unclear. While some tools simply give a generic description of the warning type, others, such as Dynamite, provide more detailed information, yet it is still difficult to completely explain to developers how the analysis reasons about the warning, especially when the warning is complex and spans over a wide part of the code base, which is the third reason for not understanding warnings.

Table 11 details the treatment of warnings that developers do not understand (**Q37**). Overall, we see three main types of behaviour appear: neutral, positive, and negative. A majority of the developers (56.1%) adopt the neutral behaviour of leaving the warnings for

Table 12: Reasons why developers ask for help. (Q39)

Reason	% of Devs
Others have experience with the code base	46.3%
Others have experience with the type of issue	39%
Others have experience with the analysis tool	31.7%
The developer does not ask for help	14.6%

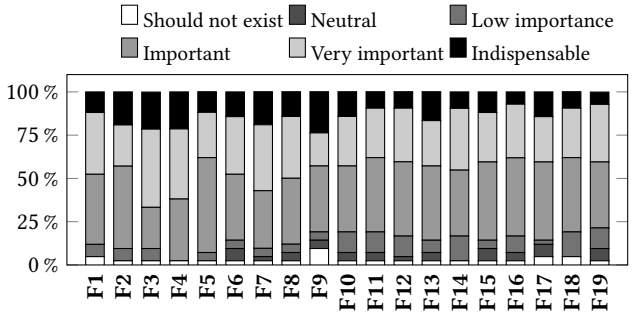
later, which is supported by Figure 5: the number of warnings tend to remain relatively constant, until right before a major release or milestone. More negative solutions are to ignore or suppress the problematic warning. 14.6% and 7.3% of the developers admit to using them, respectively, which should be discouraged. Other participants opt for positive actions and spend more time asking for help, escalating, or researching the warning. On average, participants ask for their colleagues' help for 27.8% of the warnings in an analysis report (min = 0%, max = 70%, σ = 42.2) (**Q38**).

When developers ask for help (**Q39**), they are interested in the three particular aspects that we discussed in **Q36**: the issue, the code base, and the analysis tool (in particular, what the tool means when explaining the warning), as seen in Table 12. The first three aspects confirm our observations from **Q36**. In particular, with the second one, we see that of the warnings asked about, 46.3% are due to code base issues, while only 4.9% of the participants find warnings confusing due to lack of understanding of the relevant codebase. We infer that developers rarely ask about confusing warnings, and that a large fraction of the ones they ask about are due to codebase clarity issues. This confirms the need for better warning explanations, especially with respect to information about the issue and the analysis tool, which is less at hand to the developers than codebase information they can ask colleagues about. Lastly, 14.6% of the participants do not ask for help. We suppose that this behaviour could be caused by time constraints, discouragement of working on a warning for too long, or the social consequences of admitting that they do not understand the warning.

5.4 Summary (RQ3)

Acme developers primarily use severity to prioritize warnings (**Q24**, Figure 5). They also use internal knowledge and heuristics to distinguish false positives from true positives (**Q33**). Integrating such knowledge in the analysis would help bridge the gap between how the analysis understands the code base and how the developer does.

Properly understanding analysis warnings is important, and giving the developer the proper knowledge to make a call on what is a true/false positive, what to fix first, and how to fix it should be a priority when designing an analysis tool. Warnings that are too hard to understand may discourage a developer who keeps trying to learn more about the codebase, the analysis tool, and the warning, and usually results in a negative treatment of the warning (**Q35**–**Q37**) or strategies that are quick but inaccurate (**Q33**, **Q37**). Facilitating explainability is not only restricted to finding better explanations than the ones already provided by the tool. Better explainability can also be achieved through better visualizations of those explanations, or encouraging better solving practices such as asking questions and collaborating with colleagues (**Q38**–**Q39**).



F1 Responsiveness of the analysis
F2 Responsiveness of the UI
F3 Explain how a bug works and what it is about
F4 Explain how severely the bug can affect the code
F5 Explain how the bug can be executed in the code
F6 Explain how the warning is related to other warnings
F7 Explain how the bug can be fixed, on a high level
F8 Explain how the bug can be fixed in the code
F9 Quick fixes
F10 Visualisation of the individual warnings in the code
F11 Dashboard of the warnings
F12 Search and sort through warnings
F13 Select which bugs to fix first (prioritization)
F14 Keep a list of the developers' bugs
F15 Tracking progress (clear view of what was achieved so far)
F16 Configurability of the analysis
F17 Provide feedback to the tool
F18 Customization of the analysis rules
F19 Collaboration options (with colleagues)

Figure 6: Ranking the importance of tool features. (Q30)

6 TOOL FEATURES

In light of the developers' motivations identified in **RQ1**–**RQ3**, we discuss features for the user interface (UI) of an analysis tool (**RQ4**).

In **Q26**–**Q28**, we asked participants which UI features they most often look at when opening, using, and closing an analysis tool. Developer attention is most attracted to the dashboard (a high-level summary of the project health) when opening the tool (47.7%) and the warning list (the detailed list of warnings, supplemented with bug information, severity, code location, etc.) when they close it (59.1%). The warning list is central in all cases, in particular, when using the tool (68.2%), showing the importance of the information conveyed in this list: the issues, where they are located, and how much still needs to be achieved to meet the company's standards.

We identified a set of 19 UI features from commercial (e.g., Checkmarx [8], CodeSonar [13], etc.), academic (e.g., FlowDroid [1], Cheetah [19], etc.), and open-source (e.g., FindBugs [23], IntelliJ's Code Inspection [15], etc.) analysis tools, and from past work on their usability [3, 9, 11, 17], and asked participants to rank their importance between six categories: *should not exist*, *neutral*, *low importance*, *important*, *very important*, and *indispensable* (**Q30**). In the remainder of this paper, we will refer to those features as **F1**–**F19**, all listed in Figure 6. **F1**–**F2** focus on the responsiveness of the tool, **F3**–**F6** address different aspects of explainability, **F7**–**F9** deal with fixing

warnings, **F10**–**F12** aim at visualizing the project's health, **F13**–**F15** help keep track of individual warnings, **F16**–**F18** concern analysis configuration and feedback, and **F19** focuses on collaboration.

Figure 6 shows that the most popular features are **F3** (explain a bug), **F4** (bug severity), **F7** (how the bug can be fixed), and **F9** (quick fixes). The first three have a total of 28, 26, and 24 developers respectively marking them as *very important* or higher, with an overall average of approximately 19 developers. **F9** had 10 developers marking it as *indispensable*, against an average of approximately 6. The popularity of **F3** and **F4** echoes our findings from **RQ3**. Since developers are more interested in severity and understanding the warning, those two features are most important to them. **F7**, which explains how the warning can be fixed on a high level, is highly appreciated. However, **F8**, which does the same but gives more specific recommendations with regards to the code base, receives less support. Although we cannot be certain, it is possible that a fix generated by the analysis would add to what developers currently have to understand, and the risk of introducing more potential bugs in the code base is too high. Those reasons would also explain the low ratings of **F9**, which has the highest score for *should not exist* with 4 developers compared to an average of approximately 1. **F9** is thus among the most popular features (supposedly for its gain of time), and among the least popular ones.

Features such as collaboration (**F19**), customization of the analysis rules (**F18**), and visualization features (**F10**–**F11**), which we have identified in **RQ3** as ones that can potentially enhance the usability experience of the developers, have received the lowest ratings by a margin of four developers or fewer when compared to the average. In particular, **F10** and **F11** are often used by the developers (**Q26**–**Q28**). With our current data, it is difficult to say whether those lower ratings are caused by the developers disliking such features in their current analysis tools, them disliking the general idea behind those features, or, if having not experienced those features yet, they are wary of them. On the other hand, all features in the survey were deemed *important* or more by at least 75% of the participants, so when designing an analysis tool, even the least popular features would be worth including.

Summary (RQ4)

When first opening an analysis tool, Acme developers often look at a dashboard to get a general overview of the warnings in the code base (**Q26**). When closing the tool and while using it, they most often look at the warnings list, which seems to be their primary working tool (**Q27**–**Q28**).

Features provided by this list are among the most important features marked in (**Q30**). While the features we have identified as potentially useful to have in an analysis tool received weaker ratings, all 19 features received positive ratings, and should be considered when designing an analysis tool. One should keep in mind that the integration of certain novel, or less popular features should be designed with care, keeping in mind the developer's goals and motivations to best assist them.

7 DISCUSSION

In this section, we discuss the outcomes of the study presented in Sections 3–6, and identify recommendations for building and using analysis tools for software companies such as Acme.

Analysis platform: While it is generally recommended to use multiple static analysis tools in conjunction and recoup their warnings together, we also recommend adding different types of tools (e.g., dynamic analyzers, profilers) to cover aspects that are less reported on by static analysis tools (performance, memory, or concurrency information) (Section 3.3). When using different tools, we recommend the use of one single reporting platform to handle all warnings. On top of a general overview of the project's health, such a platform can also provide a unified reporting format, which would reduce developer work in consulting multiple interfaces and learning how the different tools work (Section 5.3), and provide more common grounds for sharing information and encouraging collaboration (Section 5.4). In addition, the platform can work as a central hub to configure analyses for different projects, allowing the possibility of a common format for sharing analysis rules across different projects (Section 5.2), and reducing the individual developer's configuration time (Section 3.2). The choice of the platform format should match the tools developers are used to, which, in Acme's case, is IDE or dedicated tools (Section 3.1).

Developer / tool interactions: To ease the developer's work with analysis tools and make their working sessions shorter (Section 4.1), analysis tools should be designed with the developer's main tasks in mind: prioritizing warnings, understanding them, and developing a fix. The ease of navigating those tasks depends on the tool's ability to explain warnings and bridge the gap between its and the developer's understanding of the warning in the context of the code (Section 5.3). Examples of features in this direction are: allowing developer to integrate their knowledge in the analysis, giving more granular explanations on why the analysis reports certain warnings, presenting a warning in a way that would require developers to spend less time researching it, or providing better sharing and collaboration support to leverage the knowledge of various developers and build a common repository of known warnings (Section 5.4).

Another gap that should be bridged is the interpretation of company goals into concrete code metrics. The limited working times and company policies influence developer working strategies (Section 4.2). For example, in the case of Dynamite, Acme developers use severity as their primary metric. While this metric allows developers to fix many severe issues, a fair share of the high severity warnings are false positives, and a large part of the other warnings are left unchecked (Section 5.1). When designing an analysis tool, we recommend including more granular metrics to help prioritize and identify true from false positives more efficiently. Such information could be mined by the analysis tool from already existing data: warnings marked as true / false positives, and then be reviewed and shared across different projects.

Tool features: The quality of the fixes depends on the developer's engagement with the tool and their motivation to put in the extra effort in a difficult situation. At Acme, developer motivation revolves around fixing a certain number of warnings (until a fixed time, or certain standards are met) (Section 4.2), which in

some cases, can lead to negative behaviour such as ignoring or dismissing warnings (Section 5.3), especially when they are hard to understand. It is thus important for an analysis tool to provide an engaging and persuasive UI that adapts to the developer's interests at a given moment, suggests appropriate actions, and encourages good behaviour. Such features (e.g., **F9**) can be ill-received if badly designed, especially if they are new or give risky advice (Section 6). Therefore, tool features should be carefully designed to maximize usefulness while minimizing the risk of leading the developer towards a dangerous action. An example for such a feature is drawing developer attention to different types of warnings depending on how much time they have, and the history of the warnings they fixed (**F13**, Section 5.1). Grouping warnings to minimize how much developers must learn would minimize the risks of demotivation (**F12**, Section 5.4). Another example would be to propose contacting other developers who have fixed similar warnings when a developer is stuck (**F19**, Section 5.3).

Company policies: The success of analysis tools directly depends on the company's backing. Encouraging the use of analysis tools, spreading awareness, defining quality standards, regularly keeping up to those standards, and enforcing a strong reviewing policy all contribute to the widespread use of analysis tools at a company (Sections 3, 5.1). Such policies should be maintained and developed, with for example, initiatives and trainings to sensitize developers towards good behaviour when it comes to choosing what to do when they do not understand a warning, or to encourage them to ask and share knowledge on such warnings (Section 5.3).

8 THREATS TO VALIDITY

Our study is limited to Acme and therefore does not necessarily generalize to every software company. However, the participants showed a large diversity in experience and programming languages, and have years of experience working with static analysis tools at Acme (**RQ1-2**). Thus, the results of our survey reasonably generalize to similar companies who use analysis tools internally.

The formulation of the survey questions could also be subject to misunderstandings. To minimize such errors, we ran a pilot survey with five developers from Acme, and during the data extraction process, we did not find any responses that we could interpret as a response to a misunderstood question. Another threat to validity is the subjective interpretation of the free-text (**Q15**, **Q31**, **Q40**) and "Others" survey responses when we re-classified them in different categories. In the end, we did not use the free-text questions in this paper, and verified the classification of the "Others" responses with two raters, with 100% agreement. The survey questions and anonymized responses are made available online [2].

While the two projects which reports we accessed do not fully represent all of Acme's projects, they are major projects of the company, and are contributed to regularly. We included all of their sub-projects in our study, which offers diversity in terms of project type, target platforms, and exposure time to Dynamite.

9 RELATED WORK

Many studies evaluating the usability issues of analysis tools were conducted in the past. Bessey et al. [6] report on the early issues of Coverity. Ayewah et al. [4] observe student developers use FindBugs. Johnson et al. [17] present interviews for 20 developers on their experience of various analysis tools, focusing on tool features. Lewis et al. [18] deploy two bug prediction algorithms at Google and interview developers. All those studies reported usability issues such as waiting times, and more importantly, common explainability issues that are, as we have shown, shared by Acme developers: understanding the warning, evaluating whether it is a true / false positive, and how to fix it.

Other studies focus on developer strategies and motivations. Ayewah et al. [3] survey Google developers for fixing strategies with FindBugs. In contrast, our study reports on a more realistic setting for industry where developers use various types of analysis tools in conjunction. Christakis et al. [9] survey developers and study live site incidents at Microsoft to extract recommended features for analysis tools. On the other hand, we consider developer motivation and strategies to determine useful features.

Witschey et al. [27] and Xiao et al. [28] show that company policies and standards positively influence tool adoption, which supports our recommendations.

While many approaches have addressed usability issues of analysis tools, one in particular stands out. Nanda et al. [20] build Khasiana, an analysis hub at IBM that provides a unified analysis center and useful features to help developers understand warnings. Approaches that aim to adapt analysis tools for particular users in a particular situation have been proposed in the past. Bodden et al. [7], Johnson et al. [16] and Nguyen Quang Do et al. [11] propose design ideas for customizing reporting tools and analysis systems.

Persuasive technologies have been successfully used to engage users in performing repetitive, intellectually challenging tasks (e.g., gamification in software engineering [10, 14, 25]), or to encourage them towards good habits and behaviour (e.g., healthcare [22]). The key to successfully designing such tools revolves around identifying user types and their needs [26], and to carefully design personalized interfaces that tightly cater to those needs [21] but still clearly enable users to reach their original goals [5, 12]. To encourage good developer behaviour, it would be interesting to apply persuasive techniques to the domain of static analysis tools.

10 CONCLUSION

Through a developer survey an analysis of two projects at Acme, we have drawn a picture of how static analysis tools are used in industry. We saw that Acme widely uses analysis tools at all points of the software development process, and that its developers heavily use analysis tools in their spare time to fix warnings. They have developed internal knowledge and strategies to prioritize warnings, determine if they are true or false positives, and decide how to handle the warnings. Some of those strategies are to be encouraged, such as collaborating with colleagues, and others are to be discouraged, such as suppressing warnings one doesn't understand. Based on this knowledge, we have identified recommendations for designing and using analysis tools, which we classify in four categories: the description of a central analysis platform, the needs for improving the interactions between developers and analysis tools, desirable features for the tool's interface, and company policies to encourage the use of analysis tools.

REFERENCES

[1] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. 2014. FlowDroid: Precise Context, Flow, Field, Object-sensitive and Lifecycle-aware Taint Analysis for Android Apps. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '14)*. ACM, New York, NY, USA, 259–269. https://doi.org/10.1145/2594291.2594299

[2] Anonymous authors. 2019. https://sideresearch.wordpress.com.

[3] N. Ayewah, D. Hovemeyer, J. D. Morgenthaler, J. Penix, and W. Pugh. 2008. Using Static Analysis to Find Bugs. *IEEE Software* 25, 5 (Sep. 2008), 22–29. https://doi.org/10.1109/MS.2008.130

[4] Nathaniel Ayewah and William Pugh. 2008. A Report on a Survey and Study of Static Analysis Users. In *Proceedings of the 2008 Workshop on Defects in Large Software Systems (DEFECTS '08)*. ACM, New York, NY, USA, 1–5. https://doi.org/10.1145/1390817.1390819

[5] K. Berkling and C. Thomas. 2013. Gamification of a Software Engineering course and a detailed analysis of the factors that lead to it's failure. In *2013 International Conference on Interactive Collaborative Learning (ICL)*. 525–530. https://doi.org/10.1109/ICL.2013.6644642

[6] Al Bessey, Ken Block, Ben Chelf, Andy Chou, Bryan Fulton, Seth Hallem, Charles Henri-Gros, Asya Kamsky, Scott McPeak, and Dawson Engler. 2010. A Few Billion Lines of Code Later: Using Static Analysis to Find Bugs in the Real World. *Commun. ACM* 53, 2 (Feb. 2010), 66–75. https://doi.org/10.1145/1646353.1646374

[7] Eric Bodden. 2018. Self-adaptive Static Analysis. In *Proceedings of the 40th International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER '18)*. ACM, New York, NY, USA, 45–48. https://doi.org/10.1145/3183399.3183401

[8] Checkmarx. 2019. Checkmarx home page. https://www.checkmarx.com/.

[9] Maria Christakis and Christian Bird. 2016. What Developers Want and Need from Program Analysis: An Empirical Study. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering (ASE 2016)*. ACM, New York, NY, USA, 332–343. https://doi.org/10.1145/2970276.2970347

[10] M. R. d. A. Souza, K. F. Constantino, L. F. Veado, and E. M. L. Figueiredo. 2017. Gamification in Software Engineering Education: An Empirical Study. In *2017 IEEE 30th Conference on Software Engineering Education and Training (CSEE T)*. 276–284. https://doi.org/10.1109/CSEET.2017.51

[11] Lisa Nguyen Quang Do and Eric Bodden. 2018. Gamifying Static Analysis. In *Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2018, Lake Buena Vista, FL, USA, November 04-09, 2018*. ACM, New York, NY, USA, 714–718. https://doi.org/10.1145/3236024.3264830

[12] Gartner. 2018. Gartner Says by 2014, 80 Percent of Current Gamified Applications Will Fail to Meet Business Objectives Primarily Due to Poor Design. https://www.gartner.com/newsroom/id/2251015.

[13] Grammatech. 2019. CodeSonar home page. https://www.grammatech.com/products/codesonar.

[14] Scott Grant and Buddy Betts. 2013. Encouraging User Behaviour with Achievements: An Empirical Study. In *Proceedings of the 10th Working Conference on Mining Software Repositories (MSR '13)*. IEEE Press, Piscataway, NJ, USA, 65–68. http://dl.acm.org/citation.cfm?id=2487085.2487101

[15] JetBrains. 2019. IntelliJ home page. https://www.jetbrains.com/idea/.

[16] Brittany Johnson, Rahul Pandita, Emerson Murphy-Hill, and Sarah Heckman. 2015. Bespoke Tools: Adapted to the Concepts Developers Know. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2015)*. ACM, New York, NY, USA, 878–881. https://doi.org/10.1145/2786805.2803197

[17] Brittany Johnson, Yoonki Song, Emerson Murphy-Hill, and Robert Bowdidge. 2013. Why Don't Software Developers Use Static Analysis Tools to Find Bugs?. In *Proceedings of the 2013 International Conference on Software Engineering (ICSE '13)*. IEEE Press, Piscataway, NJ, USA, 672–681.

[18] C. Lewis, Z. Lin, C. Sadowski, X. Zhu, R. Ou, and E. J. Whitehead. 2013. Does bug prediction support human developers? Findings from a Google case study. In *2013 35th International Conference on Software Engineering (ICSE)*. 372–381. https://doi.org/10.1109/ICSE.2013.6606583

[19] Nguyen Quang Do Lisa, Ali Karim, Livshits Benjamin, Bodden Eric, Smith Justin, and Murphy-Hill Emerson. 2017. Just-in-time Static Analysis. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2017)*. ACM, New York, NY, USA, 307–317. https://doi.org/10.1145/3092703.3092705

[20] Mangala Gowri Nanda, Monika Gupta, Saurabh Sinha, Satish Chandra, David Schmidt, and Pradeep Balachandran. 2010. Making Defect-finding Tools Work for You. In *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 2 (ICSE '10)*. ACM, New York, NY, USA, 99–108. https://doi.org/10.1145/1810295.1810310

[21] Rita Orji, Regan L. Mandryk, and Julita Vassileva. 2017. Improving the Efficacy of Games for Change Using Personalization Models. *ACM Trans. Comput.-Hum. Interact.* 24, 5, Article 32 (Oct. 2017), 22 pages. https://doi.org/10.1145/3119929

[22] Rita Orji, Lennart E. Nacke, and Chrysanne Di Marco. 2017. Towards Personality-driven Persuasive Health Games and Gamified Systems. In *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems (CHI '17)*. ACM, New York, NY, USA, 1015–1027. https://doi.org/10.1145/3025453.3025577

[23] Haihao Shen, Jianhong Fang, and Jianjun Zhao. 2011. Efindbugs: Effective error ranking for findbugs. In *International Conference on Software Testing, Verification and Validation (ICST)*. 299–308.

[24] Fortify Software. 2019. Fortify home page. http://www8.hp.com/us/en/software-solutions/static-code-analysis-sast/.

[25] Nikolai Tillmann, Jonathan De Halleux, Tao Xie, Sumit Gulwani, and Judith Bishop. 2013. Teaching and Learning Programming and Software Engineering via Interactive Gaming. In *Proceedings of the 2013 International Conference on Software Engineering (ICSE '13)*. IEEE Press, Piscataway, NJ, USA, 1117–1126. http://dl.acm.org/citation.cfm?id=2486788.2486941

[26] Gustavo F. Tondello, Rina R. Wehbe, Lisa Diamond, Marc Busch, Andrzej Marczewski, and Lennart E. Nacke. 2016. The Gamification User Types Hexad Scale. In *Proceedings of the 2016 Annual Symposium on Computer-Human Interaction in Play (CHI PLAY '16)*. ACM, New York, NY, USA, 229–243. https://doi.org/10.1145/2967934.2968082

[27] Jim Witschey, Olga Zielinska, Allaire Welk, Emerson Murphy-Hill, Chris Mayhorn, and Thomas Zimmermann. 2015. Quantifying Developers' Adoption of Security Tools. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2015)*. ACM, New York, NY, USA, 260–271. https://doi.org/10.1145/2786805.2786816

[28] Shundan Xiao, Jim Witschey, and Emerson Murphy-Hill. 2014. Social Influences on Secure Development Tool Adoption: Why Security Tools Spread. In *Proceedings of the 17th ACM Conference on Computer Supported Cooperative Work & Social Computing (CSCW '14)*. ACM, New York, NY, USA, 1095–1106. https://doi.org/10.1145/2531602.2531722