Usage of Static Analysis Tools in the Integrated Development Environment

Master's Thesis

Tim van der Lippe

Usage of Static Analysis Tools in the Integrated Development Environment

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Tim van der Lippe

Software Engineering Research Group
Department of Software Technology
Faculty EEMCS, Delft University of Technology
Delft, the Netherlands
www.ewi.tudelft.nl

Usage of Static Analysis Tools in the Integrated Development Environment

Author: Tim van der Lippe
Student id: 4289439
Email: T.J.vanderLippe@student.tudelft.nl

Abstract

Developers make use of automation to perform repetitive and potentially error-prone tasks. One such automation can be categorised as *static analysis*, which aims to analyse program properties. The particular focus of this investigation are so-called ASATs (Automatic Static Analysis Tools). These ASATs are readily available for many programming languages and can be used to check coding style guidelines, elements of functional correctness and maintainability related issues. Previous studies on static analysis involved qualitative developer interviews and quantitative repository mining studies. This thesis uses automated telemetry to carry out a field study within the Integrated Development Environment (IDE), to obtain fine-grained data on developer behavior with regard to the actual use of ASATs. In addition, we have carried out a survey to validate the observed patterns. The field study is based on the Eclipse and IntelliJ plugin WatchDog, for which we elaborate upon an extensive investigation of static analysis observation techniques in the IDEs. Based on the quantitative data, we conclude the majority of all observed static analysis IDE events originate from few categories of warnings. Moreover, most of the warnings are resolved within one minute, with warnings related to type resolution being resolved the quickest. Developers corroborate these findings, but also confirm perceptions of earlier research that warnings contain large numbers of false positives. Based on both datapoints, we envision a data-driven future of static analysis tooling to optimize for usefulness for the developer rather than absolute correctness of tool implementations.

Thesis Committee:

Chair:	Dr. A.E. Zaidman, Faculty EEMCS, TU Delft
University co-supervisor:	M. Beller, Faculty EEMCS, TU Delft
Committee Member:	Prof. Dr. A. van Deursen, Faculty EEMCS, TU Delft
Committee Member:	Prof. Dr. C. Witteveen, Faculty EEMCS, TU Delft

Preface

This master thesis is the final project of my university career at the Delft university of Technology. Over the years, I have been able to acquire the necessary knowledge and experience to be fully prepared for my future career. Along the way, the many colleagues I met later also became my friends. First of all, I want to thank Andy Zaidman for his support and guidance during this final chapter. We have successfully worked together throughout the years on both educational and managerial challenges, which was my motivation to keep on collaborating for my thesis. While the process has not been easy, I am very grateful for the listening ears of my friends and family, which helped me to complete this project, and study as a whole, with success. I also want to thank my study and SERG colleagues for the interesting discussions, with special thanks to Moritz Beller for his critical reviews and discussions as part of our work on WatchDog. To that end, the previous work on WatchDog was also fundamental to the work elaborated upon in this thesis, for which I thank everyone who previously contributed to the project. Lastly, I am grateful to be a member of the Polymer team, of which my team members have remained supportive and in particular helped me distribute our survey to their acquaintances.

Tim van der Lippe
Delft, the Netherlands
June 25, 2018

Contents

List of Figures

List of Tables

Chapter 1

Introduction

As software projects grow, Software Engineering practices have to be adopted to ensure high quality software can be built [53]. The practice of Software Engineering becomes more and more crucial when software is used in critical systems for society such as medical surgeries [66], autonomous vehicles [17, 65] and (online) banking systems [26]. Failures in these systems can have and have had lethal consequences [37]. The challenges of ensuring that critical societal systems do not result in loss of life (such as fault tolerance, failure cascading and security) are known for quite some time [36]. Solutions and precautionary methods include clear software specifications as well as formal verification of these specifications.

While system failures can have lethal consequences, there are also non-life-threatening software failures with large consequences. Examples include widespread outages of social media platforms preventing citizens to communicate [28, 46], but also singular incidents with devastating effects such as loss of property for individuals [34]. Small bugs can be the cause of such failures which have a much larger impact.

To prevent bugs from having a large impact, multiple approaches have been proposed. One approach to reduce the overall impact of a bug is an auto-recovery system that can be used to rollback a faulty program to the last known healthy state [54]. However, while mitigation can be successful, prevention of the bug is more desirable. Thoroughly testing a software product can prevent failures, as improper testing has been shown to be disastrous for large systems [18]. Besides testing, the practice of static analysis is also used to detect bugs by analyzing the source code to alert developers of software failures before they occur [4]. Facebook uses static analysis to prevent new employees unfamiliar with C++ to make common (yet detrimental) mistakes [13]. Other popular open source projects like Chromium are regularly audited for security issues using static analysis tools [32].

Previous research has focused on observing and interviewing developers regarding their usage of static analysis tools [31], including a large study on tool configuration and warning evolution over a sequence of commits [8, 67]. However, a behavioral study on the usage of static analysis tools in the Integrated Development Environment (IDE) has not been conducted. Given the evolution between commits, we want to obtain more fire-grained data by analyzing how developers deal with warnings while working on their software project.

1.1 Research Questions

To learn more about the behavior and usage of static analysis tools in the IDE, the following research questions (combined with their respective motivation) are the basis of this master thesis:

RQ1 What steps are required to monitor static analysis warnings occurrences in the IDE?

To be able to answer any question regarding behavior and usage of static analysis tools, we need to obtain the required information from the IDE. Therefore, we need to investigate how we can instrument an IDE to be able to listen to the warning occurrences and discover any potential complicating factors of the chosen approach.

RQ1.1 What steps are required to monitor external static analysis plugins that augment an IDE?

Not only the IDE, but also other static analysis plugins issue static analysis warnings. To be able to integrate additional plugins alongside the core IDE, we need to also monitor external plugin messages.

RQ2 What is the relative frequency between static analysis warning categories?

Static analysis tools are used for a variety of use-cases. To that end, we hypothesize that static analysis warnings from one category are more frequently occurring than other categories, as developers prioritize failures in one category over another. We are interested in the distribution of warning categories, whether several categories are dominant or if the overall distribution is largely uniform.

RQ2.1 Do experienced developers have lower warning frequency?

Besides the relative frequency between categories, we are also interested in the relative frequency between developers. Here, we want to know whether the programming experience of developers is influential in the total warning frequency.

RQ3 What is the lifetime of a static analysis warning?

Static analysis warnings are shown to the developer to incentivize them to resolve a particular issue. The end goal of a warning is therefore the resolution of the warning. Therefore, the lifetime of a warning is interesting to understand how developers react to these warnings. Here, we assume that a shorter resolution time indicates that the warning was more easy and/or more important to resolve.

RQ3.1 Do experienced developers resolve warnings quicker?

Besides overall lifetime, we are interested in the effect of programming experience on the lifetime. We anticipate to find a difference in lifetime, as we think experience indicates that developers have more knowledge on how to tackle these warnings.

RQ3.2 Do developers with a lot of warnings resolve these warnings quicker?

While experience is a factor, another viewing angle is the lifetime compared to actual frequency. It might be the case that developers who see very few warnings also take a long time to resolve them, while developers who see a lot of warnings also solve them much quicker. Or vice versa: developers who see a lot of warnings are overwhelmed and are disinclined to resolve any warning at all.

RQ4 Are there differences between sections of a file in terms of created or resolved warning frequency?

Not only the frequency, but the location of a warning can also be a factor in terms of resolution frequency. If a warning is in a section that is not regularly viewed by a developer, chances are this warning is unlikely to be resolved at all. Therefore, we would like to know if particular sections of a file (categories of) warnings are more frequently occurring than other sections.

RQ4.1 Are there differences between sections of a file in terms of unresolved warning frequency?

With successful resolution, the developer not only was aware of the warning, but was also able to apply an appropriate fix. For all warnings that have not been resolved (yet), the location of the warning could be an indicator for why the warning is not resolved.

For example, imports are declared at the top of the file. Any unused imports warnings are usually not in view when a developer is working on a particular piece of code, so could be unaware of the existence of these warnings.

RQ5 How do developers perceive the usefulness of static analysis warnings?

Besides a quantative analysis, we are interested in the qualitative view of developers on static analysis tools. Questions will focus on usability as well as appliance of general practices.

RQ5.1 How do developers try to resolve a particular static analysis warning?

We would like to know what approach developers take to resolve warnings. This data-point augments our observations with the motivations for and way of resolving these warnings.

RQ5.2 Does the usage of static analysis tools deter developers from contributing to a project?

Together with the general practices, we would like to know the impact of tools being used to influence contribution efforts of developers. In here, we want to know if adoption of static analysis tools is perceived as a roadblock to contributions or not.

1.2 Chapter overview

This master thesis should be placed in the appropriate research context. To that end, Chapter 2 elaborates on previous research, their findings and interesting results that provide a basis for this work. The first step of analyzing behavior is deciding the mode of observation. Chapter 3 lists several options and explains the final decision on tool choice. The implementation of the tool in the two IDEs IntelliJ Idea and Eclipse, explained in Chapter 4, answers research question **RQ1**. As a result of a lack of detailed API, Chapter 5 expands on the implementation by introducing a classification algorithm for static analysis warnings. By classifying warnings, we lose detailed information, which has implications for answering research question **RQ1.1**.

Research questions **RQ2** through **RQ4** are answered by the results analysis based on the obtained telemetry, listed in Chapter 6. Chapter 7 answers research question **RQ5** based on the responses to a survey we published. Based on our findings, we list several threats to validity and we propose several improvements to both research and industry implementations of the IDEs and static analysis tools in Chapter 8.

Chapter 2

Related work

This master thesis should be placed in the context of related work performed on static analysis and editors. This chapter therefore includes two sections for both topics as well as a section that elaborates on the combination of these two topics.

2.1 Static Analysis

Static analysis is the practice of analyzing the structure of a program, with the goal of obtaining certain properties of that program [23]. Analyses can be run on different representations of the program: the Abstract Syntax Tree (AST) [71], the call graph of functions with their calling dependencies [6] or the control flow graph of a single function [41]. The properties that can be obtained include type checking to ensure correctness of a program on compile time [16], performing optimizations based on a class hierarchy analysis [21] or finding potential security issues [43]. While static analysis can obtain a variety of properties of a program, the practice remains undecidable and therefore prohibits static analysis from perfectly predicting the outcome without running the program [40].

One application of static analysis is in an Automated Static Analysis Tool (ASAT), which can be used for different purposes [31]. A common use-case is the automation of tasks that would otherwise be fully manual. By automating such tasks, developers not only save time, but the risk of missing potential issues is also lower. A concrete example is the automation of checking for missing license headers in open source projects in order to prevent code being published without a license [70]. Another example is that static analysis is also used to enforce consistency of code style across developers in the same development team [8, 31].

ASATs are run in multiple development contexts. Vassallo et al. define three contexts: *continuous integration*, *code review*, and *local programming* [68]. Running ASATs in a Continuous integration (CI) build can lead to a reduction in total number of warnings in a project [67]. During code review, the ASATs results obtained from a CI build are an important factor for integrators when they need to decide whether or not to accept a contribution [27]. Lastly, ASATs are integrated into the local programming context via command line interface (CLI) [10] as well as in their Integrated Development Environment (IDE) [31].

2.2 Integrated Development Environment

An IDE is an application that contains a variety of tools to assist developers, while offering document editing capabilities. Early research on software development discovered the need for unification of and streamlining on software tools to improve the productivity of developers [12]. Over time, IDEs evolved into a one-stop-shop with support for language features such as syntax highlighting, autocompletion, documentation-on-hover and quick navigation using reference resolution [51, 69]. Moreover, the language workbench Spoofax has been developed to program these language features using Domain Specific Languages (DSLs) [33]. Kats and Visser developed Spoofax as they realized that successful adoption of a programming language largely depends on the external support for this language. Over time language features like syntax highlighting became a prerequisite for adoption of the programming language. However, since implementing these features is a time-consuming task, using the DSLs available in Spoofax allows for faster development as solutions for common subproblems are provided by the DSL codegenerator. To that end, IDE features became a vital part of the current software engineering process.

Based on the PopularitY of Programming Language (PYPL) index, the most popular IDEs are Visual Studio[1], Eclipse[2], Android Studio[3] (based on IntelliJ), Netbeans[4] and IntelliJ[5] [15]. Some of these IDEs have a dedicated marketplace for extensions, for example the Eclipse Marketplace[6] or the Visual Studio Code marketplace[7]. The marketplaces contain a wide variety of extensions that can enhance the functionality of the editor. Examples include improved editor autocompletion for import paths[8,9] or the ability to debug a running application in a web browser[10]. Extensions became such an integral part of the ecosystem that curated lists are available that list the most popular/useful extensions for an editor[11,12].

Previous research has been conducted to obtain quantatitive data on how developers use the IDE to execute automated tests [7], perform debugging activities [62] and which views of an IDE are commonly used [47]. Murphy et al. found out that in Eclipse the *package explorer* view is most commonly used to navigate to code of interest [47]. The *package explorer* is the view that displays the folder structure, Java package structure as well as the project dependencies. Other commonly used views are the *console*, *search* and *problems* view. The *problems* view is the view that shows all static analysis warnings that exist in the currently opened project, which was used by 95% of the developers.

[1] https://www.visualstudio.com/
[2] https://projects.eclipse.org/
[3] https://developer.android.com/studio/
[4] https://netbeans.org/
[5] https://www.jetbrains.com/idea/
[6] https://marketplace.eclipse.org/
[7] https://marketplace.visualstudio.com/
[8] https://marketplace.visualstudio.com/items?itemName=christian-kohler.path-intellisense
[9] https://atom.io/packages/autocomplete-module-import
[10] https://marketplace.visualstudio.com/items?itemName=msjsdiag.debugger-for-chrome
[11] https://github.com/viatsko/awesome-vscode
[12] https://github.com/mehcode/awesome-atom

Listing 2.1: Dead code detection on the second *if*-statement

```
if (text == null) {
    text = "";
}
if (text != null) {
    System.out.println(text);
} else {
    System.out.println("Text is null!");
}
```

Listing 2.2: Refactored solution after invoking the developer action in IntelliJ

```
if (text == null) {
    text = "";
}

System.out.println(text);
```

Figure 2.1: Before and after of invoking a developer action on a dead code warning

2.3 Static analysis in the IDE

There are multiple ways how static analyses (warnings) are integrated in the IDE. IntelliJ Idea by default ships with integrated static analyses such as dead code detection and finding probable bugs [30]. Any issues that are detected are shown in the editor with appropriate developer actions. For example, given a *if*-statement which is always true, the corresponding developer action is to delete the condition. Figure 2.1 shows the before and after of invoking the developer action on the *if*-statement condition.

Besides the default analyses, there are many IDE extensions that integrate ASATs into the editor. SpotBugs[13] (formerly known as FindBugs[14]) is an open source ASAT. While ASATs can be used during code reviews [50], Johnson et al. found that integration of FindBugs into the IDE is an important factor for the successful application of static analysis [31]. By warning the developer as soon as possible, issues can be found and resolved more quickly, possibly even automatically. In general, the longer it takes in the software engineering development lifecycle to mitigate/resolve a problem, the more expensive it becomes [64].

While most warnings that are generated have an automated fix that can be applied to resolve the warning (also known as refactoring [24]) this is not always the case. This problem

[13] https://spotbugs.github.io/

[14] http://findbugs.sourceforge.net/

mostly stems of the inability to implement perfect refactoring techniques which do not alter the program behavior [45]. The original definition of refactoring was coined by Opdyke which defined a refactoring as behavior-preserving when the program before and after the refactoring produces the same output given the same input set [49]. A previous exploration on the behavior-preservingness of refactorings implemented in the IDEs Eclipse, JRRT and Netbeans has shown that implementations differ and sometimes produce invalid refactorings [60].

Chapter 3

Automated telemetry

To be able to gather statistics of the usage and occurrence frequency of static analysis warnings in editors, either a manual analysis must be performed or a tool that gathers automated telemetry must be developed. This chapter elaborates upon the various options and has a deep dive into the multiple ways of implementation and basis of an tool for automated telemetry.

3.1 Modes of observation

Babbie defines several modes of observation for social research, including experiments, survey research, field research, unobtrusive research and evaluation research [5]. Of these modes of observation, several modes are not usable for this thesis.

Firstly, evaluation research focuses on evaluating the results of a social intervention. In other words, given a social intervention in a society, what are the changes to this society and do they match the intended results? Since we want to observe the real-world practices of software developers, we have no intention to change their behavior via the means of a social intervention. Evaluation research is therefore not applicable for us.

Similarly, social experiments consist of letting experiment subjects perform a particular action and observe any effects of this action on the subjects. This is almost equivalent to evaluation research with the distinction that social experiments are not targeted to a specific expected result. In other words, while evaluation research expects a certain result, social experiment are not concerned with a desired effect.

In contrast to the former two options, unobtrusive research more closely matches the desired research method. For unobtrusive research, existing statistics must be available. Babbie provides several examples, including inspecting the radio dial settings when a car is at the repair shop, to be able to determine the popularity of various radio stations. By counting radio dial settings, car drivers (the subjects in this research) are unknowingly surveyed for their radio station choice and are therefore not influenced by knowing they are being surveyed. While this is desirable for our research, there are no existing statistics available for research. We must therefore implement the statistics gathering process ourselves, to be able to draw conclusions in a similar manner to unobtrusive research.

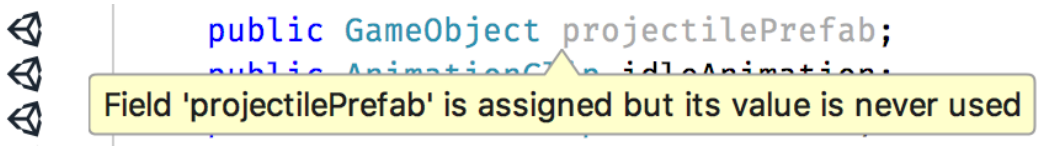

Figure 3.1: A tooltip above a static analysis warning in Rider[1]

The last two active ways of obtaining statistics are survey research and field research. In both research types subjects are aware of the existence of an observer, which is the explicit difference with unobtrusive research. However, the presence and influence of the observer can be limited to reduce the influence of "the problem of reactivity" [5]. Reactivity means that subjects modify their behavior if they are aware of the existence of an observer. This problem is most prevalent in survey research, where subjects might answer questions for what they expect the researchers are looking for. Moreover, the behavior of subjects can be influenced if they are simply aware of any observance at all, even if the researcher has not performed any action or changed the social setting [58]. Even if the researchers' intent to perform an action to induce a social effect, the effect might be an artifact of the researchers performing the research, rather than the desired result of the action. This effect is also known as the Hawthorne Effect [1].

3.1.1 Static Analysis Warnings Observation

Given the described modes of observation, we are looking for a measure to observe developers, while preventing our research to influence their behavior. To that end, observing developers in a controlled environment with manual inspection of the static analysis output is undesirable. Manual analysis requires analyzing the tooltips of generated warnings (an example is shown in Figure 3.1) or the full "Static Analysis" perspective (which is shown in Figures 3.2 and 3.3), which would interrupt the developer workflow and influence their behavior. Therefore, we need a field study to silently observe characteristics while the developer resides in their own environment, to minimize both the realization of developers they are being observed and to maintain their own working environment.

All in all, we have to investigate the possibility of an automated method to gather statistics of static analysis warnings. This automated telemetry tool should be a silent observer, installed in the developer workflow and be able to gather any interesting characteristics. Once we have a corpus of existing characteristics, we can use the method of unobtrusive research to deduce behavior of our developer subjects. Afterwards, we can extend our research findings with a developer survey to ask for targeted clarification of our findings.

[1]Source: https://blog.jetbrains.com/dotnet/2017/08/30/getting-started-rider-unity/
[2]Source: https://blog.jetbrains.com/idea/2016/05/intellij-idea-2016-2-eap-162-426-1-is-out/
[3]Source: https://wiki.eclipse.org/CDT/designs/StaticAnalysis

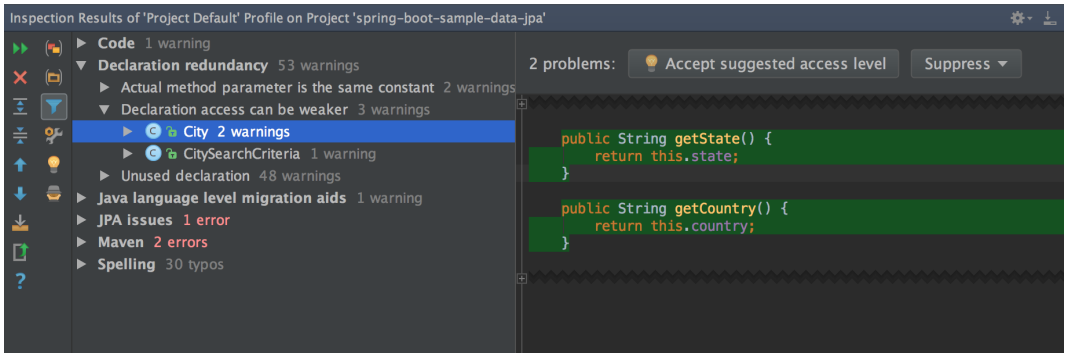

Figure 3.2: The Inspection tool window in IntelliJ IDEA[2]

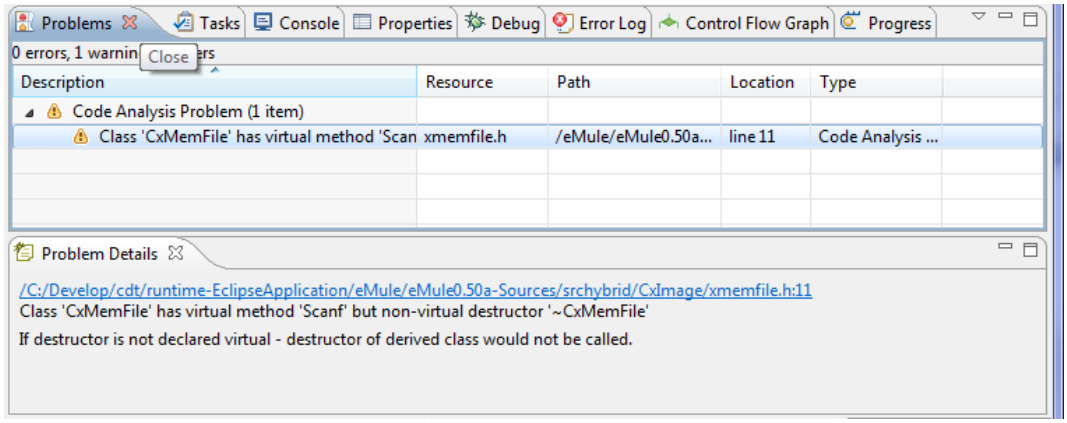

Figure 3.3: The Problem Details view in Eclipse[3]

3.2 Automated tool options

Our chosen mode of observation for the static analysis warnings is field research. To that end, we need an automated tool that can silently observe the developers behavior. Investigating potential ready-made solutions for such a statistics analysis tool resulted in no tools capable of tracking static analysis warnings. This left us with two options:

1. **Develop a new tool from scratch**

   The first possibility is to develop a new tool from scratch. This option provides full freedom in the implementation, the architecture design and flexibility in the development process.

2. **Adapt an existing tool**

   Another option is to adapt an existing static analysis tool and implement the necessary extensions to also be able to gather statistics on static analysis warnings.

There are positive and negative aspects to both options. The dilemma between starting from scratch or adapting an existing tool is also prevalent in the migration from legacy

systems to a newer version [14]. An existing system poses architectural challenges on the new functionality, for which limitations can be removed by starting fresh. By starting from scratch, the developers can have a full focus on implementing the analysis as-is, without the need to consider an existing architecture. In contrast, adapting an existing tool means that certain aspects are already implemented. (Obviously a requirement for adaptation of a tool is the existence of such a tool) For example the data storage, networking capabilities and overall dataflow design have been thought out. While an existing architecture is nice, there could be unspecified assumptions that could slow down or completely prohibit implementing the required extension.

One other aspect to consider is the need of participants that will use the tool to gather the data. To be able to analyze data, users must use the tool to generate this data. Starting from scratch means that potential users have to be convinced to participate by installing the tool. Instead, an existing tool that already has users can be updated to use the new observation features. Updating an existing tool imposes a significantly lower barrier for users than going through installation and learning a new tool.

The choice of the automated telemetry tool is therefore two-sided: the technical aspects of the implementation and the social aspects in terms of users. Given the time-constraints of the master thesis combined with the technical skills of the authors, the technical aspects can be resolved in less time than the social aspects of user recruitment. As such, we made the assumption that, to be able to analyze a substantial amount of data, adapting an existing system would eventually result in the most useful and diverse data.

3.3 Analysis existing tools

Searching source code hosting websites such as GitHub[4] for editor plugins capable of statistics tracking gave us the following results:

- **Code::Stats**[5]

  Code::Stats is a small, programming experience gathering and open-source[6] service. The various editor plugin implementations listen for keystrokes and assign experience points for the amount of code the developer typed. While the software is open-source, the gathered data is stored on a server maintained by the authors and is available through a REST API[7].

- **WakaTime**[8]

  WakaTime is a proprietary, yet open-source[9], Software As A Service (SAAS) solution for time tracking development by users. It offers an interactive dashboard to monitor the development hours over time, including possible filters by programming

[4] https://github.com
[5] https://codestats.net/
[6] https://github.com/code-stats
[7] https://codestats.net/api-docs
[8] https://wakatime.com/
[9] https://github.com/wakatime

language, project and usage of a versioning system. Just like with Code::Stats is the software of WakaTime open-source, but requires a REST API[10] to access all characteristics.

- **WatchDog**[11]

  WatchDog is an open-source[12] editor plugin for IntelliJ Idea[13] and Eclipse[14] developed by SERG-Delft, led by Moritz Beller (co-supervisor of this thesis) [7]. WatchDog keeps track of editor events such as keystrokes, test runs and debug events. The generated data of WatchDog is stored in a MongoDB[15] instance accessible to research members of SERG-Delft.

3.4 Existing tools pros and cons

There are several influencing factors for choosing an existing tool. First of all, there should be sufficient infrastructure available to allow new features to be added. This infrastructure includes ease of deployment for the authors, to be able to make updates on the data gathering process. Moreover, the generated data should be easily accessible to the authors. Based on these factors, the following conclusions were made regarding the list of existing tools.

Code::Stats is focused on translating the number of keystrokes to approximate programming experience. While keeping tracking of keystrokes can be easily expanded to also track static analysis warnings, there is an explicit focus on programming experience. We expect that the maintainers of Code::Stats will likely be reluctant to support other functionality, which would complicate the introduction of static analysis features. Therefore, we decided to not expand upon Code::Stats as it would be unlikely to land the required features to obtain the data.

Development of WakaTime requires approval by the company maintaining the SAAS solution. This means that, before we as researchers are able to analyze the data, we would need to go through an approval cycle with the authors, update all corresponding plugins and then convince users to allow our application from querying the REST API. Given the time constraints of this thesis and the risk of potential rejection of product updates in conjunction with the existing need of convincing existing users to allow us access to their data, WakaTime is not a viable option.

WatchDog is a TU-Delft hosted solution, which means that the authors can publish updates and receive data more easily than a proprietary solution like WakaTime. There is existing infrastructure for tracking of events with corresponding networking and database storage capabilities. However, there has not been active development on WatchDog for 1,5 years. This could potentially be problematic in terms of discovering maintenance problems while working on expanding the feature set.

[10] https://wakatime.com/developers
[11] https://testroots.org/testroots_watchdog.html
[12] https://github.com/testRoots/watchdog/
[13] https://www.jetbrains.com/idea/
[14] https://www.eclipse.org/
[15] https://www.mongodb.com/

3.5 Final choice

Overall, WatchDog is both developed by SERG-Delft and has sufficient existing infrastructure and users to built upon. The other two tools are proprietary and expose greater risk in terms of feature acceptance and gaining access to the gathered users data. While WatchDog has not seen active development for a while, these technical challenges are easier to overcome than the political challenges of convincing businesses of implementing the research functionality.

Chapter 4

Static analysis in editors

The implementation of the static analysis tracking in WatchDog requires two editor-specific implementations: IntelliJ IDEA and Eclipse. These two editors are already supported by WatchDog and have hundreds of users per editor [9]. The overall programming models are different in each editor, which requires custom solutions for each editor. This section describes the thought process and implementation details of the editor plugins. In the end, the capabilities of each editor have an impact on the fidelity of the gathered data that can be analyzed by the authors.

4.1 Code Inspection with IntelliJ IDEA

JetBrains uses the concept of Code Inspection: "IntelliJ IDEA features robust, fast, and flexible static code analysis. It detects the language and runtime errors, suggests corrections and improvements before you even compile." [63] Code Inspections are integrated in the application in multiple ways. While editing source code, potential warnings and compilation errors are shown in the text editor, see Figure 3.1. Developers can quickly navigate between these warnings to resolve any possible issue in a timely manner. Secondly, any compilation error is also shown in the project file tree view. Figure 4.1 shows how a compilation error in *Launcher.scala* produces a red squiggly underline on the file and all its parent directories. Since there is no compilation error in the *core* directory, this directory does not have a red underline. Using these highlights, developers can find non-compiling files in a large project. Lastly, developers can execute the full set of available code inspections in the *Analyze* submenu with bulk mode[1]. All discovered issues or warnings are shown in the Inspections Result Tool Window, as shown in Figure 3.2.

[1] `https://www.jetbrains.com/help/idea/running-inspections.html`

[2] Source: `https://blog.jetbrains.com/scala/2017/06/26/intellij-idea-scala-plugin-2017-1-3-simplified-project-view-scalatest-selection-by-regexp-improved-akka-support/`

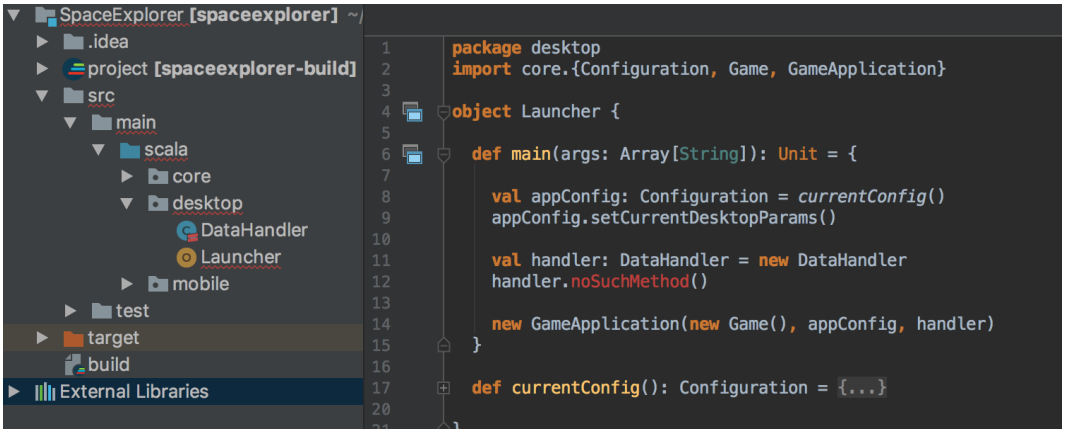

Figure 4.1: File tree view in IntelliJ IDEA highlighting a compilation error in *Launcher.scala*[2]

4.1.1 Internal implementation

While there is extensive documentation on how developers can use the code inspections, there is no documentation on the internal implementation of the editor. This required extensive research by us to determine how these features are implemented.

The features described in the previous section are all based on the *DaemonCodeAnalyzer*[3]. This Daemon runs in the background to run all registered inspections retrieved from *InspectionToolRegistrar*[4]. Each *InspectionTool* can be independently run and is either *Local* or *Global*. *LocalInspectionTools*[5] are tools that take 1 file as input, to compute potential issues. In contrast, *GlobalInspectionTools*[6] runs on an *AnalysisScope* with a *GlobalInspectionContext*. These classes contain the required traversal techniques to process multiple files at once. The traversal techniques are based on the Visitor pattern [25].

Both types of tools return a list of *ProblemDescriptor*[7] objects which describe the issues that have been found, which *PsiElement* they are tied to in the Abstract Syntax Tree (AST) and whether they can be automatically fixed using a *QuickFix*[8]. The *ProblemDescriptor*s are of interest for the static analysis statistics gathering, as they indicate the found problems and are also shown to the developer in the editor. To create a *ProblemDescriptor*, the inspection

[3] https://github.com/JetBrains/intellij-community/blob/b205caf9f3/platform/analysis-api/src/com/intellij/codeInsight/daemon/DaemonCodeAnalyzer.java

[4] https://github.com/JetBrains/intellij-community/blob/40fb7c49db/platform/analysis-impl/src/com/intellij/codeInspection/ex/InspectionToolRegistrar.java

[5] https://github.com/JetBrains/intellij-community/blob/40fb7c49db/platform/analysis-api/src/com/intellij/codeInspection/LocalInspectionTool.java

[6] https://github.com/JetBrains/intellij-community/blob/40fb7c49db/platform/analysis-api/src/com/intellij/codeInspection/GlobalInspectionTool.java

[7] https://github.com/JetBrains/intellij-community/blob/e674d7047c/platform/analysis-api/src/com/intellij/codeInspection/ProblemDescriptor.java

[8] https://github.com/JetBrains/intellij-community/blob/e674d7047c/platform/analysis-api/src/com/intellij/codeInspection/QuickFix.java

tools use the *InspectionManager*[9].

However, the classes described thus far do not allow for external actors to attach listeners to be notified whenever an issue is detected. This means that it is not possible to be notified whenever each of these API's is invoked and it is not possible to know which *ProblemDescriptor*s are created. To be able to retrieve the issues, a different method had to be developed.

4.1.2 Searching for an API hook

There are multiple possibilities for pro-actively retrieving the generated issues. This would mean that in reaction to a pre-defined event, the tool could request the current state of generated issues. To be able to investigate possibilities, we searched for all usage occurrences of *ProblemDescriptor* throughout the editor source code.

GlobalInspectionContext

The first attempt was to use the *GlobalInspectionContext* to retrieve any issues that are associated with this context. To recall, the context is used in *GlobalInspectionTool*s, which they traverse and based on the AST can report issues. The reason this seemed to be a fruitful attempt was that the context exposed a method called *getPresentationOrNull*, which returns an *InspectionToolPresentation* that in turn implements *getProblemDescriptors*. The snippet in Listing 4.2 shows the API usage of this approach. However, the problem with this approach is the inability to retrieve the *GlobalInspectionContext* that was actually used by the *DaemonCodeAnalyzer*. This meant that using this approach on a new *GlobalInspectionContext* always returns an empty *problems* list.

codeAnalyzer.runMainPasses

Since the original context was unavailable, another option was to re-run the *DaemonCodeAnalyzer* to retrieve the raw output. This would be possible with the *runMainPasses* method which can run on a document to get back a list of *HighlightInfo*. The full snippet is shown in Listing 4.3. While *HightlighInfo* does not contain the same information as *ProblemDescriptor*, it still stores the textual output which developers read in the tooltips and the Inspection Results window. As the snippet shows, this approach requires the existence of a reference to the *psiFile* and *editor*. Invocation of *runMainPasses* should therefore happen in a callback that is invoked whenever an *editor* is created.

Contrary to our expectations, *runMainPasses* always returns an empty list when invoked. Extensive research of the implementation of *runMainPasses*[10] and a similar-looking method *runPasses*[11] were not fruitful. Given the non-existence of source code documenta-

[9] https://github.com/JetBrains/intellij-community/blob/e674d7047c/platform/analysis-api/src/com/intellij/codeInspection/InspectionManager.java

[10] https://github.com/JetBrains/intellij-community/blob/62ce2234ba/platform/lang-impl/src/com/intellij/codeInsight/daemon/impl/DaemonCodeAnalyzerImpl.java#L230-L282

[11] https://github.com/JetBrains/intellij-community/blob/62ce2234ba/platform/lang-impl/src/com/intellij/codeInsight/daemon/impl/DaemonCodeAnalyzerImpl.java#L292-L399

```
final InspectionManagerEx instance =
        (InspectionManagerEx) InspectionManager.getInstance(project);
GlobalInspectionContextImpl context =
        (GlobalInspectionContextImpl) instance.createNewGlobalContext(true);
final List<InspectionToolWrapper> inspectionToolWrappers =
        InspectionToolRegistrar.getInstance().get();

final List<CommonProblemDescriptor> problems = inspectionToolWrappers.stream()
    .flatMap(wrapper -> {
        final InspectionToolPresentation presentationOrNull =
                context.getPresentationOrNull(wrapper);

        if (presentationOrNull == null) {
            return Stream.empty();
        }

        return presentationOrNull.getProblemDescriptors().stream();
    })
    .collect(Collectors.toList());
```

Figure 4.2: Retrieve all *ProblemDescriptor*s from the *GlobalInspectionContext*

```
final DaemonCodeAnalyzerImpl codeAnalyzer = (DaemonCodeAnalyzerImpl)
        DaemonCodeAnalyzerImpl.getInstance(project);
final List<HighlightInfo> highlightInfos = codeAnalyzer.runMainPasses(
        psiFile, editor.getDocument(), new EmptyProgressIndicator());
```

Figure 4.3: Manually run the *DaemonCodeAnalyzer* on a file

tion, it is unclear what the original intention of the API implementers is nor what requirements are imposed on the state of the application. The function is not pure, as it relies on the side-effect of registered highlighters. It is unclear how external developers can initiate this state such that *runMainPasses* returns the corresponding warnings. This rendered this approach useless for our intent.

InspectionEngine.runInspectionOnFile

At this point it became clear that retrieving the warnings by means of obtaining the previous output, using one of the classes described above, would not work. Besides the *DaemonCodeAnalyzer*, another class related to running inspections is the *InspectionEngine*[12]. This class exposes the method *runInspectionOnFile*, which requires similar parameters to the ones used in the previous approaches. Listing 4.4 shows the usage of this API, which relies on the variables declared in the Listing 4.2.

[12] https://github.com/JetBrains/intellij-community/blob/62ce2234ba/platform/analysis-impl/src/com/intellij/codeInspection/InspectionEngine.java

```
final List<ProblemDescriptor> problems = inspectionToolWrappers.stream()
        .flatMap(wrapper ->
                InspectionEngine.runInspectionOnFile(file, wrapper, context).stream())
        .collect(Collectors.toList());
```

Figure 4.4: Manually run each *InspectionTool* on a file with the *InspectionEngine*

```
1 import java.util.List;
2
3 public class Foo {
4     public static void main(String[] args) {
5         if (!!false) {
6
7         }
8     }
9 }
```

Figure 4.5: Example file with several issues

This approach successfully returned some issues for the example file shown in Listing 4.5. In this example, line 1 has an unused import, the expression on line 5 a double negation and the full if-statement on lines 5 through 7 is empty. These three warnings are detected by IntelliJ IDEA, where the unused import has gray text (it is a weak warning) while the latter two have a yellow background, as shown in Figure 4.6. However, closer inspection of the returned issues showed that only the yellow highlighted warnings were returned by the API. Weak warnings, such as the unused import and duplicate code detection, could not be retrieved.

Secondly, the large number of existing inspections resulted in a long computation time of *runInspectionOnFile*. On the example file, one single iteration took several seconds. During this time, the User Interface of IntelliJ is unresponsive, which is intrusive to developers using the plugin. Attempts to move the invocations to a background thread[13] resulted in *Exceptions* regarding "invalid Thread access". Invocations of *runInspectionOnFile* are only allowed in the UI thread, which would significantly impact the developer experience. This method would therefore be intrusive to the subjects that have WatchDog installed. Not only would the input delay be undesirable, it also introduces the problem of reactivity (as explained in Section 3.1) and thus influence our research results.

[13] http://www.jetbrains.org/intellij/sdk/docs/basics/architectural_overview/general_threading_rules.html#background-processes-and-processcanceledexception

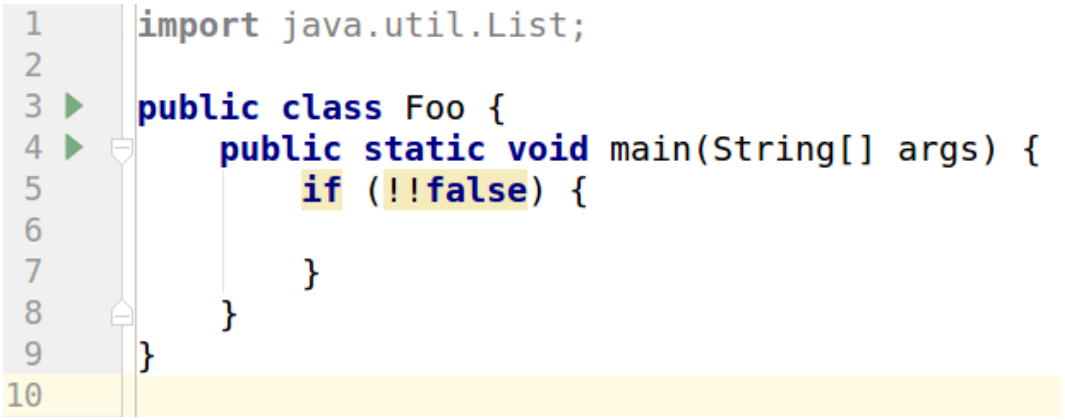

Figure 4.6: Editor warnings as shown in IntelliJ IDEA

MarkupModel.getAllHighlighters()

After reaching out to JetBrains[14], explaining the unsuccessful attempts thus far, one IntelliJ engineer pointed out the existence of an API to retrieve all warnings shown to the user. This API consists of obtaining the *MarkupModel*[15] representation of the document via *DocumentMarkupModel.forDocument(document, project, true)*. The model is the in-memory representation of all warnings, for which *MarkupModel.getAllHighlighters()* returns all *RangeHighlighter*s which represent both the weak warnings as well as the more prominent yellow highlighted warnings. *RangeHighlighter*s represent the warnings shown, but have a more limited API than *ProblemDescriptor*s.

To be able to retrieve the warning, *RangeHighlighter.getErrorStripeTooltip()* returns an *HighlightInfo*[16] which has a severity, textual description and quickfixes. The textual description is the serialized user-friendly format representing the original *ProblemDescriptor*. However, for a consumer interested in static analysis statistics gathering, the textual description poses challenges, which limit the usability and overall value of the data.

Conclusion

Concluding, the internal Code Inspection implementation in IntelliJ does not expose an API hook for external listeners. Only the in-memory representation of the warnings exposed sufficient amount of data for the use-case. However, the *HighlightInfo* class exposes only a textual description, which impacts the overall usability of the data and future data analysis. For more information about this impact, see Chapter 5.

[14] https://intellij-support.jetbrains.com/hc/en-us/community/posts/115000759224-Obtain-Inspection-output-for-a-given-file

[15] https://github.com/JetBrains/intellij-community/blob/cebd5236a3/platform/editor-ui-api/src/com/intellij/openapi/editor/markup/MarkupModel.java

[16] https://github.com/JetBrains/intellij-community/blob/master/platform/analysis-impl/src/com/intellij/codeInsight/daemon/impl/HighlightInfo.java

```
IWorkspace workspace = ResourcesPlugin.getWorkspace();
IResourceChangeListener markupModelListener =
        new EclipseMarkupModelListener();
workspace.addResourceChangeListener(markupModelListener,
        IResourceChangeEvent.POST_BUILD);
```

Figure 4.7: Attach an *IResourceChangeListener* to the workspace

4.2 IMarkers in Eclipse

Using the knowledge obtained from the investigation of Code Inspections in IntelliJ, searching for similar capabilities in Eclipse was a lot easier. Eclipse uses the notion of Resource Markers[17]. To be able to gather markers in a file, any *Resource* implements the method *findMarkers*. For example, *file.findMarkers(IMarker.PROBLEM, true, 0)* returns all (static analysis) issues in the *file*.

To be able to listen to resource changes, Listing 4.7 shows how the *EclipseMarkupModelListener* in WatchDog is attached to a workspace. Every time the user saves a file and invokes a build, the *IResourceChangeListener*[18] is invoked. In the callback of the listener, a delta of all changed files is provided. Using the Visitor pattern, a *IResourceDeltaVisitor*[19] can iterate through all changed files and then invoke *findMarkers* on each resource.

At this point, we obtained a list of current warnings that exist in the resource. However, unlike the IntelliJ implementation, we are not notified when markers are removed. This means that we have to keep track of all changes ourselves, by comparing these snapshots of markers and performing a diffing algorithm on these lists.

4.2.1 Diffing algorithm

The abstract problem of efficiently computing a diff of markers in Eclipse is the Longest common subsequence problem. Implementations for this problem are included in programs such as Unix *diff*[20] and *git diff*[21]. The implementation in WatchDog Eclipse is inspired by the $O(ND)$ algorithm introduced by Myers [48].

This algorithm is based on an edit-graph, in which modifications are represented as horizontal or vertical movement, while equivalent characters are diagonal steps through the graph. Figure 4.8 shows how the difference between source string *xCxDxE* and target string *xCDxEx* is visually represented. In this case, a horizontal edge indicates the deletion of a character in the source string, while a vertical edge indicates the addition of a character in

[17] https://help.eclipse.org/neon/index.jsp?topic=%2Forg.eclipse.platform.doc.isv%2Fguide%2FresAdv_markers.htm

[18] https://github.com/eclipse/eclipse.platform.resources/blob/79b63f480a/bundles/org.eclipse.core.resources/src/org/eclipse/core/resources/IResourceChangeListener.java

[19] https://github.com/eclipse/eclipse.platform.resources/blob/79b63f480a/bundles/org.eclipse.core.resources/src/org/eclipse/core/resources/IResourceDeltaVisitor.java

[20] hunt1976algorithm

[21] https://git-scm.com/docs/git-diff

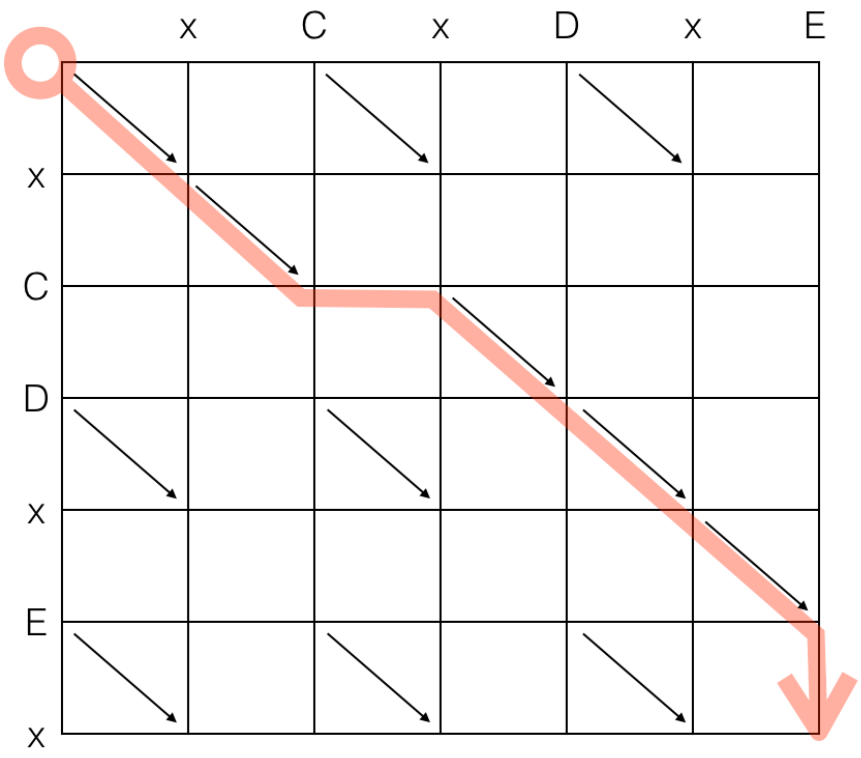

Figure 4.8: Visual graph representation of the difference between *xCxDxE* and *xCDxEx*[22]

the target string. The diff between the two strings is therefore -x on position 3 and +x after position 6.

In WatchDog, instead of using characters in single positions, the textual representation of a static analysis warning is used to compute the diff. For example, the diff from the warnings shown in Figure 4.9 is the removal of the warnings *Unused import java.util.List* and *Duplicate field "Bar"*, while the warning *Unused local variable "local2"* has been introduced. The implicating factor of this approach is the inability to detect unique warning changes.

In Figure 4.10, both listings will produce the warning *Unused parameter "local1"*. However, the original warning of the parameter in *function1* has been resolved, while instead the warning has been introduced on the parameter in *function2*. Since Eclipse produces warnings in snapshots, both snapshots will have the same textual warning. The diffing algorithm will therefore not detect these changes, as a result of not having an API in Eclipse which exposes the required information. This limitation is therefore a direct result of the lack of API, in a similar fashion to the lack of API in IntelliJ, as described in Section 4.1.2.

[22]Source: `https://epxx.co/artigos/diff_en.html`

Listing 4.1: Static analysis warning snapshot before

```
Unused import java.util.List.
Unused import java.util.Set.
Duplicate field "foo".
Duplicate field "Bar".
Unused local variable "local".
```

Listing 4.2: Static analysis warning snapshot after

```
Unused import java.util.Set.
Duplicate field "foo".
Unused local variable "local".
Unused local variable "local2".
```

Figure 4.9: Example diff between two static analysis warning snapshots

Listing 4.3: Initial definition of two Java methods with an unused local variable

```
public static void function1(String local1) {}
public static void function2() {}
```

Listing 4.4: Second definition of two Java methods with an unused local variable

```
public static String function1(String local1) {
    return local1 + local1;
}
public static void function2(String local1) {}
```

Figure 4.10: Example of two method definitions which produce equivalent static analysis warning snapshots

```
Added warning "Unused import java.util.List."
Added warning "Unused import java.util.Set."
Removed warning "Unused import java.util.List."
Added warning "Unused import java.util.List."
Removed warning "Unused import java.util.Set."
```

Figure 4.11: Sequence of warnings generated in IntelliJ

4.3 Lowest common denominator implementation

To eliminate the variable of editor choice in the data analysis, the implementation used in each editor is based on the lowest common denominator between the implementations of the editors. As a result of this approach, the frequency of warnings in IntelliJ is artificially delayed to match the frequency of warnings in Eclipse. In other words: while IntelliJ provides real time updates when a developer types, Eclipse only provides this information after every build, which happens when the developer saves a file. The warnings in IntelliJ are therefore gathered over time and after a file is saved, the snapshot is calculated.

The sequence of generated and removed warnings is stored between file saves. Given the sequence of events in Figure 4.11, the end-result that is processed is 1 generated warning "Unused import java.util.List", since the first "List" and the "Set" warning are both removed before the file was saved. This approach is in line with the fidelity of the API in Eclipse, which will produce one snapshot consisting of "Unused import java.util.List" at the moment of saving the file. The implementation in WatchDog is shown in Figure 4.12. Here, *generatedWarnings* and *removedWarnings* are *Set*s which are processed and cleared after the file is saved. While insertion, lookup and removal are normally $O(N)$ in a *Collection*, the backing implementation in WatchDog is a *HashSet*, which has constant time performance for collection modification and traversal[23].

By using the lowest common denominator, an editor implementation must be adapted to the other editor implementations. In our case, this mostly involved adapting the IntelliJ implementation to maintain equivalent behavior to the Eclipse implementation. To make sure the performance impact is minimal, the in-memory cache is backed by an efficient *Collection*. While the computational time impact is low, this adapter does require additional memory and is therefore not without cost. Moreover, it introduces an extra indirection for developers of WatchDog, which has an impact on the overall maintainability.

Secondly, while IntelliJ allows for high fidelity of static analysis monitoring, we now make explicit use of a less granular approach. This has an impact on the overall utility of the generated data, which could have been more fine-grained if we would only use IntelliJ.

[23] https://docs.oracle.com/javase/8/docs/api/java/util/HashSet.html

[24] https://github.com/JetBrains/intellij-community/blob/7fd2dc58fa/platform/editor-ui-ex/src/com/intellij/openapi/editor/impl/event/MarkupModelListener.java

```
@Override
public void afterAdded(@NotNull RangeHighlighterEx rangeHighlighterEx) {
    this.generatedWarnings.add(rangeHighlighterEx);
}

@Override
public void beforeRemoved(@NotNull RangeHighlighterEx rangeHighlighterEx) {
    // Only process a deletion if we hadn't encountered this marker
    // in this session before. If we did encounter it, remove returns
    // `true` and the warning is not saved as removed.
    if (!this.generatedWarnings.remove(rangeHighlighterEx)) {
        this.removedWarnings.add(rangeHighlighterEx);
    }
}
```

Figure 4.12: Usage of the in-memory cache in the implementation of the callbacks defined by *MarkupModelListener*[24]to generate Eclipse-equivalent snapshots

4.4 Static analysis IDE extensions

While we wanted to obtain a basic set of warnings generated by the IDEs, there are also various other Static Analysis plugins that can report warnings in the IDEs as well. Popular plugins include FindBugs[25], SonarLint[26] and CheckStyle[27]. Based on the popularity of the available IntelliJ plugins[28] (metrics gathered in 2018), we started with an investigation into the CheckStyle plugin. CheckStyle was the most popular plugin with almost 1,5 million downloads, while FindBugs had near 700 thousand downloads and SonarLint nearing half a million.

An important aspect to keep in mind while investigating these plugins is that the IDE plugins (that are accessible by any IDE plugin, such as WatchDog) are different than the actual static analysis projects. In the case of CheckStyle, the CheckStyle Eclipse plugin[29] is in a different package than the core CheckStyle implementation[30]. Moreover, while the Eclipse plugin is maintained by the CheckStyle team, the equivalent IntelliJ plugin[31] is maintained by a community member. Since the plugin is maintained separately and potentially by a different person, it could be possible that not all capabilities of the core implementation are available through the plugin API. However, we are not aware of any concrete problems of the plugins we analyzed.

[25] http://findbugs.sourceforge.net/downloads.html

[26] https://www.sonarlint.org/

[27] http://checkstyle.sourceforge.net/index.html#Active_Tools

[28] http://plugins.jetbrains.com/search?correctionAllowed=true&pr=&orderBy=downloads&tags=Inspection&tags=Tools+integration&search=

[29] https://github.com/checkstyle/eclipse-cs

[30] https://github.com/checkstyle/checkstyle

[31] https://github.com/jshiell/checkstyle-idea/

An initial investigation of the architecture of the CheckStyle plugins uncovered that the IntelliJ plugin dynamically loads the CheckStyle core implementation. Practically speaking, this means that the implementation of CheckStyle is not accessible by WatchDog via the Java classpath. This posed an additional challenge, as obtaining information from CheckStyle became significantly harder. We started debugging and instrumenting CheckStyle in an example project with two source files both generating numerous Static Analysis warnings. The debugging sessions focused on stepping through the implementation and to understand how CheckStyle initializes and loads the logic for its analyses. Alongside the debugging session, reading through the source code of the project was necessary to get an overall view of the project and particular classes to look out for.

It quickly became clear that the implementation of CheckStyle was difficult to grasp from a bird's-eye point-of-view. Inspection of the public API of CheckStyle did not show any possibility for instrumentation of the inner-workings of the check-implementations. (The check-implementations are the concrete instances that contain the logic of determining whether a warning should be issued for the provided code structure.)

Eventually, after stepping through the main-entrypoint (the *Checker*[32] class), the map that we are looking for appeared to exist in the *PackageObjectFactory*[33]. This private *Map* contains for each check that is loaded the location in the jar where the class resides. For every package (in other words sets of similar static analysis checks), there is one *messages.properties* that contains all textual descriptions of the checks. The layout is equivalent to the ones used in the IDEs and can therefore be processed in the same fashion as we did before.

All in all, the debugging session took several days, which led us to the realization that the existing static analysis plugins are not written with monitoring capabilities in mind. Our expectation is that the other plugins such as SonarLint and FindBugs have the same kind of characteristics. The factors of difficult-to-debug implementation combined with the lack of API-capabilities for our use-case resulted in the integration taking a lot longer than expected.

4.5 Summary

Normally the internal implementation is a minor detail (it is a means to obtain the required data), but in this research the availability (or lack thereof) of an API has an impact on the overall outcome. Since there is a discrepancy between the Eclipse and IntelliJ APIs and the lack of exposure of the required objects, the usability of the available data is limited. Moreover, as a result of the discrepancy, the implementation in each editor is based on the lowest common denominator, which reduces the potential fidelity of the data. Lastly, additional work is required for each static analysis IDE extension, to be able to understand its implementation to be able to observe its behavior.

[32] https://github.com/checkstyle/checkstyle/blob/e9fac97cdd/src/main/java/com/puppycrawl/tools/checkstyle/Checker.java

[33] https://github.com/checkstyle/checkstyle/blob/489ce031e8/src/main/java/com/puppycrawl/tools/checkstyle/PackageObjectFactory.java#L97-L98

For future work, adding the necessary API hooks in the editor implementations is a first step to obtain more granular data. However, even though both editor implementations are open-source, improving the APIs requires extra time, as the maintainers of each implementation have to be convinced. Our expectation is that the architecture of the editors allow for these extensions, but the communication with the maintainers will require additional time. For this thesis the additional time is infeasible, but with careful planning it is possible to incorporate this in future work.

RQ1: IDEs do not expose a high-level API to monitor generated static analysis warnings in the editor. Currently, observance of the internal in-memory cache is the only option. Future extensions to the IDEs can improve integration and increase the utility of obtained static analysis warning data.

External static analysis libraries incorporated in IDE extensions are not developed with the use-case of monitoring in mind. While the most popular plugin (CheckStyle) is integrated into WatchDog, several others are not. Integrating other plugins is expected to be time-intensive and requires appropriate exposure of the plugins inner workings to observe their behavior.

RQ1.1: External static analysis plugins and tools are not written with monitoring in mind. Integration requires extensive engineering effort and deep knowledge of the inner workings of the static analysis tools themselves.

Chapter 5

Classification of warnings

Our own analysis of static analysis warnings will be based on real-world data retrieved from developer activities. Static analysis warning messages frequently include references to code. To make sure the privacy of developers is guaranteed, we have to anonymize this data in such a way that it is still useful for us. This chapter describes the necessary steps to perform the anonymization as well as other technical challenges that have an impact on the research results.

5.1 Possible code references

Before we discuss the anonymization process, we first need to introduce examples of privacy-sensitive static analysis warnings. There are two categories of warnings: constant and dynamic warnings. In the case of constant warnings, the warning for a given static analysis category is the same for every occurrence of that warning. Figure 5.1 shows several constant warnings as defined in IntelliJ[1] and Eclipse[2]. Since constant warnings do not contain any user-provided input, processing such warnings would not have an impact on the privacy of developers.

In contrast, dynamic warnings do contain user-provided input. The majority of the warnings defined in IntelliJ and Eclipse contain user-provided input. Both IDEs use similar implementations of constructing a concrete warning based on a dynamic pattern. While IntelliJ[3] relies on the JDK implementation of MessageFormat[4], Eclipse implements its own version of the formatting function[5]. Nonetheless, both implementations are based on the

[1] https://github.com/JetBrains/intellij-community/blob/2444cc9165/platform/platform-resources-en/src/messages/InspectionsBundle.properties

[2] https://github.com/eclipse/eclipse.jdt.core/blob/efc9b650d8590a5670b5897ab6f8c0fb0db2799d/org.eclipse.jdt.core/compiler/org/eclipse/jdt/internal/compiler/problem/messages.properties

[3] https://github.com/JetBrains/intellij-community/blob/a5d8116251/platform/util-rt/src/com/intellij/BundleBase.java#L61-L68

[4] https://docs.oracle.com/javase/9/docs/api/java/text/MessageFormat.html

[5] https://github.com/eclipse/eclipse.jdt.core/blob/efc9b650d8/org.eclipse.jdt.core/compiler/org/eclipse/jdt/internal/compiler/problem/DefaultProblemFactory.java#L138-L204

```
Replace with a null check
Passing <code>null</code> argument to parameter annotated as @NotNull
All implementations of this method are empty

String constant is exceeding the limit of 65535 bytes of UTF8 encoding
Empty block should be documented
Null comparison always yields false: this expression cannot be null
```

Figure 5.1: Constant warnings as defined in IntelliJ and Eclipse

```
{0} name <code>#ref</code> doesn''t match regex ''{1}'' #loc
Method has {0, choice, 1#usage|2#{0,number} usages} but they are not
    reachable from entry points.
<html><body>Duplicate string literal found in<br>{0}</body></html>

The type parameter {0} is hiding the type {1}
The field {0}.{1} is deprecated
Unnecessary cast from {0} to {1}. It is already compatible with the
    argument type {2}
```

Figure 5.2: Dynamic warnings as defined in IntelliJ and Eclipse

same principle: dynamic parts in a pattern are enclosed by {\d+} where \d+ matches 1 or more digits. IntelliJ also supports inline references and code snippets with #loc and <code>#ref</code> respectively. Several dynamic warnings are shown in Figure 5.2.

The possible values that are substituted in the messages include names of classes, methods, types, fields, etcetera. Figure 5.3 shows several code snippets with their corresponding static analysis warning with references to the code in the snippets. While the listed examples are harmless, real-world usages will very likely contain sensitive information. For example, an "Unused field"-warning being generated on a field named *password* can expose crucial security-sensitive credentials. For our research, we are only interested in the occurrence of the "Unused field" and are not interested in the actual content of the warning. Therefore, we have to anonymize these messages to their category, rather than collecting the warnings verbatim.

Listing 5.1: The type parameter Foo is hiding the type Foo

```
public class Foo {}
public class Bar<Foo> {}
```

Listing 5.2: Unnecessary cast from String to String

```
String foo = (String) new String();
```

Listing 5.3: Boolean method 'method' is always inverted

```
public static boolean method() {
    return false;
}

public static void main(String[] args) {
    boolean call1 = !method();
    boolean call2 = !method();
}
```

Figure 5.3: Several code snippets with their corresponding dynamic static analysis warning containing references or names from the code

5.2 Classification of warnings

To be able to anonymize the warnings and categorize the warnings, we have to implement a classification algorithm that, given a concrete static analysis warning, returns the original message pattern this message was based on. The process consists of two steps:

1. Collect all message patterns that can be used to report static analysis warnings.

2. Based on the message patterns, write a classification algorithm that can iterate through the list of patterns and perform matches to determine what the original pattern was.

5.2.1 Collection of message patterns

There are various options for achieving step 1. First of all, we could construct our own patterns based on observations of the warnings generated on our own code. This might seem like a simple (yet time-intensive) solution (as the code we write will generate warnings that other developers more likely encounter as well), but there are actually numerous problems with this approach. It is highly unlikely that we will be able to obtain all existing patterns by writing code ourselves and observe what warnings are generated. There is a very high

chance that the warnings that we encounter are only a small subset of the total set of warnings, for which we do not know beforehand how large this set is. Secondly, even if we would be able to obtain an exhaustive list, we would need to update the list for every new warning we observe. This is maintenance-wise a very intensive task and updating the list used in the plugin would mostly be on an ad-hoc basis.

Instead, we should investigate how IDEs and other plugins load their messages and how we can obtain these messages as well. This means that we need to understand how the IDEs and plugins are implemented and then reconstruct the required minimal functionality for our use-case. This is a significantly more difficult problem, as we are now dependent on understanding the source-code of large-scale IDEs and popular plugins, but if we can make it work we will be more resilient against future changes in the IDEs and plugins. The following sections elaborate on the process of obtaining the message patterns in both the IDEs and an other plugin.

Collecting IDE message patterns

As pointed out in Section 5.1, both IntelliJ and Eclipse define the message patterns in *.properties* fields. These files are loaded using a *ResourceBundle*[6] which is essentially a map of *String* to *String*. Every message pattern has as key a unique value (in the case of IntelliJ a dot-separated *String*, in the case of Eclipse an *int*) and its corresponding value can be retrieved using *getString*. For our collection process, we need to construct these bundles and iterate through all key-value pairs to continue processing.

IntelliJ has two separate bundles with static analysis warning messages. The first bundle is the *messages.InspectionsBundle* bundle, which contains the messages shown in Figure 5.1 and 5.2. Besides this bundle, *com.siyeh.InspectionGadgetsBundle*[7] also contains messages that are used in the *InspectionGadgets* package. It is unclear what the difference between the two packages is, but *InspectionGadgets* defines a large number of inspections in the *com/siyeh* package[8]. We must therefore load both bundles to be able to classify potential static analysis warnings.

Eclipse has only one bundle, however this bundle is localized. This means that loading the bundle for *org/eclipse/jdt/internal/compiler/problem/messages.properties* will not work, as the localization of the IDE is integrated into the file name. In other words, depending on the localization selected by the developer, the location of the bundle is different. Instead of reading the file directly, we have to use an API in Eclipse which exposes the information. To that end, we can use *DefaultProblemFactory.loadMessageTemplates(Locale)*[9] that returns a *HashtableOfInt*[10] which has an almost equivalent API to a *ResourceBundle*.

[6] https://docs.oracle.com/javase/9/docs/api/java/util/ResourceBundle.html

[7] https://github.com/JetBrains/intellij-community/blob/2444cc9165/plugins/InspectionGadgets/InspectionGadgetsAnalysis/src/com/siyeh/InspectionGadgetsBundle.properties

[8] https://github.com/JetBrains/intellij-community/tree/2444cc9165/plugins/InspectionGadgets/src/com/siyeh/ig

[9] https://github.com/eclipse/eclipse.jdt.core/blob/efc9b650d8/org.eclipse.jdt.core/compiler/org/eclipse/jdt/internal/compiler/problem/DefaultProblemFactory.java#L213-L242

[10] https://github.com/eclipse/eclipse.jdt.core/blob/efc9b650d8/org.eclipse.jdt.core/compiler/org/eclipse/jdt/internal/compiler/util/HashtableOfInt.java

Collecting other Static Analysis plugin message patterns

As pointed out in Section 4.4, the dynamic loading of CheckStyle in IntelliJ complicates the logic further, as the classes that we needed to obtain the paths to the resources were not reachable by the same *ClassLoader*. As such, we actually need two separate *ClassLoader*s to first obtain all the classes and then use the second loader to obtain the resources. Complications like having multiple *ClassLoader*s significantly impact the integration of other Static Analysis plugins. For future research, if extending the scope of the field research beyond multiple Static Analysis plugins is desired, development time investments should be carefully weighed against improved analyses. That is why, instead of incorporating multiple plugin classifiers, we focused on just to get CheckStyle to work.

Summary

Based on our experience of collecting the message patterns, it became apparent that integration with IDEs is straightforward. However, integration with other plugins is significantly more difficult to be able to collect the patterns. Especially the storage pattern of the messages (using the standard *messages.properties*) is a factor for integration. If a plugin uses this standard, integration is a lot simpler than if they would use a custom implementation.

5.2.2 Classification algorithm

The classification algorithm should, given a concrete version of a static analysis warning, return the category a warning is from. Based on the collection of message patterns, we have a large list of key-value pairs for which the key is the category and the value the *MessageFormat*-string. We must search this collection efficiently with the concrete instance of the warning to obtain the classification in a performant manner.

Multiple datastructures

As pointed out in Section 5.1, there are two different categories of warnings: constant and dynamic. As a result of this distinction, we can treat warnings of the two categories in two different ways. First of all, the constant warnings can be matched one to one, given the concrete version of the warning. In other words, we simply have to search for the warning that is exactly the same as the concrete version and return the key, rather than having to do any matching procedures. However, for dynamic warnings we can not do that and we are forced to implement a matching procedure.

Since these warnings can be treated differently, storing the two versions separately can result in a performance improvement. All constant warnings can be stored in a hash table such as a *HashMap*[11], which has read access of $O(1)$, to obtain the key for that warning. If the warning is not constant, the value retrieved from the *Map* is empty. Consequently, for dynamic warnings we have to search through a *List* of patterns to match on the warning. As soon as we match on a pattern, we can assume that the warning originates from that pattern.

[11] `https://docs.oracle.com/javase/9/docs/api/java/util/HashMap.html`

Listing 5.4: Two similar patterns based on the messages defined in IntelliJ[12]

```
Processing duplicate property key:.+
Processing duplicate property key: .+
```

Listing 5.5: Two concrete instantiations based on the two message patterns

```
Processing duplicate property key:foo
Processing duplicate property key: foo
```

Figure 5.4: Two patterns with their corresponding concrete instantiations of the warnings

To prevent incorrect matches on substrings of warnings, the *List* can be ordered on most-specific patterns first. An example of two patterns with their corresponding concrete instantiations of the warnings is shown in Figure 5.4. In these examples, we must prevent matching the first pattern on the longer instantiation, as the first pattern is a substring of the second pattern. In practice, we did not observe any concrete issues where messages are substrings of other messages and thus this might only be a theoretical shortcoming.

Putting it all together, the full algorithm is shown in Figure 5.6. Given the correct construction of the constant messages map and the list of patterns, the algorithm retrieves the warning in either data-structure. Creation of these data-structures is shown in Figure 5.5. Note that one nuance had to be introduced regarding the transformation into a regular expression. The presumption of the classification algorithm is that if the warning pattern does not exist in the *dynamicMessages* list, the category is *"unknown"*. However, if the transformation from a message pattern to a regular expression results in a regular expression with an automata that always accepts, this pattern will always match. To prevent misclassifying all warnings (for which the data-structures do not contain a pattern), these regular expressions should not be included in the list. For more information, see Section 5.2.3.

[12]Source: `https://github.com/JetBrains/intellij-community/blob/1c4ecdf108/platform/platform-resources-en/src/messages/InspectionsBundle.properties#L594-L596`

```
Data: A list of warning categories and the corresponding message patterns
Result: The two data-structures storing the warning categories
constantMessages ← Map();
dynamicMessages ← List();
for [category, pattern] ∈ inputList do
    if containsDynamicParts(pattern) then
        regex ← transformPatternIntoRegex(pattern);
        if ¬ (automata from regex always accepts) then
            dynamicMessages.push([regex, category])
        end
    else
        constantMessages[pattern] ← category
    end
end
sortOnPatternLengthDescending(dynamicMessages);
return (constantMessages, dynamicMessages)
```

Figure 5.5: Pseudocode of the data-structure creation algorithm

```
Data: A concrete instantiation of a warning, a map of constant messages and a
      list of tuples of regular expressions and categories of dynamic messages
Result: The corresponding category of the warning
if warning ∈ constantMessages then
    return constantMessages[warning]
end
for [regex, category] ∈ dynamicMessages do
    if regex matches warning then
        return category
    end
end
return "unknown"
```

Figure 5.6: Pseudocode of the classification algorithm

5.2.3 Missing classifications

As shown in the classification algorithm, when a particular message originates from a message pattern that is not included in the data-structures, the category is *"unknown"*. This section lists several examples of message patterns WatchDog currently misses.

Patterns included in IDE plugins

As described in Section 5.2.1, integration of the plugins of community-maintained static analysis tools is time-intensive. If a developer has such a plugin installed and the messages are not processed by WatchDog, the classification will fail. However, not only community-maintained plugins are missing, but also official IDE plugins. One example is the Plugin Development Environment plugin for Eclipse. This plugin defines numerous messages[13] which are used in the development environment built by the Eclipse team.

An example of such a message is the detection of invalid references in the plugin XML-configuration files. These files can reference classes as defined in the development plugin. If the class does not exist, the plugin will issue a warning in the *plugin.xml* file. These warning messages are thus not related to implementation code, but configuration code instead.

Classification of messages that are part of an extension plugin for the IDE are thus missing and have to be integrated for every extension plugin that exists, in a similar fashion for community-maintained plugins.

Patterns as defined in source code

There are also instances of static analysis warning messages being constructred directly in the source code of the IDE or static analysis tool. An example is the *DataFlowInspection* built into IntelliJ, where various dataflow-related messages are directly put into the source code[14].

Detection and/or extraction of these messages can not be automated. Instead, to be able to classify the messages, we would need to construct a curated list of messages manually extracted from the source code. However, as pointed out in Section 5.2.1 this is a maintenance-intensive and error-prone task. The omission of these message patterns is therefore a known limitation without appropriate solution.

Patterns that compile to always-accepting automata

As explained in Section 5.2.2 the data-structure creation algorithm in Figure 5.5 has to filter out patterns that would compile to always-accepting automata. This, to prevent a "catch-all"-situation were all unknown messages would be incorrectly classified under this category. However, this does means that patterns such as the one shown in

[13] https://github.com/eclipse/eclipse.pde.ui/blob/69fd0ac8fe/ui/org.eclipse.pde.core/src/org/eclipse/pde/internal/core/pderesources.properties

[14] https://github.com/JetBrains/intellij-community/blob/3a95eeee9d/java/java-analysis-impl/src/com/intellij/codeInspection/dataFlow/DataFlowInspectionBase.java#L491-L510

Figure 5.7 will not be matched. Currently this pattern compiles to .+ which matches everything. In the future, a more sophisticated transformation algorithm can be used to compile message patterns in the equivalent regular expressions. However, this requires parsing and recursive traversal through the pattern, as there can be an arbitrary deep level of nesting. The current compilation (shown in Figure 5.8) is sufficient for nearly all message patterns with a handful of exceptions. Implementing such a parser and traversal is time-intensive and the benefits will be marginal, yet required to obtain a perfect one-to-one translation to the correct regular expression.

Maintenance of locations of message patterns in IDEs/plugins

While previously stated that manual maintenance of all possible message patterns is infeasible; the automatic generation of patterns is not maintenance-free. It is unlikely the IDEs or plugins change the location of the message pattern files, yet it can happen in the future. This would thus require updating the source code of WatchDog to be able to handle the new location as well. However, maintenance of the location poses a significantly lower cost, as it is merely an update of 1 string in the source code, rather than a large list of message patterns.

Localization of message patterns

One feature of the retrieval of message patterns using *ResourceBundle* is the automatic inclusion of localization. To do so, one can create multiple files with the names including their respective localization. Close inspection of the source code of Eclipse and IntelliJ did not show any actual usage of this localization pattern. Both IDE frameworks show support for the feature, but do not ship the localized properties files. However, CheckStyle does make use of this pattern[15]. WatchDog has been instructed to handle the localized messages, however we have not verified that this works for developers not developing in an English development environment.

Changes during development

Lastly, changes to any of the configuration of both the IDEs and the plugins while WatchDog is loaded are not reflected. This means that changing the locale or version of a plugin does not refresh the internal data-structures of WatchDog used for the classification. The problem is in most cases hypothetical, as IDEs enforce reloading the program after installing an update to a plugin. Yet, the CheckStyle IntelliJ plugin allows for changing the loaded CheckStyle version. This means that any messages generated up to the next time the IDE is started could potentially be misclassified. Again, this would only be the case if the message is added in the newly loaded version or if a message pattern would be updated. Given the very rare occassions and the complexity of implementing this behavior, solving this issue is left for future work.

[15] https://github.com/checkstyle/checkstyle/tree/e018b2d662/src/main/resources/com/puppycrawl/tools/checkstyle

[16] Source: https://github.com/JetBrains/intellij-community/blob/1c4ecdf108/platform/platform-resources-en/src/messages/InspectionsBundle.properties#L283

```
{0, choice, 0#|1#(1 item)|2#({0,number,integer} items)}
```

Figure 5.7: Message pattern that compiles to a regular expression representing an always-accepting automata[16]

```
final String regex = message
    .replaceAll("''", "'")
    .replaceAll("'", "'")
    .replaceAll("\\(", "\\\\(")
    .replaceAll("\\)", "\\\\)")
    .replaceAll("\\[", "\\\\[")
    .replaceAll("<code>[^<]+</code>", "'[^']+'")
    // Perform the next replace twice, as there can be
    // nested brackets. It is performed twice, as there
    // were no messages in Eclipse and IntelliJ that had
    // more than two levels of nesting.
    .replaceAll("\\{[^{}]+}", ".+")
    .replaceAll("\\{[^{}]+}", ".+")
    .replaceAll("\\{}", "\\\\{}")
    .replaceAll(" #loc", "")
    .replaceAll(" #ref", "");
```

Figure 5.8: Transformation of a message pattern into a regular expression built in WatchDog

5.3 Summary

As a result of a lack of an API that exposes the required information, we only have access to the textual static analysis warning messages. To anonymize the data obtained from real-world development activity, we have to classify messages based on their original pattern. This problem consists of two phases: collecting the message patterns and then developing a classification algorithm that can match on these patterns. Collection of messages poses several limitations as a result of the abstraction based on human-friendly textual messages. Overall, the classification is a necessary step to use the real-world data, but it lowers the utility of the obtained data.

Chapter 6

Telemetry results

In this chapter, we will elaborate on how we retrieve the telemetry results, after which we will compute statistics and deduce patterns based on this data.

6.1 Data retrieval

As explained in the previous chapters, the work in WatchDog is the basis for the data collection. We are tracking the following characteristics:

- **Static analysis warning created**

  For every warning that is generated and shown in the user interface, we create 1 event. This events contains information including timestamp, ip address, line number, IDE type, WatchDog user ID, WatchDog project ID, WatchDog session ID, warning category classification (if known) and document information including hashed file name and total number of lines.

- **Static analysis warning removed**

  This event contains all information that a warning creation event contains, with the addition of a time diff. This diff is the number of seconds since its creation time (if known in this session) or -1 if it removes a warning that was already present before opening the file in the IDE.

- **Static analysis warnings file snapshot**

  A snapshot contains a list of warnings that exist in the document at the moment of taking the snapshot. The gathered information is the same as for the warning creation event.

All tracked information is sent periodically, as well as at the moment of closing the IDE, to the WatchDog Ruby server. The Ruby server processes the information and inserts it into the MongoDB database.

Previous data analyses of the data gathered by WatchDog were using an analysis pipeline based on processing Comma Separated Values (CSV) data with R[1] [62]. However, the analysis pipeline was no longer usable for us, as the R scripts were undocumented and not maintained since the previous research project. This meant that the inner workings of the R scripts were unknown to us and figuring out how the analyses were implemented proved to be too difficult. To that end, we wrote a new data analysis pipeline based on the Python Jupyter Notebook[2] [35]. The choice for using Python and Jupyter Notebook is based on the fact that there are numerous widely-used Python packages suitable for Data Science analyses [11]. To visualize data statistics, Matplotlib can be used to generate graphs such as histograms, heatmaps and boxplots [29]. Data manipulation and processing can be done using NumPy to, for example, construct the histogram data needed to visualize with Matplotlib [2].

The Jupyter Notebook contains a list of Python data manipulation and visualization scripts, separated and grouped by a particular data characteristic. As input data, it uses the BSON[3] exports from the MongoDB database. Every day, the server generates a BSON snapshot of all tables in the database. For our analysis, we require both the *user* and *events* table exports. We download the latest available snapshot and then run the Jupyter Notebook scripts, including reading in the data using the Python *bson* package[4].

6.2 Statistics

This section includes all statistics that are computed using the Jupyter Notebook. Each subsection provides a general overview and a deeper analysis of each result.

6.2.1 General statistics and user demographics

The data that we analyzed has been gathered from May 9th 2018 up to June 20th 2018. In this time, we have obtained 61538 static analysis events, of which 37045 were static analysis creation events and 24493 static analysis removal events. Additionally, we have obtained 9689 snapshot events, including a total of 20077 warnings. Of the total of 9689 snapshot events, 3097 events contain zero warnings. Therefore, 68.04% of the opened files contains at least 1 snapshot warnings, with an average of 3.05 unresolved warnings per file.

RQ4.1: 68.04% of the files contain at least 1 unresolved warning, with an average of 3.05 unresolved warnings per file.

[1] https://www.r-project.org/
[2] https://jupyter.org/
[3] http://bsonspec.org/
[4] https://pypi.org/project/bson/

Programming experience	Number of users
> 10 years	3
7-10 years	1
3-6 years	12
1-2 years	12
< 1 year	23
N/A	31

Table 6.1: Number of users for each programming experience category

Programming experience	Number of events
> 10 years	287
7-10 years	3068
3-6 years	1519
1-2 years	8864
< 1 year	7658
N/A	2224

Table 6.2: Number of events for each programming experience category

The events are generated by 81 users with a total observed development time of 2421.15 hours, which amounts to 1.37 years of work based on the average annual hours worked per worker[5]. Table 6.1 contains an overview of the number of users with a particular programmer experience. If a user did not want to provide us this information, their programming experience is N/A. Additionally, the total number of events per programming experience category is shown is in Table 6.2.

6.2.2 Static analysis warning categories

The very first data analysis focuses on the different warning categories and their corresponding frequencies. The analysis relies on the successful classification of the warnings, as described in Chapter 5. In total, 13.25% of all warnings could not be classified. Next to that, there are only 7 observed CheckStyle warnings. Despite being the largest static analysis IDE plugin in terms of installations in the Eclipse and IntelliJ marketplace, we have observed barely any warning. It might be possible that our 81 users all do not use the CheckStyle plugin, albeit the odds of that happening is low. Another explanation could be that the integration with WatchDog is not working as intended. However, extensive testing on our side has not shown such issues. Since we do have 13.25% warnings that could not be classified, it could be that a fraction of them originate from CheckStyle. However, as we have no access to the original messages, we are not able to deduce what is going wrong in the classification process.

[5] https://data.oecd.org/chart/5cEg

RQ2: Despite the large popularity of the CheckStyle IDE plugin, the fraction of observed CheckStyle warnings is close to zero.

Table 6.3 lists the 25 categories with the most frequent warning creation and resolution actions. Based on the corresponding frequencies, we can see that a few categories generate a significant number of events, while other warnings are occurring much less frequently. Consequently, we plot the exponential trend line of the frequencies in Figure 6.1. As expected, the occurrences largely follow an exponential trend, with an R-squared factor of 0.904. Moreover, the most frequent categories are largely related to *type resolution* and *import management*. This indicates that both import statements and the type system are the primary responsibles for the static analysis warnings developers see in their IDE.

RQ2: Warning categories frequencies show an exponential distribution, with the top categories focused on *type resolution* (category indices 1, 2, 6, 7, 8, 11, 12, 16, 19, 21, 22, 23, 24, 25 from Table 6.3), *import management* (categories 4, 5, 13) and *unused declarations/tokens* (categories 9, 14, 15).

If we then plot the percentage of warnings resolved as part of the observed warnings that are created, we obtain the results shown in Figure 6.2. Of the top 25 warning categories, for 23 categories the majority of the warnings in the category are resolved.

One interesting outlier is warning category 6. This warning is actually a compiler error shown in Eclipse whenever a method invocation on an object references to a method that is not defined in that class. Figure 6.3 shows an example of a code snippet that would generate such a warning. Since compilation errors preventing a project from being built, it is unclear why we have very few observed resolutions in this warning category. Our suspicion is that these warnings are generated on methods from objects defined that should be defined in a project dependency that is missing. Resolving these kinds of warnings can be done by reimporting a project to fix the dependency resolution. Reimporting a project will not generate a warning resolution event, as during the process the editor can not open files in the project. This means that WatchDog can not actively monitor the project files and thus miss the resolution event.

Some warnings are resolved almost always, particularly the *unused.assignment* and *local variables is not used* warnings. One scenario we can think of that would result in these warnings to regularly pop up during development is based on the assumption that developers regularly write code top to bottom. For example, given the code in Listing 6.4 being written down top to bottom, the developer would first write the line with the assignment to the *String value*. Afterwards, the developers writes the line with the return statement. Before the developer wrote the second line, the assignment was unused. However, directly after writing the second line, the assignment was used and the warning is "resolved".

In this scenario, while the warning is created and resolved quickly, the developer might not have actively been aware of the warning, as it was simply resolved by writing the second line of code. While technically the assignment was unused in between writing the first and

Index	Warning category	Frequency
1	{0} cannot be resolved to a type	13558
2	{0} is a raw type. References to generic type {1} should be parameterized	6710
3	The serializable class {0} does not declare a static final serialVersionUID field of type long	4747
4	The import {0} is never used	4592
5	The import {0} cannot be resolved	4195
6	The method {1}({2}) is undefined for the type {0}	2457
7	{0} cannot be resolved	2186
8	{0} cannot be resolved to a variable	2067
9	The value of the local variable {0} is not used	1348
10	Syntax error, insert "{0}" to complete {1}	1185
11	Type safety: The method {0}({1}) belongs to the raw type {2}. References to generic type {3} should be parameterized	1102
12	{0} cannot be resolved or is not a field	809
13	unused.import.statement	641
14	The value of the field {0}.{1} is not used	483
15	inspection.unused.assignment.problem.descriptor1	386
16	The method {1}({2}) from the type {0} refers to the missing type {3}	364
17	Resource leak: ”{0}” is never closed	337
18	inspection.javadoc.method.problem.missing.tag.description	333
19	Unhandled exception type {0}	292
20	Syntax error on token "{0}", delete this token	286
21	The method {1}({2}) in the type {0} is not applicable for the arguments ({3})	266
22	Type mismatch: cannot convert from {0} to {1}	246
23	Duplicate local variable {0}	204
24	The type {0} is deprecated	195
25	The method {0}({1}) of type {2} must override or implement a supertype method	194

Table 6.3: The 25 categories with most frequent warning creation and resolution

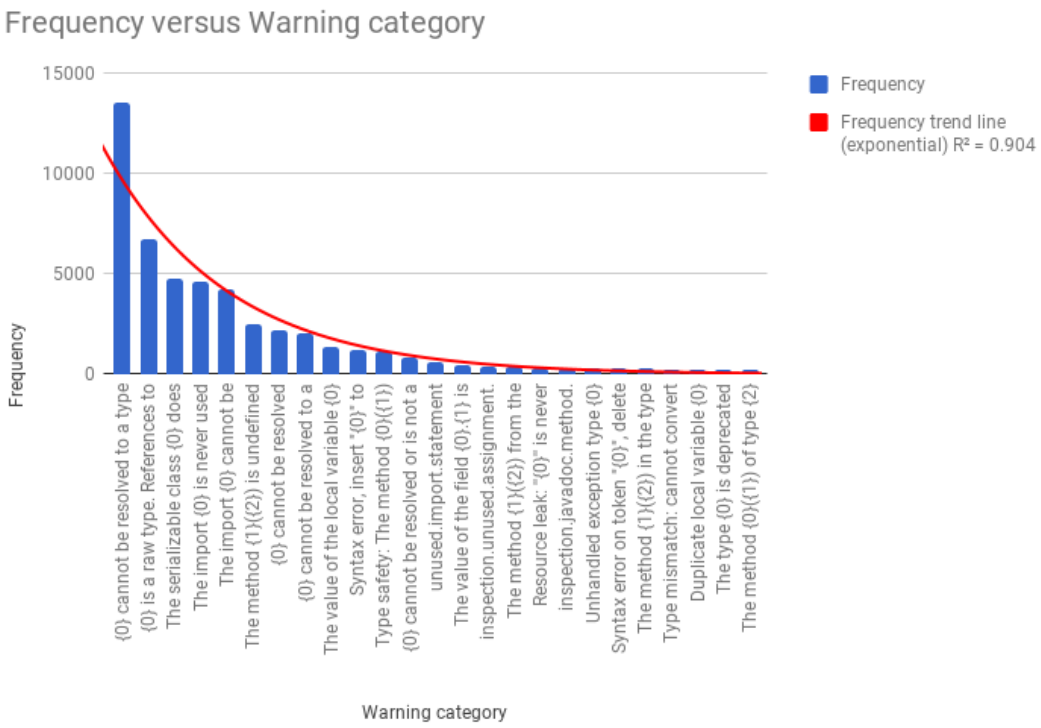

Figure 6.1: Top warning category frequency trend line plot

second line, the developer did not react explicitly to the warning. In terms of usefulness for the developer, this would be equivalent to not generating the warning at all. A high frequency and resolution rate could therefore possibly indicate that the warning category is less useful for the developer.

> **RQ2**: Of the 25 most frequent warning categories, 23 categories are resolved a majority of the time. Possibly, frequent warnings that are (almost) always resolved provide little value to the developer.

Influence of programming experience on the category frequency

If we take programming experience into account for computing the average number of events per user, we end up with the results in Table 6.4. However, the number of users per programming experience is too low (see Table 6.1) for performing a statistical test, as these tests require at least 30 datapoints per category. This is primarily the effect of developers unwilling to disclose their programming experience, as this was optional in the

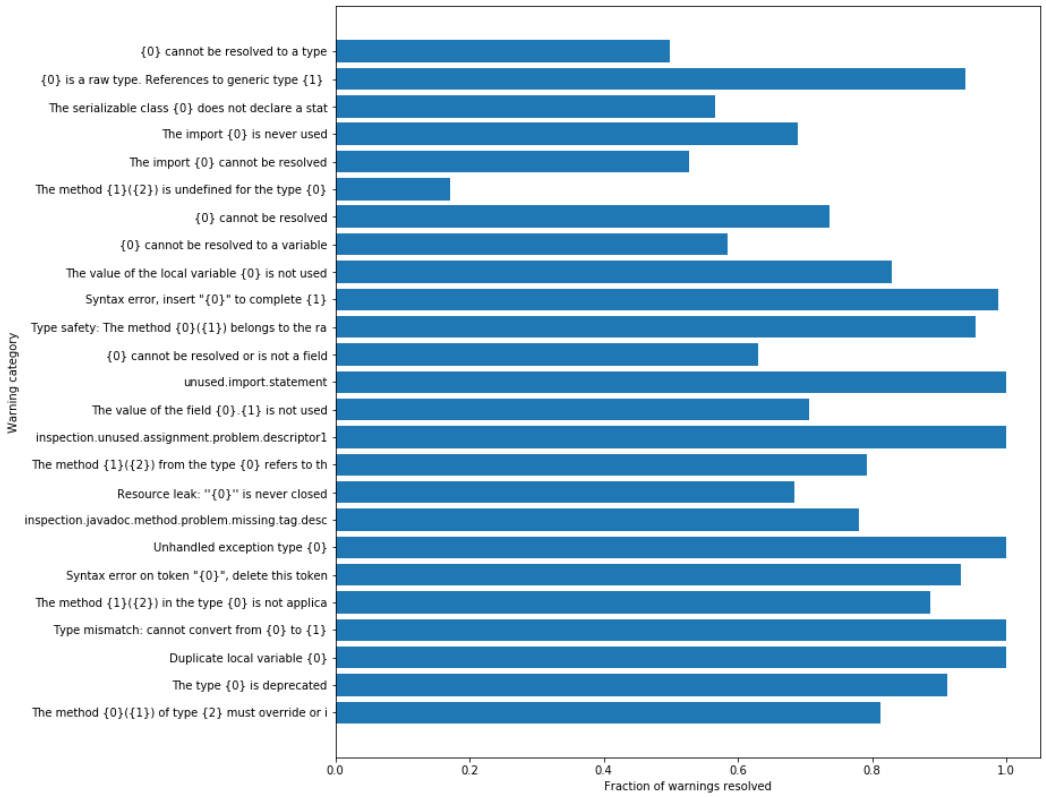

Figure 6.2: Fraction of warnings resolved

```
public class Foo {

        public static void main(String[] args) {
                new Bar().foo();
        }

        class Bar {}
}
```

Figure 6.3: Code snippet that would generate a warning from category 6

```
String value = store.getValue();
return value.substring(value.length() - 1);
```

Figure 6.4: Code snippet that would generate an *unused.assignment* warning on line 1 when writing code top to bottom

Programming experience	Average number of events per user
> 10 years	95.67
7-10 years	3068.0
3-6 years	126.58
1-2 years	738.67
< 1 year	332.96
N/A	71.74

Table 6.4: Average number of events per user grouped by their programming experience

WatchDog user registration. To reach conclusive results, data of more developers who are willing to disclose their programming experience is required.

RQ2.1: We are not able to confirm nor deny that there is a difference in warning frequency when taking into account programming experience, as the number of developers willing to disclose their programming experience is too low.

6.2.3 Lifetime of a warning

The lifetime of a warning is measured for all warnings that are removed when the warning is also created in the same session. Of the 23620 warnings that are resolved, a mere 873 are not created in the same session. This means that 96.44% of the warnings that are resolved are also introduced in the same developer session. For all warnings that are resolved in the same session, Figure 6.5 shows the time distribution in seconds between the 5th and 95th percentile. Here, we can observe that a majority of the warnings are resolved within 1 minute. Moreover, the 25th, 50th and 75th percentile are all within the first 6 minutes, with a long tail of warnings that took longer to resolve. The maximum recorded lifetime of a warning is 246310 seconds, which is not included in the figure for readability's sake. This large number likely means the developer left open their editor for a very long time, probably during the night as well.

RQ3: Of all warnings that are resolved, 96.44% were introduced in the same session by the developer. Most of the warnings are resolved within 6 minutes, with a median of 50 seconds.

Lifetime of the most frequent categories

Categorizing the lifetime for each type of warning results in the time distributions shown in Figure 6.7. Based on these results, we can conclude that the lifetime of a particular warning category can be very different per category. Most of the warnings are resolved quickly, but

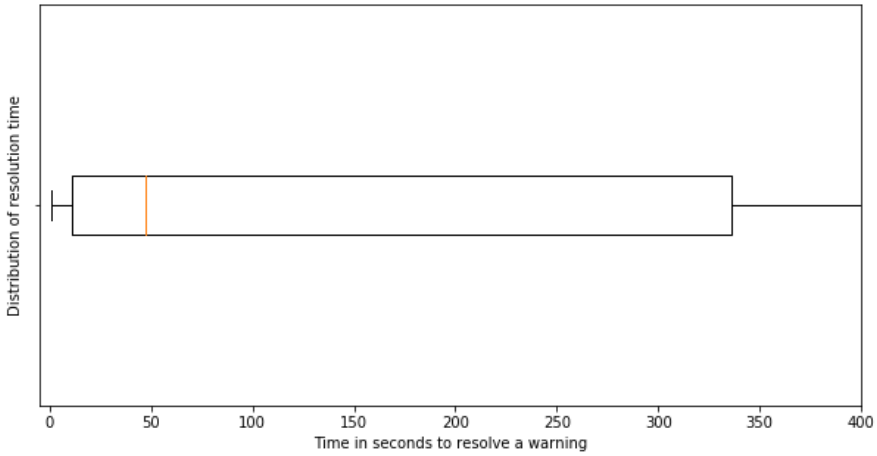

Figure 6.5: Lifetime distribution for warnings resolved in the same developer session

there are instances where warnings regularly remain for a long period of time. Examples include missing Javadoc tags, fields being unused or potential NullPointerExceptions.

When we cluster the lifetime of the top categories focused on *type resolution*, *unused declarations/tokens* and *import management* as listed in Section 6.2.2, we obtain the lifetime distributions shown in Figure 6.6. Similarly to the number of categories related to *type resolution* and *import management*, these high level categories are also resolved much quicker than the warnings from *unused declarations/tokens*. Overall, warnings related to types are resolved the quickest, which is probably related to type checking errors preventing a project being built. Developers are therefore incentivized to resolve such warnings quicker, as the application can not be run while they are present.

One other interesting category is the *unused.import.statement* category. This category is resolved all the time (see Figure 6.5), but the time it takes to resolve can be extremely long (see Figure 6.7). One possible explanation could be that, while the warning is important for developers, they only resolve them at a set point in time. For example, the developer resolves such warnings just before creating a *git commit*. It is possible to use this resolution strategy, because a program can run fine with extraneous imports, in contrast to the type checking issues. However, for code hygiene purposes, developers are inclined to resolve the unused import warnings before they publish their changes.

RQ3: The lifetime of a warning is dependent on the warning category. Warnings related to *type resolution* and *import management* are resolved the quickest.

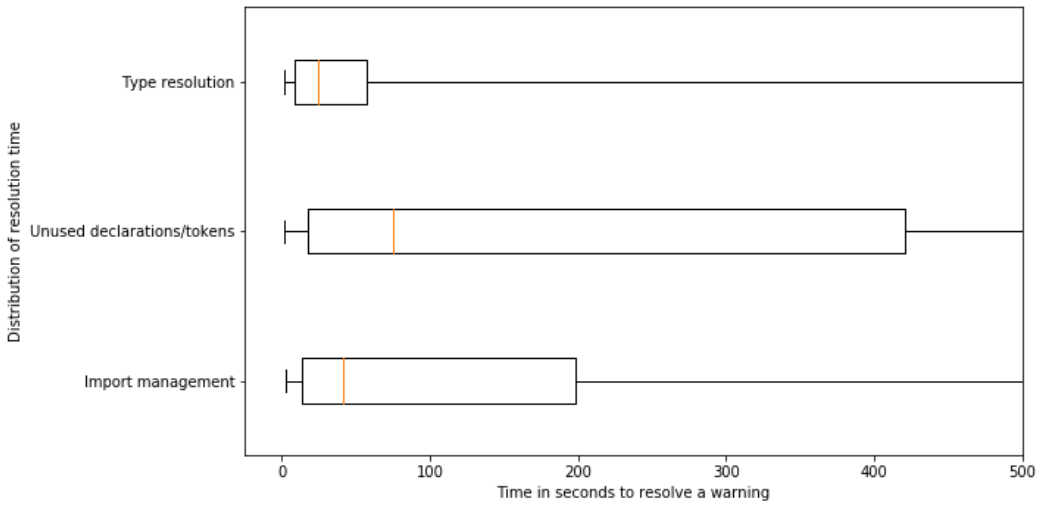

Figure 6.6: Lifetime distribution for warnings clustered by top categories as defined in Section 6.2.2

Influence of programming experience on the lifetime

Even though few users provided their programming experience, we have obtained large numbers of events, which allows us to perform statistical analyses on the dataset. Figure 6.8 shows the lifetime distribution for each programming experience subset. However, a prerequisite for many of the statistical tests is that the dataset is normally distributed. Performing the D'Agostino and Pearson's omnibus test for normality [20] showed none of the subsets nor the overall dataset had a normal distribution, as the p-values approximated or were equal to zero. Moreover, since the events are from different observations, correlation tests can not be performed unless sampling is used. Even then, the amount of events and the mode of sampling would introduced discrepancies in the comparisons.

One of the few tests that can be performed is the Kruskal-Wallis H-test [38]. This test can be used to compare the median of different populations where the null-hypothesis states the population median are equal. Calculating the Kruskal-Wallis H statistic for the programming experience subsets, disregarding the subset N/A, results in a H statistic of 2625.49 and a p-value approximating zero. We can therefore reject the null-hypothesis and conclude that the median of the populations are statistically significantly different. However, we can not conclude how they are related nor what the correlation is between the lifetime and the years programming experience.

To be able to differentiate between the sub-groups, we have to run a post-hoc test. We can use the Dunn-test to investigate which sub-groups are (dis-)similar [22]. The null hypothesis of the Dunn-test is "that the probability of observing a randomly selected value from the first group that is larger than a randomly selected value from the second group equals one half" - Alexis Dinno, author of the R package *dunn.test* [6].

[6] https://cran.r-project.org/web/packages/dunn.test/dunn.test.pdf

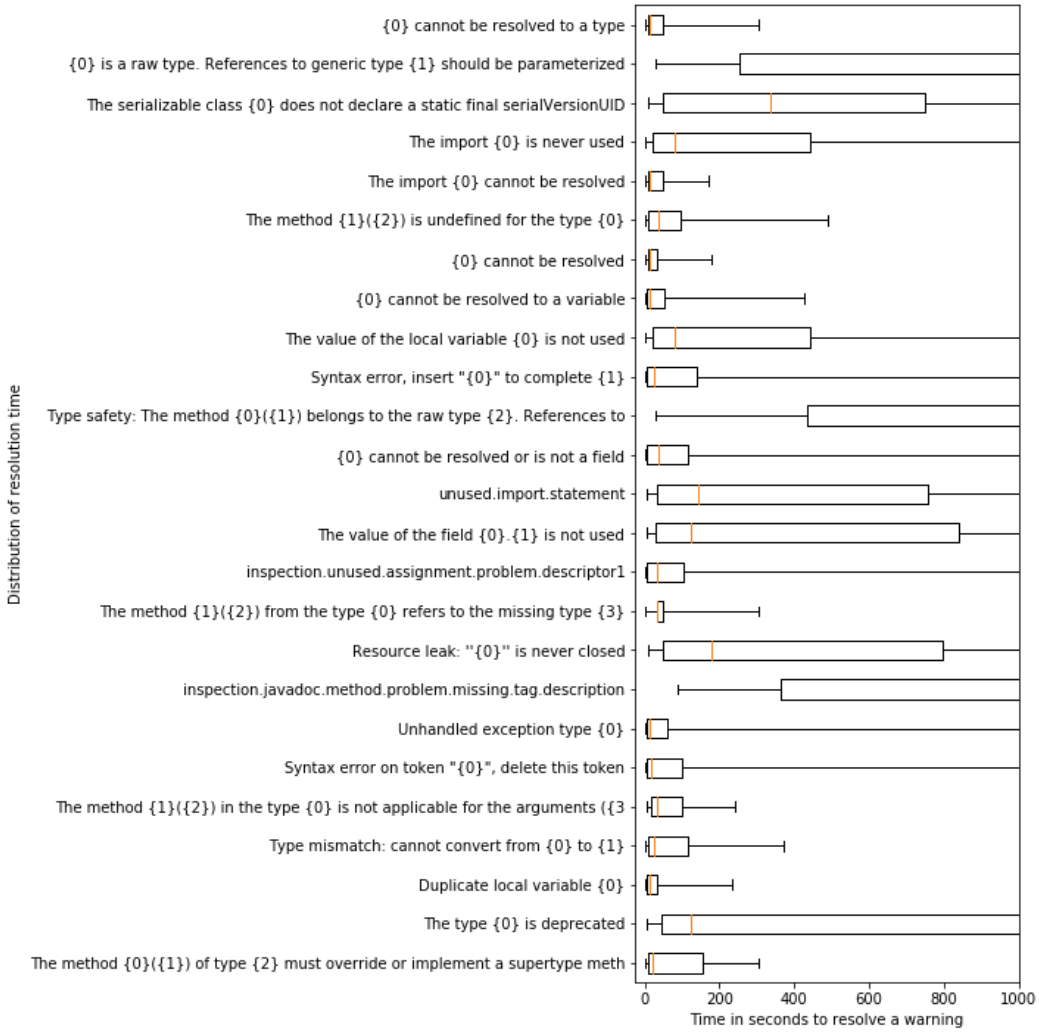

Figure 6.7: Lifetime of the 25 most frequent warning categories

	7-10 years	3-6 years	1-2 years	<1 years
>10 years	0.00274	0.00000	0.01027	0.15841
7-10 years	-	0.00000	0.13906	0.00000
3-6 years	-	-	0.00000	0.00000
1-2 years	-	-	-	0.00001

Table 6.5: p-value from the Dunn-test for each pairwise combination of programming experience subset

Table 6.5 shows, for each pairwise combination of programming experience subset, the p-value outcome of the Dunn-test. For all p-values <0.025, we can reject the null hypothesis and thus conclude that the subsets are dissimilar. Therefore, subsets 1-2 years and 7-10 years are similar as well as the subsets <1 years and >10 years. There is no correlation between the programming experience and whether two subsets are similar. If there would be, we would expect subsets close to each other (e.g. 7-10 years and 3-6 years) to be similar and subsets with a larger discrepancy in terms of years to be more dissimilar.

RQ3.1: We can conclude that the different programming experience subsets do not have a median equal for all subsets. A post-hoc analysis based on the Dunn-test shows that two combinations of subsets are similar, but there is no correlation with the years of programming experience and the lifetime.

Warning frequency versus lifetime

Lastly, we are interested in the effect of the warning frequency on the lifetime of a warning. An ordered list of the developers with the most number of events and their corresponding distribution of the lifetime is shown in Figure 6.9. Based on the boxplot distributions depicted in the Figure, we suspect that there is no correlation between warning frequency and lifetime.

RQ3.2: Warning frequency does not appear to have a significant impact on the lifetime of a warning.

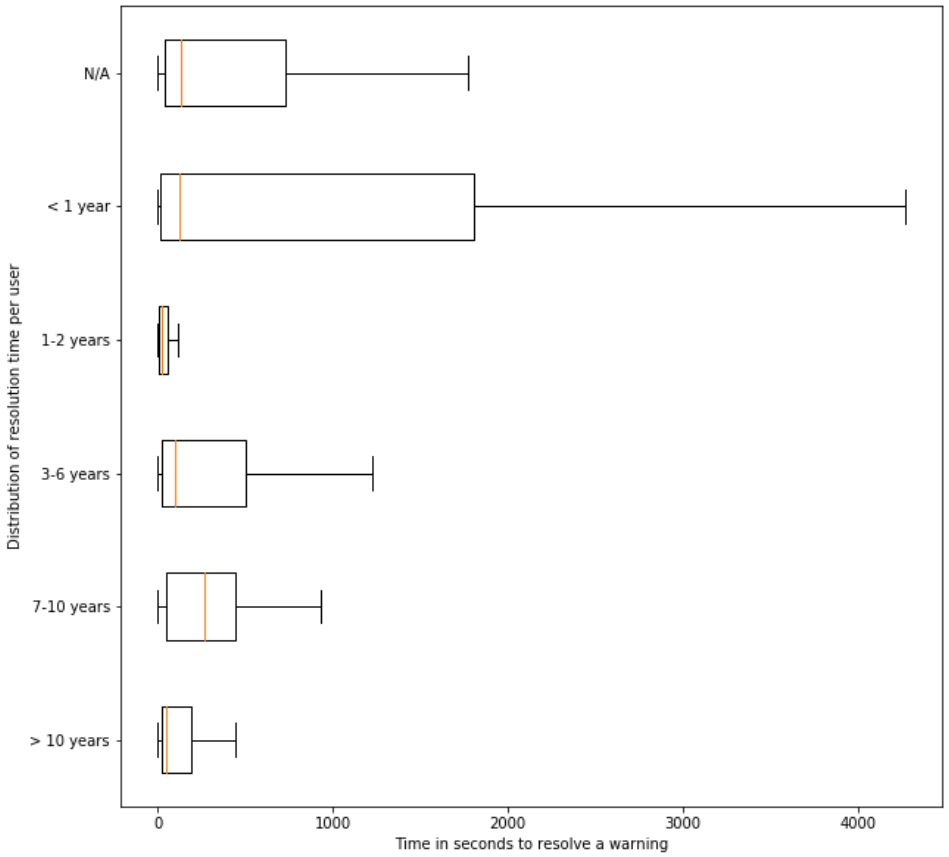

Figure 6.8: Lifetime grouped by programming experience

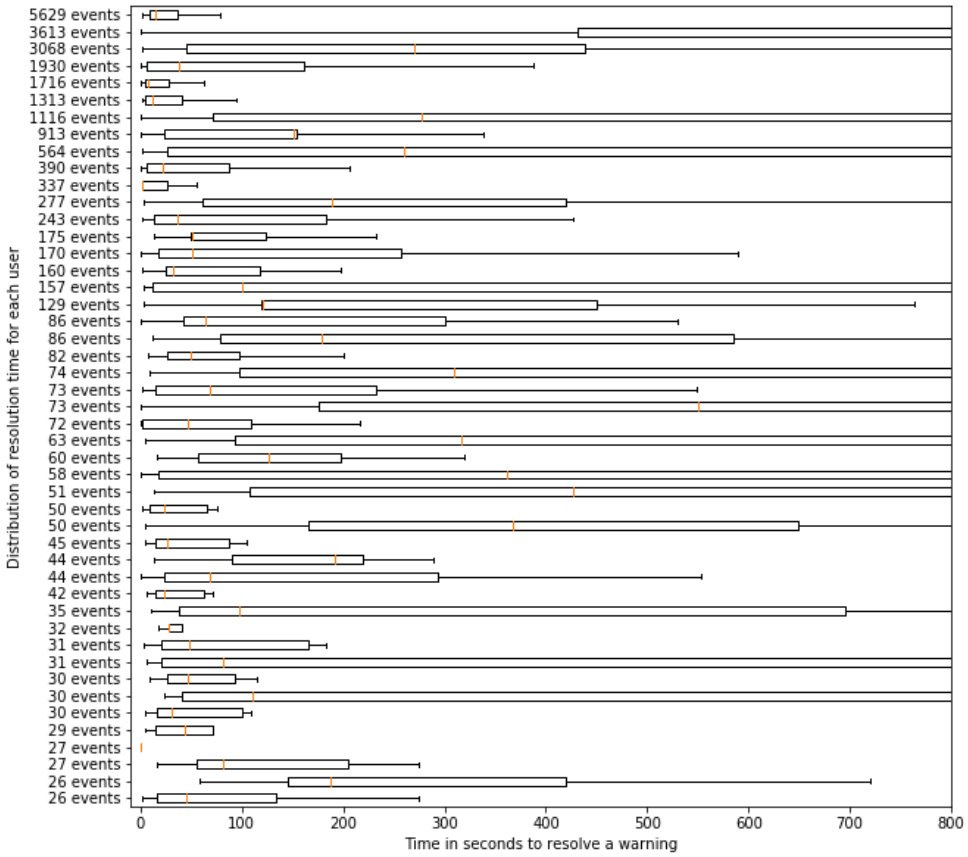

Figure 6.9: Lifetime for the users with the most number of events

6.2.4 Position in file and frequency

The position of a warning in a file is measured at the moment the warning is created or removed. At that time, we calculate both the line the warning is on, as well as the total length of the document. Additionally, for all file snapshots, the line number is stored for each warning and the total length of the document is stored once per snapshot.

The heatmap in Figure 6.11 is based on all created and resolved warnings, while the heatmap based on the snapshots is shown in Figure 6.12. Since snapshots contain all warnings that exist when a file is opened, we can therefore draw the conclusion that these warnings remained unresolved in a previous development session. As such, the top heatmap contains data about developer activity regarding (resolved) warnings, while the bottom heatmap is about potentially stale (unresolved) warnings.

An initial observation based on this distinction is the difference in occurrence frequency at the start and end of a file. There is almost no development activity in these parts of a file, while the snapshots show a non-trivial amount of warnings exist there.

> **RQ4**: There are little to no occurrences of warning creation and resolution at the direct start and end of a file.

There also appears to be a high frequency of warnings in both unresolved warnings in sections in the range of 0.35 and 0.5 of a file.

> **RQ4** and **RQ4.1**: The sections of a file with the highest frequency of unresolved warnings are in the range of 0.35 and 0.5 relative to the file length.

Based on this range, we had the suspicion that these sections include the class declaration and/or field declarations in Java files. To confirm or deny this hypothesis, we performed an analysis on the position of the class declaration relative to the file length of a Java file. The Bash script to obtain the relative position of a class declaration in a Java file is shown in Figure 6.10. We ran this script on open source projects including: Mockito[7], JUnit5[8], JUnit4[9], ElasticSearch[10], JPacman-framework[11], WatchDog[12] and RxJava[13]. The script analyzed a total of 10007 Java files producing the heatmap shown in Figure 6.13. When comparing to the warnings in the snapshots to the class declarations, we see that they both have a high frequency of occurrence in the same sections. Moreover, the sections in the range of 0.15 and 0.25 frequently have the class declaration as well. Warnings in Java files could therefore be likely either directly on the class declaration or shortly thereafter.

[7] https://github.com/mockito/mockito
[8] https://github.com/junit-team/junit5
[9] https://github.com/junit-team/junit4
[10] https://github.com/elastic/elasticsearch
[11] https://github.com/SERG-Delft/jpacman-framework
[12] https://github.com/TestRoots/WatchDog
[13] https://github.com/ReactiveX/RxJava

```
#!/usr/bin/env bash
grep -nrE "^((public|protected|private) )?class " --include=*.java . \
  | while read -r line ; do
    fileName=$(echo $line | grep -ohP ".+:" | cut -d : -f 1)
    lineNumber=$(echo $line | grep -ohP ":(\d+):" | cut -d : -f 2)
    numberOfLinesInFile=$(cat $fileName | wc -l)
    relativePosition=$(bc <<<"scale=2; $lineNumber / $numberOfLinesInFile")
    printf $relativePosition,
done
# Force newline for easier copy-pasting into Python histogram computation
echo
```

Figure 6.10: Bash script to obtain relative position of class declaration in Java files

RQ4 and **RQ4.1**: There is a correlation between the relative position of a Java class declaration and the position of unresolved warnings in a file.

6.3 Summary

To be able to analyze the telemetry data, we wrote a new data analysis pipeline in Python. We have found that there is an exponential distribution in terms of frequency for the static analysis warning categories. Of the top 25 categories, most of the categories are related to *type resolution*, *unused declarations/tokens* and *import management*. We could not confirm nor deny an effect of the programming experience of the developer on the frequency of categories. Warnings are resolved fairly quickly, with a median of 50 seconds, with warnings related to *type resolution* are resolved the quickest. Programming experience and warning frequency appear to be uncorrelated to the expected lifetime of a warning. Lastly, we also found a correlation between the position of a Java class declaration in the file and the positions of the highest frequencies of unresolved warnings in a file.

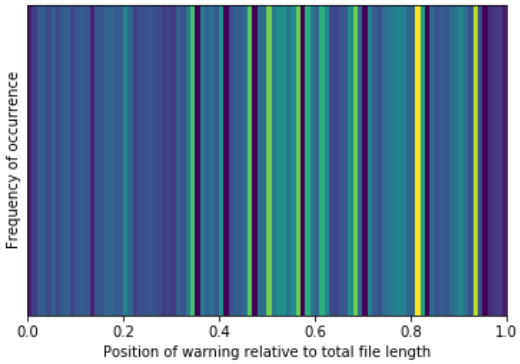

Figure 6.11: Heatmap of all created and removed warnings relative to the file length

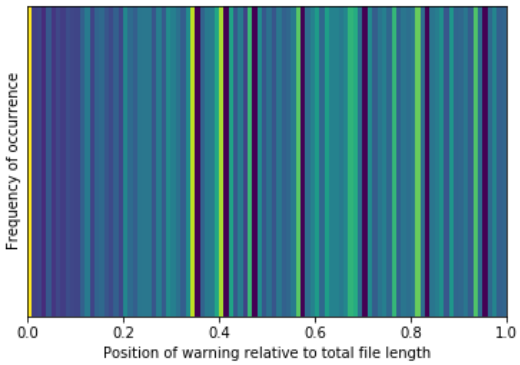

Figure 6.12: Heatmap of all warnings in file snapshots, relative to the file length

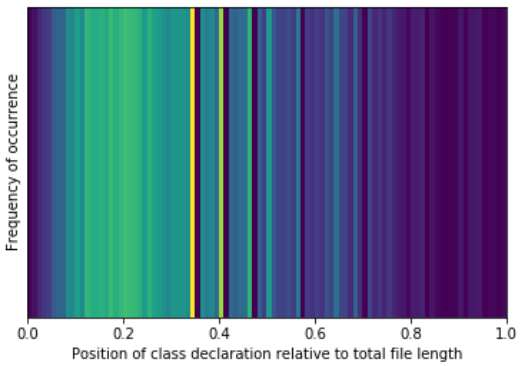

Figure 6.13: Heatmap of all class declarations, relative to the Java file length

Chapter 7

Developer perception

To complement the quantitative data we obtained from the field research study, we conducted a survey among software developers regarding their opinion and experiences on static analysis tools.

7.1 Survey design

The survey focused on several topics where each section was a separate page of the survey and focused on a specific topic. First of, general characteristics of the developer were requested, including programming experience, whether they were also part of the field research study and the company they are employed. Next to that, developers were asked in which cases they use static analysis tools. The third section contained questions regarding resolution of and strategy of resolution for static analysis warnings. Questions regarding the configuration of tools were part of the fourth section. Lastly, several statements were given and the developers were asked to provide their (dis-)agreement. The last page also included a field to list static analysis tools they used (usable as guidance for future work) and a general remark input field.

There was one branching point in the survey before the first section, as we asked developers if they were a registered WatchDog user. If a developer responded yes, they were asked for the user and project identifiers. If the answer was no, several of the characteristics that are included in the WatchDog user registration were asked in this survey. Using these characteristics, we can potentially cross-correlate the quantitative results with the qualitative answers of these developers.

The survey was filled in several times by us, to ensure the flow and branching point was correct. After launching the major update of WatchDog, we also published our survey on the internet. To that end, we sent out 4 tweets spanning over separate days, as well as a Facebook and a LinkedIn post. Next to that, we have spread the survey link in multiple group chats of Bachelor and Master Computer Science students from Delft University of Technology as well as a group chat of one Google team. Lastly, we have asked several of our company contacts to spread the survey in their respective company.

7.1.1 Open question classification

Our survey contains one question regarding the usage of static analysis tools. To be able to classify the tools, we used the open card sorting technique as previously used by Spruit which they also used to classify open ended survey questions [61, 62]. At first, all answers were normalized such that case-sensitivity and general wording was uniform across all answers. For example, *IntelliJ*, *Intellij*, *Intellij IDEA* and *intellij* were all normalized to the official name *IntelliJ Idea*. Then, we grouped the tools after which we formulated a classification name. To improve the validity of this classification, we asked an external student to, given the pre-defined categories, perform another classification. The given task was to classify each tool in the pre-defined category, after which we discussed any potential discrepancies.

7.2 Survey results

Based on the survey responses, we will now dive deeper in the results.

7.2.1 Respondent demographics and general statistics

In total, 61 developers responded to our survey, of which none were also part of our field research study. Of all respondents, 22 developers (36%) listed their company, of which 7 are employed at Google and 3 were student or staff member at Delft University of Technology. Other companies include LinkedIn, ASML and Oracle. As expected by our dissemination strategy, there are multiple respondents from the same company from the companies for which we posted the survey link in the chat rooms, rather than the generic social media posts. This effect is largely in-line with previous research on response rates of web-based research surveys, where personalization and having personal contact with a respondent results in overall higher response rates [19].

A majority of the respondents has more than 3 years experience in software development, 47.5% has 3-6 years, 23.0% 7-10 years and 24.6% >10 years programming experience. 1.6% has less than 1 year experience and 3.3% 1-2 years.

An overview of all static analysis tools that the respondents provided to the open question is shown in Table 7.1. A total of 68.9% respondents provided at least one static analysis tool. The tools were classified as described in Section 7.1.1. Overall, we had the same classifications, but there were 3 points of discussion:

- **Resharper**:

  Resharper is an extension to the Visual Studio Code IDE. To that end, the external student did not classify Resharper as an IDE, but as a Functional Bug Finder. Together, we consulted the documentation of Resharper and concluded that Resharper not only featured a bug finder, but also code editing techniques commonly found in an IDE. We thus resolved this classification by concluding Resharper was more an IDE, based on the fact that its features were not limited to only finding functional bugs.

- **Detekt**:

  The external student initially classified this as a tool collection. This was based on a search using a search engine for which the first result was a Wikipedia page stating the tool was deprecated[1]. However, extending our search query by appending "static analysis" provided the first result as the static analysis tool for Kotlin[2]. Based on the other answers provided and the context of the open question, we resolved this classification as the latter rather than the former.

- **SonarQube**:

  Lastly, we had a discussion regarding the meaning of *tool collection* and *Functional Bug Finder*. We classified SonarQube as a tool collection, while the external student classified SonarQube as a bug finder. The definition of a tool collection was initially unclear to the student. Since SonarQube contained not only a bug finder but also other quality metrics, we resolved this conflict by concluding that the most appropriate classification was the most general one, in this case the tool collection. Classifying SonarQube as just a bug finder would disregard the other features.

Of the 41 respondents, 25 (60,98%) respondents indicated they use at least 1 linter. We also asked developers whether their project uses static analysis tools to enforce a uniform code style, which is one of the purposes of a linter, for which 78.7% indicated they do. Similarly, 21,95% of the respondents listed a functional bug finder, whereas 63.9% answered the closed question that they use static analysis tools to find functional bugs. While the percentages to the closed question are very similar, we do not know where the discrepancy for the open question answers comes from. Since not all participants answered the open question, we attribute the difference to the willingness to answer a closed yes/no-question rather than an open-ended question.

Lastly, only 34.15% of the respondents answered that they use an IDE as static analysis tool. We were surprised by the low percentage of respondents, as we expect developers to make use of an IDE more often. One potential explanation is that developers did not consider the built-in static analysis tools of an IDE when they answered the question.

RQ5: Most developers use a linter to enforce a uniform code style. Functional bug finders and tool collections are used less often. Only 34.15% explicitly mentioned the use of an IDE for static analysis purposes.

7.2.2 Resolving static analysis warnings

Respondents were asked for their estimate of how long it would take them to resolve one static analysis warning. The time-scale was in seconds, minutes, hours, days and more than a week. Of all respondents, 88.5% expects warnings to resolve in less than an hour: 39.3%

[1] https://en.wikipedia.org/wiki/Detekt

[2] https://github.com/arturbosch/detekt

Tool	Nr. of respondents using	Classification
ESLint	10	Linter
CheckStyle	9	Linter
IntelliJ Idea	8	IDE
SonarQube	7	Tool collection
TSLint	7	Linter
FindBugs	6	Functional Bug Finder
TypeScript Compiler	6	Compiler
PMD	4	Functional Bug Finder
Polymer Linter	4	Linter
Closure Compiler	3	Compiler
PHPStorm	2	IDE
PyCharm	2	IDE
ReSharper	2	IDE
SonarLint	2	Linter
SpotBugs	2	Functional Bug Finder
Brakeman	1	Security Vulnerability Finder
Clang Analyzer	1	Functional Bug Finder
Clang-format	1	Formatter
Rust-Clippy	1	Linter
Codesniffer	1	Linter
Coverity	1	Security Vulnerability Finder
CPPCheck	1	Functional Bug Finder
Detekt	1	Linter
Eclipse IDE	1	IDE
Gofmt	1	Formatter
GoLint	1	Linter
Haskell Compiler	1	Compiler
In-house tools	1	Tool collection
KTLint	1	Linter
PEP8	1	Style guide
PyLint	1	Linter
Robocop	1	Linter
Rustfmt	1	Formatter
ScalaStyle	1	Linter
Visual Studio Code	1	IDE

Table 7.1: Tools used by respondents and their classification

estimates resolution takes a couple of seconds and 49.2% a couple of minutes. The amount of respondents answering a couple of hours and more than a week was equivalent: 4.9%. Lastly, a couple of days is only estimated by 1.6%. Overall, the estimations of developers is largely in-line with the time distributions from WatchDog (Section 6.2.3).

> **RQ5.1**: In correspondence to the quantitative data, most developers also estimate warning resolution takes less than an hour.

The interesting distribution in the resolution times is that developers rarely estimate resolution takes a couple of days. As such, developers either expect to resolve warnings on the same day or it would take them weeks. Our quantitative data (Section 6.2.3) once again confirms this estimation, where in a majority of the cases warnings are resolved in the same developer session.

> **RQ5.1**: Most of the time, developers estimate resolution of warnings happens within a day, likely in the same developer session.

When asked about the how often the developer uses the tooling supplied by an IDE to automatically resolve static analysis warnings, the responses were more mixed. 19.7% responded always, 42.6% often and 14.8% sometimes, 11.5% seldom and 8.2% never. 2 respondents (3.3%) indicated that they were not aware of the functionality being available to them in the IDE.

> **RQ5.1**: Almost all developers were aware of automatic refactoring techniques integrated in an IDE and a majority of the developers uses these techniques often.

Based on the resolution, we also asked developers how often they need to look up a static analysis warning message on the internet for clarification. 4.9% indicated always, 11.5% often, 39.3% sometimes, 42.6% seldom and the remaining 1.6% never. In general, developers appear to be able to resolve warnings on their own without consulting the internet for additional help. Moreover, the chosen answers are roughly the same across the different programming experience subsets. This means that most warnings are resolvable either based on the text or based on experience of previously resolving a particular warning. However, most developers rely on the internet for additional help from time to time.

> **RQ5.1**: Most developer resolve static analysis in a majority of the cases without consulting internet for clarification, irrespective of their programming experience. However, from time to time they require external help to resolve a warning.

1	(The first result of) a search engine
2	Question-Answer websites (e.g., StackOverflow)
3	Static analysis tool documentation
4	Static analysis tool issue tracker/mailing list
5	Static analysis tool source code.

Table 7.2: Resources listed that can be used to resolve warnings

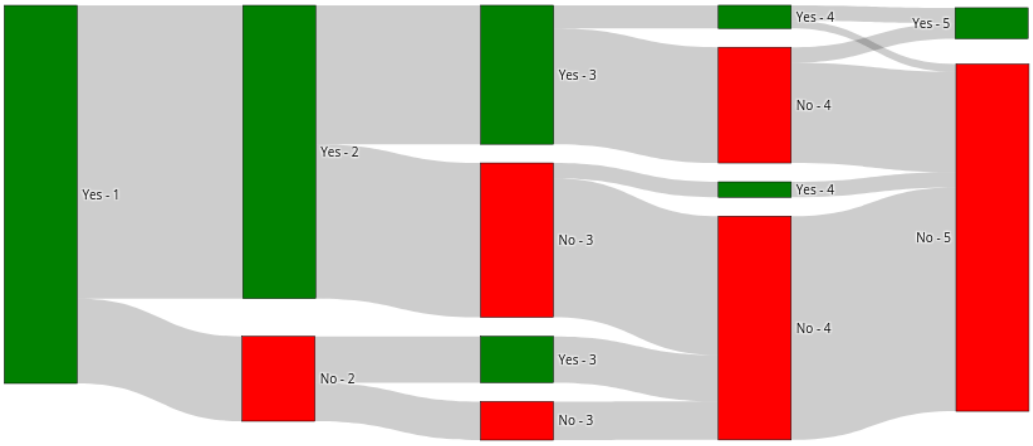

Figure 7.1: Fractions of chosen resource combinations. Indices are listed in Table 7.2

		Relies on Question-Answer websites	
		Yes	No
Relies on documentation	Yes	2	4
	No	3	0

Table 7.3: Combinations of using Question-Answer websites and/or using Static analysis tool documentation for developers not relying on a search engine

As a continuation of the previous question, we also asked developers for a list of resources whether they consult them when resolving a warning. The listed resources are shown Table 7.2. For all 49 (84.5%) respondents that answered they are using "(The first result of) a search engine", the chosen resource combinations are displayed in a Sankey diagram in Figure 7.1. The diagram shows the fractions of respondents that answered yes/no in a green/red flow respectively. Moreover, the inter-dependencies between choices can be deduced, as for example everyone who answered no to resources 3 and 4 also answered no to resource 5. For those that do not rely on a search engine, there were only two tools that were chosen: Question-Answer websites and Static analysis tool documentation. This indicates that issue trackers and source code are not consulted without also using a search engine. Table 7.3 shows these four chosen combinations. In total, 3 respondents did not choose any of the options, presumably not using any of the provided resources.

The distribution of yes/no-answers when the respondent indicated they use a search engine, shows that mailing lists and source code are rarely consulted. Moreover, if one of the two resources is consulted, the previous options of Question-Answer websites and tool documentation are consulted as well. None of the developers that does not consult a search engine relied on the mailing list and source code.

> **RQ5.1**: Developers heavily rely on search engine results, Question-Answer websites and static analysis tools documentation. Question-Answer websites and documentation are rarely consulted without a search engine. Mailing lists and source code are usually consulted when other resources are also consulted.

7.2.3 Configuring static analysis tools

Besides resolution, configuration of static analysis tools can also be used to tune and hide static analysis warnings. Firstly, we asked respondents what kind of configuration options they employ. Half of the respondents (50.8%) answered they use the default configuration with small updates, 29.5% the default configuration and 19.7% a custom defined configuration. This finding is similar to the findings from Beller et al., whom found commonly only one rule in the configuration files of open source repositories has been changed compared to the default configuration [8].

> **RQ5**: Developers say they use the default configuration of tooling or a slightly-customized version. This complements previous findings on tool configuration files of open source repositories performed by Beller et al..

Then we asked more specifically how individual warnings are ignored. The options we gave the respondents are listed in Table 7.4. The corresponding answer distributions are shown in Figure 7.2. It appears that half of the developers never change their IDE configuration. Moreover, only 8 developers indicated they change their IDE configuration (more than) often. In contrast, more than half of the developers answered they sometimes (or more frequently) alter source code to ignore warnings.

> **RQ5**: Developers change their IDE configuration rarely and resort more often to alter source code to ignore warnings.

Since source code alterations are applied by a large majority of the developers, we were also interested in their experience with false positives/negatives. We asked developers "How often do you encounter a false positive/negative reported by a static analysis tool?" with time-scale options of "1 in {10, 100, 1000, >1000}" or never.

Responses were largely uniform, with "1 in 100" being chosen more often and "never" by only 1 developer: 23% every 10 warnings, 34.4% every 100, 24.6% every 1000 and

1	In the source code (e.g. *@SuppressWarnings*)
2	Remove checks from the project configuration
3	Remove checks from the IDE configuration

Table 7.4: Options listed that can be used to ignore warnings

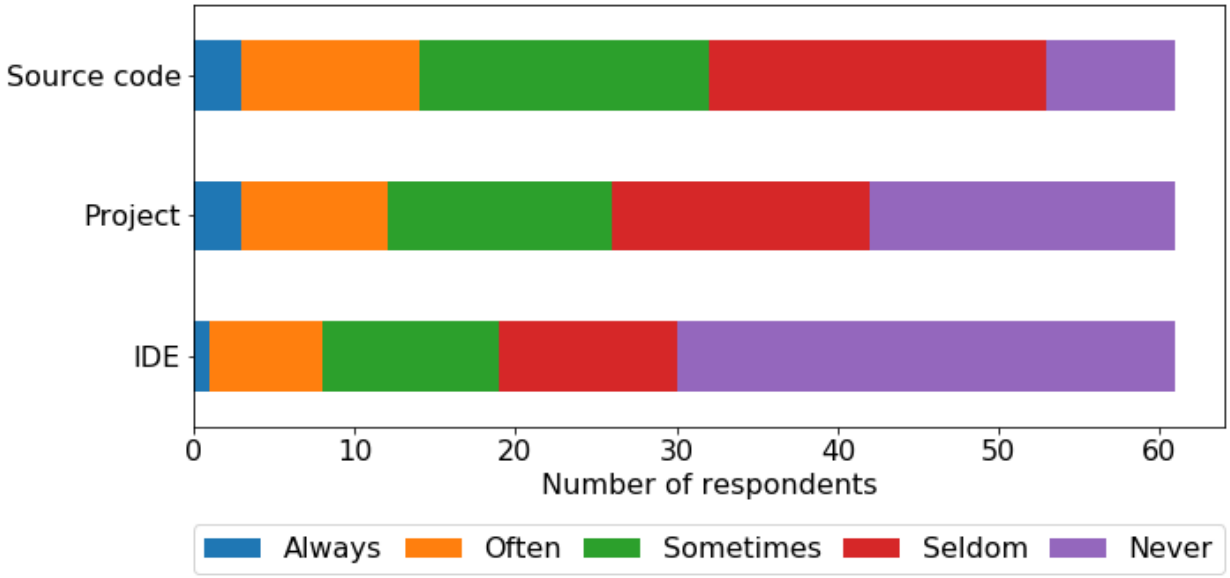

Figure 7.2: Distribution of respondents on how often they use a method to ignore a warning. Indices are listed in Table 7.4

16.4% once in >1000 warnings. Almost all developers therefore regularly and often have to deal with inaccuracies of static analysis tools. One respondent also pointed out that they even encounter false positives more often than 1 in 10. The Java project this developer was working in was applying nullability analyses on existing source code. The analyses often erroneously flagged values as nullable while in practice they would be not. This anecdote points out the difficult challenges of approximations in certain static analyses.

RQ5: Almost all developers have encountered and still regularly do encounter false positives/negatives in static analysis tools. Some static analyses can be wrong very frequently as they try to approximate the actual program behavior.

In conjunction with reducing warning numbers by ignoring them, van Graafeiland found that active monitoring of static analysis warnings on Continuous Integration (CI) also leads to a reduction in warnings. Therefore, we asked developers whether they actively run static analysis tools on CI and, if so, whether they let the build fail when a warning has been

introduced. Two-thirds of the developers (65.6%) indicated that they run static analysis tools on CI, of which about half (56.3% of 65.6%) also let the build fail.

RQ5: Usage of static analysis tools on Continuous Integration is common among developers. In half of the cases the build is configured to fail on static analysis tool warnings.

7.2.4 Feasibility and general perception

The very last section of our survey contained several statements regarding the feasibility and general perception of static analysis tools. Each statement was accompanied by a five-level Likert scale [42] ranging from "Strongly agree" to "Strongly disagree".

The first statement was focused on the static analysis plugin integration into the IDE: "A static analysis tool plugin distracts me from the task at hand during my development time". We wanted to know whether developers are discouraged to integrate these plugins into their IDE, as the continuous stream of warnings could overwhelm them or distract them from their work. However, developers overwhelmingy responded that they are either neutral (26.2%) or in disagreement: 39.3% disagreed and 26.2% strongly disagreed with this statement. Only 4.9% agreed and 3.3% strongly agreed with this statement.

RQ5: Developers point out they generally do not feel distracted after installing a static analysis tool plugin in their IDE.

Then we asked developer whether they think installation of a plugin can or does save overall development time. Previous research has estimated the cost of a defect for each phase in a project cycle [64]. The estimations by NASA indicate that the cost of a defect increases exponentially over time. Catching potential functional defects while working in the IDE, as generated from and shown by the static analysis tool plugin, therefore would be a cost-saving measure. The majority of developers also perceive plugins as cost-saving in terms of development time: 18.0% strongly agrees and 65.6% agrees. 13.1% is neutral and a sheer 3.3% disagrees, while no one strongly disagrees.

RQ5: Similarly to the findings from previous research that catching issues early is cost-saving, developers largely regard the installation of a static analysis tool plugin in their IDE as a net benefit to their overall development time.

While installation could be beneficial, we were also wondering whether enforcement of a (wide variety of) static analysis tool(s) in an open source project would defer developers from contributing. The question focuses on open source projects, because such projects normally do not have any specific team or company requirements for external developers.

Contributions to and working with an open source project is usually on a voluntary basis, with motivators like career advancement [39] or in general fun and enjoyment [59].

To investigate whether developers regard enforcement of static analysis tools as a barrier for contributing, we posed the statement "I do not want to contribute to an (open source) project which enforces zero static analysis tools warnings". Overall, developers generally disagree with this statement: 14.8% strongly disagrees, 39.3% disagrees and 29.5% is neutral. Only 13.1% agrees and 3.3% strongly agrees and would not consider to contribute to an open source project that enforces static analysis tools.

RQ5.2: In general, developers do not consider enforcement of static analysis tools as a barrier for contributing to an open source project.

Lastly, we wanted to know if there could be a relation between adoption of static analysis tools and writing tests for a project. On the statement "I will write fewer test cases if I can rely on static analysis tools being used in my project", there were very mixed results. While only 1 developer strongly agrees with this statement, the other options are largely uniformly distributed: 26.2% agrees, 23% is neutral, 32.8% disagrees and 16.4% strongly disagrees.

RQ5: There is no clear sentiment with respect to the adoption of static analysis tools and writing tests.

7.3 Summary

We conducted a survey to obtain a deeper insight in the perception of developers on their usage of static analysis tools. A majority of the developers indicated they use a linter and functional bug finder. Time estimations of how long it takes to resolve a warning is in-line with our quantative data: it takes less than an hour. Developers largely resolve warnings without consulting the internet. However, if they need to do so, they consult a search engine and Question-Answer websites. Removing warnings by adjusting the configuration of tools happens rarely in the IDE and more commonly by altering source code. False positives/negatives remain an important issue for developers, but usage of static analysis tools is still common on CI. Overall, developers regard the installation of static analysis tool plugins in their IDE as beneficial and performance-enhancing. It is therefore also not a barrier for contributing to an open source project.

Chapter 8

Discussion

This chapter includes a discussion with an overview of the research results. Then, we list several threats to validity on our findings.

8.1 Results overview

The previous two chapters contained detailed results based on our two datasets: a field research and a survey dataset. We would now like to give an overview of the results and answer each individual research question.

RQ1: Monitor capabilities of the IDE and plugins
The very first question we asked was related to the implementation of our choice for the mode of observation in the IDE. As such, the question was targeted towards the core engineering problems rather than a scientific analysis of results. To our knowledge, no previous work has implemented a static analysis observation framework. The closest related work is the work of WatchDog with an analysis framework for test usage and debugging analysis [7, 62]. Based on our experience, we can conclude that neither IDEs (**RQ1**) nor external static analysis tool plugins (**RQ1.1**) do not offer a fine-grained yet high-level API for static analysis observations. Chapter 4 explains our work of building a generic cross-editor implementation with the few APIs that were available.

A possible improvement for use-cases like ours is to introduce a dedicated listener framework for static analysis warning processes in the IDE. Such a listener framework should allow observers to obtain detailed information such as originating analysis rule, the dynamic data used in constructing the warning and fine-grained timing information for when the warning was introduced. All these improvements will prevent measures such as the classification process (Chapter 5) to improve the utility of the data.

RQ2: Frequency of different static analysis warning categories
After we successfully retrieved the classified warnings, we analyzed the relative frequency of warning categories. Table 6.3 shows the frequency of the top 25 categories, for which Figure 6.1 contains the graph with a trendline. The distribution of warning categories is an exponential distribution where few categories produce the majority of all observed warnings

(**RQ2**). Besides that, warnings in the top categories are in a majority of the time resolved as well (Figure 6.2). We were not able to confirm nor deny an influence of programming experience or overall warning frequency on the distribution of warning categories (**RQ2.1**).

From the top 25 warning categories, we could classify 17 categories into three high-level categories we introduced: *type resolution*, *unused declarations/tokens* and *import management*. Developers encounter *type resolution* warnings most frequently, which might be related to the problem of type errors preventing the compilation of a program. Other warnings are related to code hygiene where developers pay most attention to *unused declarations* and general issues with *imports*.

RQ3: Lifetime of a warning
Developers react to warnings quickly, as the median of the lifetime is 50 seconds (**RQ3** and Figure 6.5). Similarly to the warning frequency, *type resolution* warnings are resolved quickest as well (Figure 6.7). Next to that, we also observed a lot warnings taking considerably longer to resolve, up to a day. Respondents to our survey estimated that they solve most of the warnings in a couple of minutes. Developer perception therefore closely matches the observed reality. Again, we could not confirm nor deny a correlation between developer experience (**RQ3.1**) or warning frequency (**RQ3.2**) and the lifetime of a warning.

RQ4: Position of warnings in a file
A comparison of Java class declaration positions and the position of unresolved warnings (Figures 6.12 and 6.13) shows that there is a correlation between the two (**RQ4.1**). The warnings we observed were frequently near of just after a class declaration. However, the positions of warnings that are resolved is more focused on the last part of the file (**RQ4**). This indicates that there is a discrepancy (in terms of relative position in a file) between the warnings that are generated and the warnings that are resolved. The position of a warning is therefore a factor whether developers resolve a warning or not.

RQ5: Developer perception on static analysis tools
Developers indicate that they largely consider static analysis tools to be effective and time-saving, yet false positives are still encountered frequently (**RQ5**). They are also aware of automated refactoring techniques available in an IDE and a majority of the developer actively uses techniques to resolve warnings. Whenever developers encounter a warning they are unsure how to resolve, most developers indicate they consult a search engine (**RQ5.1** and Figure 7.1). Moreover, a search engine is most of the time a prerequisite for the consultation of other resources such as tool documentation and issue trackers. Figure 7.2 shows that developers tend to rarely change the configuration of static analysis tools and they rely on the default configuration or a slightly customized version. The default options in a static analysis tool are therefore influential for the behavior and perception of the tool, corroborating findings of Johnson et al. where developers indicated customizability is required when they disagree with the (default) functionality.

Continuous Integration is used in conjunction with static analysis tools, although not all developers actively monitor warnings by letting the build fail. Lastly, developers do not perceive the adoption of static analysis tools in open source projects as a blocking issue for them to publish contributions (**RQ5.2**).

8.2 Threats to validity

In this section, we discuss **limitations** of the results of both datasets as well as the three threats to validity as proposed by Perry et al.: **construct validity**, **internal validity** and **external validity** [52].

Limitations: The limitations of the field research dataset largely revolve around the mis-classification of static analysis warnings, as explained in Section 5.2.3. Since privacy of the developer's actions was a requirement for data collection, classification of the warnings became a necessity. The impact of missing classifications was limited as 13.25% of all warnings could not be classified.

Next to that, the field research results are based on the telemetry data obtained by WatchDog. Since WatchDog is a plugin for Eclipse and IntelliJ, behavior of developers using different IDEs/editors is not considered. From the list of static analysis tools provided by our survey respondents we can deduce that the majority of respondents are familiar with Java and frontend programming languages such as JavaScript and TypeScript. The analysis of survey results and telemetry data might therefore not be applicable for developers largely proficient with different languages.

Construct validity: our field research is based on the editor plugin WatchDog. In Section 3.4 we explained our reasoning for choosing WatchDog over other open source options. However, we also added our observation that WatchDog had not seen maintenance work for 1,5 years. To that end, while extensive infrastructure including a Ruby server and plugin client was well-tested from previous work ([7, 62]), we had to perform several architectural and maintenance-related upgrade to the implementation. There is a risk of the introduction of human errors during this upgrade process. To minimize this impact, we relied on the existing test suite, accompanied by a large number of manual end-to-end tests to confirm the upgrade works for both existing and new users.

All additional functionality used to be able to observe the output of static analysis tools was accompanied by automated integration tests to verify its correctness. The integration tests were written for the Eclipse part of the WatchDog plugin, but are not present in IntelliJ. Most of the added implementation exists in the core artifact, which is thus tested by the Eclipse test suite. While the IntelliJ-specific implementation is small, it is manually tested using the editor-supplied editor sandbox.

The functionality in WatchDog is therefore also dependent on the implementation quality of the external tools. There could be issues in their implementation that would end up with incorrect results. Moreover, as the IDE extensions can be maintained by different developers, differences in the core implementation and the plugin could become problematic. However, given the widespread usage of the tools and IDEs that are used in this thesis, we expect the impact of this to be minimal.

Lastly, the static analysis observations are only possible when the IDE is opened and WatchDog is active. This means that changes performed outside the IDE are not considered. It could be that a developer reacts to a warning shown in the IDE by executing an external tool. As such, while the IDE was involved and incentivized the developer to perform an action, WatchDog is unaware of this effect.

Internal validity: WatchDog can be installed by anyone, as we publicly share the plugin on our website `https://testroots.org/testroots_watchdog.html` as well in the public plugin repositories of Eclipse and IntelliJ. To that end, we do not control our participation subjects. In our field research results we had to remove the data from two of our subjects, as they both generated significantly more data than all other subjects combined. These two users generated a total of 60647 events, which is roughly equivalent to the number of events (61538) of all other users combined. A closer inspection of their data showed that they generated several large clusters of events within a second. We have been unable to discover any potential issues with the internal implementation of WatchDog. This leads us to believe they were using automated software, possibly code generation, which were correctly caught by WatchDog. Since we are interested in the developer usage of static analysis tools, rather than the effects of automated code generation, our analysis excludes these events. Otherwise, the statistics such as the category analysis would be skewed to the behavior of very small fraction of our complete subject population.

Our choice for mode of observation, explained in Section 3.1, has a potential impact on the behavior of our subjects, also known as Hawthorne Effect [1]. The choice for a silent observer in the form of an editor plugin was based on minimizing this risk. Developers were incentivized to install the plugin on the editor plugin repositories as well as on the subreddit /r/java[1]. To make sure developers understand the purpose of this silent observer, we had to inform them of our research intents. We would otherwise disincentivize developers of installing our plugins on the basis of privacy concerns and general lack of knowledge of its intent. When WatchDog is active in the editor, there are very few clues for the developer to realize the plugin is active. As such, while developers might initially be aware of the plugin and change their behavior, over time we expect their behavior to closely match with their normal real world programming behavior. Therefore, to minimize this threat we ran our field study over an extended period of time.

External validity: Similarly to previous studies based on WatchDog, participants largely stem from the Java community. Generalizing to the full programmer community is therefore limited, as other developer backgrounds can lead to different results. There are hints for this in our survey responses, for which several respondents included compilers as static analysis tools. Based on these options, we conclude that different programming languages communities rely on different compiler features and can thus have fewer tools installed. If a compiler for language A includes more static analysis features (for example the language includes type checking) than language B, developers for language A do not have to rely on a static analysis tool that detects potential type errors, while developers using language B do. The factor of programming language thus influences the choice of tools and warnings that can be shown to the developer.

[1] `https://www.reddit.com/r/java/comments/818zl9/watchdog_an_intellij_and_eclipse_plugin_that_we/`

Chapter 9

Conclusion and future work

In this last chapter we will conclude our research findings and give a summary of this thesis. Then, we propose a vision for future static analysis tooling and potential future work.

9.1 Conclusion

In this master thesis, we investigated several research questions related to the implementation of and observations on static analysis tools in the IDE. To be able to answer these questions, we designed a field research study and sent out a survey. The field research study is based on the IDE plugin WatchDog, which was previously also used to gather user statistics and track their behavior in the IDE.

We performed an extensive analysis of various API endpoints in the IDE and explained our reasoning for choosing to observe the textual representations of warnings shown to the developer. Based on the lack of fidelity of the chosen API, we had to anonymize the warnings to make sure we can observe real-world development activity without compromising potential privacy-sensitive work of the developer.

The analysis of our telemetry results shows that, in a majority of the time, developers encounter warnings from a small number of categories: the distribution of warning category frequency is exponential. The top 25 categories are primarily related to *type resolution*, *unused declarations/tokens* and *import management* and warnings introduced are resolved in a majority of the time. Moreover, warnings are usually resolved in a relative short time, with a median of 50 seconds, which developers estimated in our survey as well. Warnings related to *type resolution* are resolved the quickest. We could not confirm nor deny an effect of the programming experience on either the warning frequency, but it is not correlated to the lifetime. We also found a correlation between the position of a Java class declaration and the position of unresolved warnings in a file.

Our survey shows that developers make use of static analysis tools for enforcing style conventions and finding functional bugs. Developers are aware of automated refactoring techniques for resolving the static analysis warnings. When they are unsure how to solve a warning, a majority consults a search engine or a Question-Answer website. Tools are configured using the default configuration or a slightly customized version thereof and the

configuration is changed rarely. Overall, developers perceive static analysis tools as a cost-saving measure and generally beneficial in their development process.

The prevalence of perceived false positives/negatives combined with the exponential distribution of static analysis warning categories leads us to believe there are several opportunities to improve. In the next section, we will elaborate on a vision for the future of static analysis tooling combined with potential future work.

9.2 Future static analysis tooling

Based on the findings of our field and survey research, we would now like to elaborate on a vision of a potential future for static analysis tools. The vision largely includes changes to the tools in terms of reporting to and working with the developer.

9.2.1 Problem of absolute correctness

Our observations in the field research show a strong exponential distribution of the unique warning categories (Figure 6.1). A small set of warning categories produce the majority of warnings shown to the developer. Combining this knowledge with the perception of false positives/negatives (Section 7.2.3), we deduce that current static analysis tools have shortcomings with regard to their reporting capabilities.

At the moment, static analysis tools approximate the behavior of the program to the best of their abilities. The need for approximation is based on the inability to predict the eventual program behavior (without running the program) as static analysis is undecidable [40]. However, even seemingly simple analyses can produce a significant number of false positives. A study of the efficacy of the static analysis tool FindBugs on the JDK, the open source project Glassfish[1] and Google internal projects showed a significant number of false positives [3]. The authors note that improvements and refinements to FindBugs resolved most of the issues, but that there are still cases where the tool is incapable of understanding the developer's intent.

To be able to battle false positives, tools need to become smarter on when to issue warnings and when they should not. At the moment, tools report issues they can detect given a strict set of constraints. The implementation of the Google Java Style guide[2] in CheckStyle[3] mentions several limitations for which the tool is incapable of reporting violations to the style guide. As a result, given the implementation and limitation of the CheckStyle check, the check will produce incorrect results given the constraints defined by the CheckStyle authors. Even though the check is not perfect, the concept of approximation is not included in the reporting of the warning.

[1] `https://github.com/javaee/glassfish`
[2] `http://google.github.io/styleguide/javaguide.html`
[3] `http://checkstyle.sourceforge.net/google_style.html`

9.2.2 How to deal with incorrectness

Static analysis implementations are inherently limited to the design of the tools and the scenarios they take into account. If an analysis is executed on a scenario that it did not take into account or was not designed to able to deal with, the tool will generate a false positive. In this case, the problem of incorrectness is passed on to the developer, who has to investigate the supposed problem reported by the warning. The impending issue with this approach is the potential dismissal of all warnings of a tool, based on the frequent occurrence of false positives, if the developer regularly constructs scenarios the tool is not able to work with [31]. Since static analysis tools are enhancements to guard a developer from making (expensive) mistakes and not a strict requirement for the product behavior, developers are inclined to either ignore the results or disable the tool as a whole [57].

To combat the potential risk of incorrectness resulting in dismissal by the developer, Google developed Tricorder [56]. Tricorder is a platform enabling Google developers to integrate various analyses into their workflow and introduces a feedback loop between tool developers and their users. The Tricorder engineers employ a different definition of a false positive (coined as "effective false positive"): a false positive is defined by the user, not by the tool developer. Google engineers can dismiss warnings as "not useful", which can then result in the analysis being disabled until the implementation of the analysis has been improved. Moreover, analyses commonly are accompanied by automated fixes, which engineers can invoke by clicking in the code review interface.

Learn from the past

Next to the incorrectness measures integrated in Tricorder, we propose other data-driven integrations in the warning reporting interfaces. Based on our findings of lifetime of warnings (Figure 6.5), the frequency occurrences of different categories of warnings (Figure 6.3) and their respective position in the file (Figure 6.11), we propose that tools integrate previous data points of user behavior to determine whether a warning should be reported. By including these data points, tools can increase the usefulness for the developers and reducing the risk of incorrectness. This concept was also coined by Johnson et al. where developers pointed out they would like to have "*a form of temporary suppression*" [31]. Given the situation of the uselessness of *local variable is not used* when writing code shown in Figure 6.4, the aspect of lifetime can be included when deciding whether to report this warning. Instead of reporting at the moment of determining that the local variable is unused, the tool should (based on the lifetime of a warning) wait with reporting. If, for example, the warning is usually either fixed within a couple of seconds or considerably longer, waiting for 10 seconds greatly reduces the risk of uselessness.

By learning from the past behavior of the developer, tools can selectively show warnings that the developer cares about at that moment in time. Any warning that is resolved during the waiting timeframe is therefore one false positive that has been prevented. (*Note that the notion of the false positive is not in terms of correctness, as the warning was correctly determined to exist. However, it is similar to the interpretation of Sadowski et al. in which the developer determines what constitutes a false positive [56].*) The selection pro-

cedure could take into account not only just the average lifetime, but also its category (with corresponding lifetime and development context [68]), as well as position in the file.

One other note we have to make regarding the selection procedure is that tools should not hide warnings indefinitely. In other words: given a specific user profile, warnings can be selectively hidden. However, whenever a tool is invoked to report any potential warnings (for example on a CI server to let the build fail, Section 7.2.3), it should include all warnings. We envision the developer workflow to thus include a certain amount of development during which reporting is selectively hidden. During a code review or compilation step (as proposed by Sadowski et al. [56]), the full analysis is run which reports all errors. Moreover, developers can force a tool to report all warnings in their IDE when they are preparing for a code review, to fix any potential issue. Employing this workflow, the developer consciously works with a static analysis tool at the moments they feel most comfortable using the tool.

9.2.3 Machine learning

The concrete implementation of such an approach can be built on top of a machine learning solution. Machine learning can be used to predict the outcome of a process based on sufficient training data [55]. The training data in our case is the previous behavior of the developer. The parameters of the internal neural network consist of the various characteristics that can be computed; and which are currently included in WatchDog.

Evaluation of this system should focus on the usefulness for the developer. Similarly to the "not useful" provided in Tricorder, warnings shown in the IDE should include an option to dismiss a particular warning. Based on the characteristics of the dismissed warning, the neural network can be iteratively improved to prevent extensive dismissals by the developer. In essence, the base-case for this approach is the current absolute correctness in tools: they always report any warning. Over time, the network will selectively hide those warnings that a developer did not care about as well as take more behavioral data into account. The end result of the network would be the correct prediction of all warnings, such that the developer rarely if ever dismisses a single warning.

However, we have to take into account the effect of overfitting by hiding too many warnings, introducing "effective false negatives": the developer did actually care about the warning, even though it was never shown to them. One potential countermeasure would be the gradual increase in usefulness for every new occurrence of a warning that would be hidden. In other words: the longer a warning remains hidden, the higher the chance it is again shown to the developer.

9.2.4 Future work

With this vision, future work should be focused on both data gathering and data processing. The current observation techniques should be implemented into a variety of editors, to obtain a complete overview of development behavior independent of IDE choice. Moreover, by implementing more observation techniques, a more detailed analysis on the impact of language communities can be confirmed or denied. Besides an increased number of editors,

future work should also include a increase in participant population and overall duration of the observations.

For data processing, the chosen characteristics and processing method are crucial for obtaining good results. Future work should therefore investigate possible characteristics (with potentially weights) and implementations, as well as a general feasibility and usefulness of these factors for developers.

Bibliography

[1] John G Adair. The hawthorne effect: a reconsideration of the methodological artifact. *Journal of applied psychology*, 69(2):334, 1984.

[2] David Ascher, Paul F Dubois, Konrad Hinsen, Jim Hugunin, Travis Oliphant, et al. Numerical python, 2001.

[3] Nathaniel Ayewah, William Pugh, J. David Morgenthaler, John Penix, and YuQian Zhou. Evaluating static analysis defect warnings on production software. In *Proceedings of the 7th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, PASTE '07, pages 1–8, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-595-3. doi: 10.1145/1251535.1251536. URL http://doi.acm.org/10.1145/1251535.1251536.

[4] Nathaniel Ayewah, David Hovemeyer, J David Morgenthaler, John Penix, and William Pugh. Using static analysis to find bugs. *IEEE software*, 25(5), 2008.

[5] Earl Babbie. *The practice of social research*. Nelson Education, 2015.

[6] David F. Bacon and Peter F. Sweeney. Fast static analysis of c++ virtual function calls. In *Proceedings of the 11th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '96, pages 324–341, New York, NY, USA, 1996. ACM. ISBN 0-89791-788-X. doi: 10.1145/236337.236371. URL http://doi.acm.org/10.1145/236337.236371.

[7] Moritz Beller, Georgios Gousios, and Andy Zaidman. How (much) do developers test? In *Proceedings of the 37th International Conference on Software Engineering (ICSE)*, pages 559–562. IEEE, 2015.

[8] Moritz Beller, Radjino Bholanath, Shane McIntosh, and Andy Zaidman. Analyzing the state of static analysis: A large-scale evaluation in open source software. In *Software Analysis, Evolution, and Reengineering (SANER), 2016 IEEE 23rd International Conference on*, volume 1, pages 470–481. IEEE, 2016.

[9] Moritz Beller, Igor Levaja, Annibale Panichella, Georgios Gousios, and Andy Zaidman. How to catch 'em all: Watchdog, a family of ide plug-ins to assess testing. In *3rd International Workshop on Software Engineering Research and Industrial Practice (SER&IP 2016)*, pages 53–56. IEEE, 2016.

[10] Al Bessey, Ken Block, Ben Chelf, Andy Chou, Bryan Fulton, Seth Hallem, Charles Henri-Gros, Asya Kamsky, Scott McPeak, and Dawson Engler. A few billion lines of code later: using static analysis to find bugs in the real world. *Communications of the ACM*, 53(2):66–75, 2010.

[11] Igor Bobriakov. Top 15 python libraries for data science in 2017. `https://medium.com/activewizards-machine-learning-company/top-15-python-libraries-for-data-science-in-in-2017-ab61b4f9b4a7`, mar 2017. Accessed May 15 2018.

[12] Barry W Boehm, Maria H Penedo, E Don Stuckle, Robert D Williams, Arthur B Pyster, et al. A software development environment for improving productivity. In *Computer*. Citeseer, 1984.

[13] Louis Brandy. Curiously Recurring C++ Bugs at Facebook. `https://www.youtube.com/watch?v=3MB2iiCkGxg`, feb 2018.

[14] Michael L. Brodie and Michael Stonebraker. *Legacy Information Systems Migration: Gateways, Interfaces, and the Incremental Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1995. ISBN 1-55860-330-1.

[15] Pierre Carbonnelle. Pypl popularity of programming language. `http://pypl.github.io/IDE.html`, June 2018.

[16] Luca Cardelli. Type systems. *ACM Computing Surveys*, 28(1):263–264, 1996.

[17] HaiYang Chao, YongCan Cao, and YangQuan Chen. Autopilots for small unmanned aerial vehicles: A survey. *International Journal of Control, Automation and Systems*, 8(1):36–44, Feb 2010. ISSN 2005-4092. doi: 10.1007/s12555-010-0105-z. URL `https://doi.org/10.1007/s12555-010-0105-z`.

[18] Robert N Charette. Why software fails. *Ieee Spectrum*, 42(9):42–49, 2005.

[19] Colleen Cook, Fred Heath, and Russel L Thompson. A meta-analysis of response rates in web-or internet-based surveys. *Educational and psychological measurement*, 60(6):821–836, 2000.

[20] Ralph B d'Agostino. An omnibus test of normality for moderate and large size samples. *Biometrika*, 58(2):341–348, 1971.

[21] Jeffrey Dean, David Grove, and Craig Chambers. Optimization of object-oriented programs using static class hierarchy analysis. In Mario Tokoro and Remo Pareschi, editors, *ECOOP'95 — Object-Oriented Programming, 9th European Conference,*

Åarhus, Denmark, August 7–11, 1995, pages 77–101, Berlin, Heidelberg, 1995. Springer Berlin Heidelberg. ISBN 978-3-540-49538-3.

[22] Olive Jean Dunn. Multiple comparisons among means. *Journal of the American Statistical Association*, 56(293):52–64, 1961.

[23] Richard E Fairley. Tutorial: Static analysis and dynamic testing of computer software. *Computer*, 11(4):14–23, 1978.

[24] Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. *Refactoring: improving the design of existing code*. Addison-Wesley Professional, 1999.

[25] Erich Gamma. *Design patterns: elements of reusable object-oriented software*. Pearson Education India, 1995.

[26] Samuel Gibbs. Warning signs for TSB's IT meltdown were clear a year ago - insider. https://www.theguardian.com/business/2018/apr/28/warning-signs-for-tsbs-it-meltdown-were-clear-a-year-ago-insider, apr 2018. Accessed May 15 2018.

[27] Georgios Gousios, Andy Zaidman, Margaret-Anne Storey, and Arie Van Deursen. Work practices and challenges in pull-based development: the integrator's perspective. In *Proceedings of the 37th International Conference on Software Engineering-Volume 1*, pages 358–368. IEEE Press, 2015.

[28] Brian Heater. Twitter is down again for some. https://techcrunch.com/2018/04/20/twitter-is-down-again-for-some/, apr 2018. Accessed May 15 2018.

[29] John D Hunter. Matplotlib: A 2d graphics environment. *Computing in science & engineering*, 9(3):90–95, 2007.

[30] Inc. JetBrains. Static code analysis. https://www.jetbrains.com/idea/docs/StaticCodeAnalysis.pdf, June 2018.

[31] Brittany Johnson, Yoonki Song, Emerson Murphy-Hill, and Robert Bowdidge. Why don't software developers use static analysis tools to find bugs? In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 672–681, Piscataway, NJ, USA, 2013. IEEE Press. ISBN 978-1-4673-3076-3. URL http://dl.acm.org/citation.cfm?id=2486788.2486877.

[32] Andrey Karpov. Chromium: the Sixth Project Check and 250 Bugs. https://www.viva64.com/en/b/0552/, jan 2018.

[33] Lennart C.L. Kats and Eelco Visser. The spoofax language workbench: Rules for declarative specification of languages and ides. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '10, pages 444–463, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0203-6. doi: 10.1145/1869459.1869497. URL http://doi.acm.org/10.1145/1869459.1869497.

[34] Hope King. Woman's home demolished after Google Maps error. http://money.cnn.com/2016/03/25/technology/google-maps-house/index.html, mar 2016. Accessed May 15 2018.

[35] Thomas Kluyver, Benjamin Ragan-Kelley, Fernando Pérez, Brian E Granger, Matthias Bussonnier, Jonathan Frederic, Kyle Kelley, Jessica B Hamrick, Jason Grout, Sylvain Corlay, et al. Jupyter notebooks-a publishing format for reproducible computational workflows. In *Proceedings of the 20th International Conference on Electronic Publishing*, 2016.

[36] John C Knight. Safety critical systems: challenges and directions. In *Software Engineering, 2002. ICSE 2002. Proceedings of the 24rd International Conference on*, pages 547–550. IEEE, 2002.

[37] Andrew J. Ko, Bryan Dosono, and Neeraja Duriseti. Thirty years of software problems in the news. In *Proceedings of the 7th International Workshop on Cooperative and Human Aspects of Software Engineering*, CHASE 2014, pages 32–39, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2860-9. doi: 10.1145/2593702.2593719. URL http://doi.acm.org/10.1145/2593702.2593719.

[38] William H Kruskal and W Allen Wallis. Use of ranks in one-criterion variance analysis. *Journal of the American statistical Association*, 47(260):583–621, 1952.

[39] Karim R Lakhani and Robert G Wolf. Why hackers do what they do: Understanding motivation and effort in free/open source software projects. *Perspectives on Free and Open Source Software*, 2003.

[40] William Landi. Undecidability of static analysis. *ACM Lett. Program. Lang. Syst.*, 1 (4):323–337, December 1992. ISSN 1057-4514. doi: 10.1145/161494.161501. URL http://doi.acm.org/10.1145/161494.161501.

[41] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, CGO '04, pages 75–, Washington, DC, USA, 2004. IEEE Computer Society. ISBN 0-7695-2102-9. URL http://dl.acm.org/citation.cfm?id=977395.977673.

[42] Rensis Likert. A technique for the measurement of attitudes. *Archives of psychology*, 1932.

[43] Benjamin Livshits and Monica S. Lam. Finding security vulnerabilities in java applications with static analysis. In *USENIX Security Symposium*, 2005.

[44] Mark Masse. *REST API Design Rulebook: Designing Consistent RESTful Web Service Interfaces*. " O'Reilly Media, Inc.", 2011.

[45] Tom Mens and Tom Tourwé. A survey of software refactoring. *IEEE Transactions on software engineering*, 30(2):126–139, 2004.

[46] Clark Mindock. Facebook down - latest updates: Social network stops working for millions worldwide. https://www.independent.co.uk/life-style/gadgets-and-tech/news/facebook-down-latest-updates-not-work-social-media-user-accounts-posts-update-a8178611.html, jan 2018. Accessed May 15 2018.

[47] Gail C Murphy, Mik Kersten, and Leah Findlater. How are java software developers using the elipse ide? *IEEE software*, 23(4):76–83, 2006.

[48] Eugene W Myers. Ano (nd) difference algorithm and its variations. *Algorithmica*, 1 (1-4):251–266, 1986.

[49] William F Opdyke. *Refactoring object-oriented frameworks*. PhD thesis, University of Illinois at Urbana-Champaign, 1992.

[50] Sebastiano Panichella, Venera Arnaoudova, Massimiliano Di Penta, and Giuliano Antoniol. Would static analysis tools help developers with code reviews? In *22nd IEEE International Conference on Software Analysis, Evolution, and Reengineering, SANER 2015, Montreal, QC, Canada, March 2-6, 2015*, pages 161–170, 2015. doi: 10.1109/SANER.2015.7081826. URL https://doi.org/10.1109/SANER.2015.7081826.

[51] Daniël Pelsmaeker. Portable editor services. Master's thesis, Delft University of Technology, 2018. URL http://resolver.tudelft.nl/uuid:c8b554de-bcb6-4896-a9bf-c03cca37e344.

[52] Dewayne E Perry, Adam A Porter, and Lawrence G Votta. Empirical studies of software engineering: a roadmap. In *Proceedings of the conference on The future of Software engineering*, pages 345–355. ACM, 2000.

[53] Roger S Pressman. *Software engineering: a practitioner's approach*. Palgrave Macmillan, 2005.

[54] Feng Qin, Joseph Tucek, Jagadeesan Sundaresan, and Yuanyuan Zhou. Rx: Treating bugs as allergies—a safe method to survive software failures. In *Proceedings of the Twentieth ACM Symposium on Operating Systems Principles*, SOSP '05, pages 235–248, New York, NY, USA, 2005. ACM. ISBN 1-59593-079-5. doi: 10.1145/1095810.1095833. URL http://doi.acm.org/10.1145/1095810.1095833.

[55] J Ross Quinlan. *C4. 5: programs for machine learning*. Elsevier, 2014.

[56] Caitlin Sadowski, Jeffrey van Gogh, Ciera Jaspan, Emma Soederberg, and Collin Winter. Tricorder: Building a program analysis ecosystem. In *International Conference on Software Engineering (ICSE)*, 2015.

[57] Caitlin Sadowski, Edward Aftandilian, Alex Eagle, Liam Miller-Cushon, and Ciera Jaspan. Lessons from building static analysis tools at google. *Commun. ACM*, 61(4): 58–66, March 2018. ISSN 0001-0782. doi: 10.1145/3188720. URL http://doi.acm.org/10.1145/3188720.

[58] Morris S. Schwartz and Charlotte Green Schwartz. Problems in participant observation. *American Journal of Sociology*, 60(4):343–353, 1955. doi: 10.1086/221566. URL https://doi.org/10.1086/221566.

[59] Sonali K Shah. Motivation, governance, and the viability of hybrid forms in open source software development. *Management science*, 52(7):1000–1014, 2006.

[60] Gustavo Soares, Melina Mongiovi, and Rohit Gheyi. Identifying overly strong conditions in refactoring implementations. In *Software Maintenance (ICSM), 2011 27th IEEE International Conference on*, pages 173–182. IEEE, 2011.

[61] Donna Spencer. *Card sorting: Designing usable categories*. Rosenfeld Media, 2009.

[62] N Spruit. What programmers know about debugging and how they use their ide debuggers. Master's thesis, Delft University of Technology, 2016. URL http://resolver.tudelft.nl/uuid:bf3325ce-f246-4977-91bc-785f877347b8.

[63] JetBrains s.r.o. Code inspection. https://www.jetbrains.com/help/idea/code-inspection.html, jan 2018.

[64] Jonette M Stecklein, Jim Dabney, Brandon Dick, Bill Haskins, Randy Lovell, and Gregory Moroney. Error cost escalation through the project life cycle. *Proceedings of the 14th Annual International Symposium organized by the International Council on Systems Engineering (INCOSE) Foundation*, 2004.

[65] Jack Stilgoe. Machine learning, social learning and the governance of self-driving cars. *Social Studies of Science*, 48(1):25–56, 2018. doi: 10.1177/0306312717741687. URL https://doi.org/10.1177/0306312717741687. PMID: 29160165.

[66] Russell H Taylor, Arianna Menciassi, Gabor Fichtinger, Paolo Fiorini, and Paolo Dario. Medical robotics and computer-integrated surgery. In *Springer handbook of robotics*, pages 1657–1684. Springer, 2016.

[67] B. van Graafeiland. Static code analysis tools: Effects on development of open source software. Master's thesis, Delft University of Technology, 2016. URL http://resolver.tudelft.nl/uuid:b157de07-e5ce-4dba-8eae-154a0002a4f5.

[68] Carmine Vassallo, Sebastiano Panichella, Fabio Palomba, Sebastian Proksch, Andy Zaidman, and Harald C Gall. Context is king: The developer perspective on the usage of static analysis tools. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 38–49. IEEE, 2018.

[69] J Wiegand et al. Eclipse: A platform for integrating development tools. *IBM Systems Journal*, 43(2):371–383, 2004.

[70] Fiorella Zampetti, Simone Scalabrino, Rocco Oliveto, Gerardo Canfora, and Massimiliano Di Penta. How open source projects use static code analysis tools in continuous integration pipelines. In *Mining Software Repositories (MSR), 2017 IEEE/ACM 14th International Conference on*, pages 334–344. IEEE, 2017.

[71] Misha Zitser, Richard Lippmann, and Tim Leek. Testing static analysis tools using exploitable buffer overflows from open source code. In *Proceedings of the 12th ACM SIGSOFT Twelfth International Symposium on Foundations of Software Engineering*, SIGSOFT '04/FSE-12, pages 97–106, New York, NY, USA, 2004. ACM. ISBN 1-58113-855-5. doi: 10.1145/1029894.1029911. URL http://doi.acm.org/10.1145/1029894.1029911.

Appendix A

Glossary

In this appendix we give an overview of frequently used terms and abbreviations.

BSON: "binary-encoded serialization of JSON-like documents" - `http://bsonspec.org`

ClassLoader: A class that is used to dynamically load code in a Java application `https://docs.oracle.com/javase/10/docs/api/java/lang/ClassLoader.html`

Continuous Integration (CI): Practice of regularly building an application automatically to catch (integration) problems faster, usually by installing a clean setup on a external server.

Eclipse: IDE developed by the Eclipse Foundation `https://www.eclipse.org/downloads/eclipse-packages/`

IDE: Interactive Developer Environment which developers use to build applications with assistance of tools to increase productivity

IntelliJ (Idea): IDE developed by JetBrains `https://www.jetbrains.com/idea/`

REST API: Architecture style to design web APIs accessible over HTTP [44]