Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

# Automatically Fixing Static Analysis Tools Violations

Diego Venâncio Marcilio

Dissertação apresentada como requisito parcial para
conclusão do Mestrado em Informática

Orientador
Prof. Dr. Rodrigo Bonifácio de Almeida

Brasília
2019

Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

# Automatically Fixing Static Analysis Tools Violations

Diego Venâncio Marcilio

Dissertação apresentada como requisito parcial para
conclusão do Mestrado em Informática

Brasília, 05 de agosto de 2019

# Dedication

Along my career, both in industry and academia, I had the great luck of being surrounded by great colleagues (including bosses and supervisors). I dedicate this work to them, as I am sure it would not have been possible without their help and support.

# Acknowledgements

I thank my supervisor, Dr. Rodrigo Bonifácio, for all the patience and support during all these last years. I also thank all my coauthors for their invaluable support and input on the work done in this dissertation. My friends and family must not also be forgotten. Thank you!

# Resumo

A qualidade de *software* tem se tornado cada vez mais importante à medida que a sociedade depende mais de sistemas de *software*. Defeitos de *software* podem custar caro à organizações, especialmente quando causam falhas. Ferramentas de análise estática analisam código para encontrar desvios, ou violações, de práticas recomendadas de programação definidas como regras. Essa análise pode encontrar defeitos de *software* de forma antecipada, mais rápida e barata, em contraste à inspeções manuais. Para corrigir-se uma violação é necessário que o programador modifique o código problemático. Essas modificações podem ser tediosas, passíveis de erro e repetitivas. Dessa forma, a automação de transformações de código é uma funcionalidade frequentemente requisitada por desenvolvedores. Esse trabalho implementa transformações automáticas para resolver violações identificadas por ferramentas de análise estática. Primeiro, nós investigamos o uso da ferramenta SonarQube, uma ferramenta amplamente utilizada, em duas grandes organizações *open-source* e duas instituições do Governo Federal do Brasil. Nossos resultados mostram que um pequeno subconjunto de regras é responsável por uma grande porção das violações resolvidas. Nós implementamos transformações automáticas para 11 regras do conjunto de regras comumente resolvidas achadas no estudo anterior. Nós submetemos 38 *pull requests*, incluindo 920 soluções para violações, geradas automaticamente pela nossa técnica para diversos projetos open-source na linguagem Java. Os mantenedores dos projetos aceitaram 84% das nossas transformações, sendo 95% delas sem nenhuma modificação. Esses resultados indicam que nossa abordagem é prática, e pode auxiliar desenvolvedores com resoluções automáticas, uma funcionalidade frequentemente requisitada.

**Palavras-chave:** Transformações de Programas, Ferramentas de Análise Estática

# Abstract

Software quality is becoming more important as the reliance on software systems increases. Software defects may have a high cost to organizations as some can lead to software failure. Static analysis tools analyze code to find deviations, or violations, from recommended programming practices defined as rules. This analysis can find software defects earlier, faster, and cheaper than manual inspections. When fixing a violation, a programmer is required to modify the violating code. Such modifications can be tedious, error-prone, and repetitive. Unsurprisingly, automated transformations are frequently requested by developers. This work implements automatic transformations tailored to solve violations identified by static analysis tools. First, we investigate the use of SonarQube, a widely used Static Analysis Tool, in two large open source organizations and two Brazilian Government Federal Institutions. Our results show that a small subset of the rules is responsible for a large portion of the fixes. We implement automatic fixes for 11 rules from the previously found set of frequently fixed rules. We submitted 38 pull requests, including 920 fixes generated automatically by our technique for various open-source Java projects. Projects maintainers accepted 84% of our fixes (95% of them without any modifications). These results indicate that our approach is feasible, and can aid developers with automatic fixes, a long requested feature.

**Keywords:** Program Transformation, Static Analysis Tools

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

Software quality is becoming more important as society is increasingly more dependent on software systems. Software defects, or bugs, may incur in a high cost to companies, especially when they cause failures [1]. To circumvent these potential problems, there are are some strategies to assess and assure quality in software, including tests and source code reviews [1].

As an example, Automatic Static Analysis Tools (ASATs) provide means of analyzing source code without the necessity of running it, aiding the quality assurance task during the software development process. ASATs can find potential problems in the source code [2], also named source-code violations, that indicate deviations from recommended coding standards or good practices [3], which are verified by rules defined within an ASAT. ASATs are widely used for both private and open-source software development [4, 5]. Some organizations even employ development workflows stating that a project needs to comply to ASATs checks before a release [3, 6].

To correct these violations, developers need to take action—commonly applying a source code transformation. Such transformations can be tedious, error-prone, require changes on multiple lines and files, and, in some cases, also require non-trivial inferences and considerations [7, 8]. These hindrances reinforce the need for automatic program transformations.

The overarching motivation for this dissertation is to aid developers in reducing ASATs' violations by automatically transforming offending Java source code. To fulfill this goal, we first explore what kind of violations developers tend to fix (see Chapter 2). With this information, we select commonly fixed violations to implement the SpongeBugs tool, which currently support program transformations that fix 11 Java rules. As shown in Chapter 3, SpongeBugs is effective to solve a high percentage (81%) of common violations while scaling, performance wise, satisfactorily to large code bases. Moreover, SpongeBugs' generated patches have a high acceptance rate, which might indicate its applicability in helping developers reduce the technical debt incurred by ASATs' violations.

This chapter introduces the concepts, research questions, research methods, contributions, and the overall structure of this dissertation.

## 1.1 Software Quality

Software quality, per ISO/IEC 24765, can be defined as the capacity of a software product in satisfying explicit and implicit necessities when used under specific conditions [9].

When trying to measure quality in software, a vast variety of dimensions and factors have been proposed. Each one of them aims at defining a set of characteristics that, when fulfilled, increase the chances of producing a high-quality software [10]. One classification for these characteristics is to divide them into two categories: 1) external attributes; and 2) internal attributes [11].

External attributes are visible to the users of a system, while internal attributes are of the developers' interest [11]. Measuring external attributes, such as usability and portability, poses several challenges and difficulties as the assessment usually involves the knowledge and experience from the users of the software [12]. Internal quality attributes are much easier to be quantified in comparison to external attributes [13]. They are independent of the environment in which they execute and can be measured by only observing its source code [13]. In general, software users are only concerned with external quality attributes, but the internal attributes are the reason that those external attributes were reached in the first place [11]. In this way, it is recommended to determine a system's external attributes by observing its internal quality attributes [14], since there are well defined metrics to measure internal quality attributes [12]. Examples of internal attributes are size, cohesion, and coupling [13], and examples of metrics that measure these attributes are, respectively [12]: LOC (Lines of Code), LCOM (Lack of Cohesion in Methods), and COF (Coupling Factor).

The ISO/IEC 25010 [15] states that internal quality software attributes represent static attributes of a software product that are related to its architecture, structure, and components. These attributes can be verified by employing revision, inspection, simulation, and/or automatic tools. One of the ways used by automatic tools to inspect source code is Static Analysis, which calculate several software metrics, among other aspects. We discuss Static Analysis in Section 1.2.

## 1.2 Static Analysis

Static analysis is a verification technique that examines source code without running the program. It aims at capturing defects earlier in the development process, aiding on improving software quality [1].

A relevant motivation for the technique is that certain kinds of defects are rarely observed, be it from rarely occurring, or from not representing problems perceived as severe enough [16]. ASATs can find an important class of problems that are typically not found neither by unit tests nor by manual inspections [17].

While software tests may detect a potentially large fraction of defects, they may be disadvantageous due to its high cost [16]. Tests are also written by humans, which might result in incorrect and incomplete judgments when assessing source code functionality [17]. ASATs should complement the test activity in a software quality assurance process [17].

ASATs can be integrated to the development pipeline by a variety of ways, such as on demand, just in time before the source code is stored in a source management system, or continuously during software development activities [1]. The latter can be achieved through the adoption of continuous integration (CI) practices, specifically by continuous inspection, which includes static analysis of source code [18].

## 1.3 Automatic Static Analysis Tools

Several tools integrate static analysis into development workflows, including SonarQube. SonarQube [19] is one of the most adopted code analysis tool in the context of CI environments [18, 20]. It supports more than 25 languages and is used by more than 85,000 organizations. SonarQube includes its own rules and configurations, but new rules can be added. Notably, it incorporates popular rules of other static and dynamic code analysis tools, such as FindBugs and PMD [18].

SonarQube considers rules as coding standards. When a piece of code violates a rule, an issue is raised. SonarQube classifies issues by type and severity. Issues' types are related to the code itself [21]. There are three broad kinds of issues on SonarQube. A **Bug** occurs when an issue is related to a piece of code that is demonstrably wrong. A **Vulnerability** occurs when a piece of code could be exploited to cause harm to the system. Finally, a **Code smell** occurs when an issue represent instances of improper code, which are neither a bug nor a vulnerability.

The severities of issues can also be categorized by their possible impact, either on the system or on the developer's productivity. **Blocker** and **critical** issues might impact negatively the system, with blocker issues having a higher probability compared to critical ones. SonarQube recommended to fix these kind of issues as soon as possible[1]. **Major** issues can highly impact the productivity of a developer, while **minor** ones have little impact. Finally, **info** issues represent all issues that are neither a bug nor a quality flaw. In SonarQube, issues flow through a lifecycle, taking one of multiple possible statuses, namely: **open**, which is set by SonarQube on new issues; **resolved**, set manually to indicate that an issue should be closed; **closed**, which is set automatically by SonarQube when an issue is fixed.

## 1.4 Program Transformation

Program transformation can be applied in diverse software engineering areas such as compilation, optimization, refactorings, and software rejuvenation [22]. The goal of transforming programs is to increase the developers' productivity by automating programming tasks, thus, allowing the developer to focus more in high-level details, which might increase external software attributes such as maintainability and reusability [22].

Even though programs are written as text, a textual representation is often not appropriate to apply complex transformations [22]. In this manner, a structured representation, along with semantic rules, of the program is needed. The structure allows programs to be transformed while the semantics provide means of comparing programs and of verifying the feasibility of applying transformations [22].

Transforming a program involves changing a program to another. It can also describe, more formally, the algorithm that implements the program transformation. The language in which the program being transformed and the resulting program are written is called the source and target languages, respectively [22]. When a program is transformed and both source and target are in the same programming language, this process is called as rephrasing [22]. Rephrasing usually aims at improving one internal quality attribute of the program [22].

A parse tree is a graphical representation that corresponds to the input program [23]. Parse trees contain syntactic information such as layout (blank spaces and commentaries) [22], and thus they are relatively large in comparison to the source code [23]. This information is usually not relevant to a large portion of transformations; thus, parse trees are frequently transformed into abstract syntax trees (ASTs) [22]. An AST retains the essential structure of the parse tree but eliminates unnecessary syntactic information [23]. ASTs are widely used in compilers and interpreters. Source-to-source systems often use ASTs to regenerate code easily [23].

[1]https://docs.sonarqube.org/latest/user-guide/issues/

In this thesis we developed SpongeBugs, a program transformation tools that aims to fix issues of ASATs. SpongeBugs is implemented in Rascal [24], a domain-specific language for source code analysis and manipulation, which facilitates several common meta-programming tasks, such as traversing the program structure, pattern matching, defining templates for program transformation, and extracting an AST from source code.

## 1.5 Research Questions

This research explores the topic of "*Automatic static analysis tools' violations*". We address two main research questions here:

(RQ1) What kind of violations developers tend to fix?

(RQ2) Is it possible to reliably aid developers with automatic fixes for static analysis tools' violations

We structure our research goals using a GQM (Goals/Questions/Metrics) template [25] in Table 1.1 and Table 1.2. Then we introduce in Sections 1.5.1 and 1.5.2 the empirical evaluation done to answer the research questions.

| Analyze | Object under measurement |
|---|---|
| *For the purpose of* | understand what kind of Java ASATs' violations developers tend to fix. |
| *With respect to* | the practitioners' practices and perceptions on fixing ASAT issues. |
| *From the view point of* | practitioners that work or contribute to organizations that employ software quality assessment using ASATs. |
| *In the context of* | private and open organizations that use ASATs in their Java projects. |

Table 1.1: GQM related to the goal of investigating what kind of violations developers tend to fix

| Analyze | Object under measurement |
|---|---|
| *For the purpose of* | evaluate whether automatically fixing ASATs' violations is applicable in supporting developers . |
| *With respect to* | the applicability of the proposed tool (SpongeBugs) on:<br>(a) how many rules it can detect and fix;<br>(b) how many of the fixes are accepted by open-source maintainers;<br>(c) how scalable it is in terms of running time on large code bases. |
| *From the view point of* | open-source projects maintainers. |
| *In the context of* | well established open-source projects that utilize SonarQube. |

Table 1.2: GQM related to the goal of studying whether automatically fixing issues is applicable in supporting developers to improve quality attributes of a system

### 1.5.1 Perceptions and practices on fixing ASATs' violations

Even though the use of ASATs provide several benefits, developers face several challenges when using and adopting ASATs [1, 26]. Practitioners may find a high number of warnings, and often in the thousands, when analyzing code with these tools, especially on the first time a set of rules is run [1]. This high number of warnings hinders the developer task of filtering through issues, which can result in violations being ignored [26]. To better select a representative set of rules that developers may find valuable, we investigate what kind of issues are frequently fixed in hundred of systems from two large open-source foundations (Apache Software Foundation and Eclipse Foundation) and two Brazilian Federal Government institutions (Brazilian Court of Account (TCU) and the Brazilian Federal Police (PF)). We follow the rationale introduced by Liu et al. [27] that frequently fixed rules are more likely to correspond to issues that developers are interested.

#### Background

Although previous works have already investigated how open-source software (OSS) projects use ASATs (e.g., [6, 21, 27]), we find gaps in their findings. Digkas et al. [21] and Liu et al. [27] study which kind of violations developers tend to fix on SonarQube and FindBugs, respectively. SonarQube and FindBugs are Static Analysis Tools widely used in practice and studied in the literature. Both studies rely on revisions of the projects and then run the ASATs. We argue that this might not reflect the precise usage of these tools. Also, their study restricted some type of rules and did not have any industrial projects in their datasets.

#### (RQ1) What kind of violations developers tend to fix?

This research question aims to build a broad comprehension of how developers use SonarQube and how they respond to the warnings reported by it. We observed a realistic usage of SonarQube, which do not rely on software revisions. As stated by Liu et al., when researchers investigate fixes by running the tool on revisions, they face the threat that developers might not use the ASAT in their toolchain. This possible lack of use might lead to a piece of code reported as fixed that was not ever perceived as a violation, thus, not fixed intentionally. Consequently, previous studies did not precisely report on how developers respond to violations.

#### Research Method

To answer this research question, we first conducted an online survey with 18 developers of the analyzed projects from the four studied organizations. We asked 6 closed questions mainly using a Likert scale [28]. The estimated time to complete the survey was between 12 to 15 minutes. The survey was available for approximately one month, and in the end, we had a completion rate of 23% (18 developers from 81 unique visits).

For our quantitative analysis, we identified 4 SonarQube instances, one for each organization studied. Our selection focused on projects from Eclipse Foundation (EF) and Apache Software Foundation (ASF), two of the largest open-source foundations [29]. For our private datasets, we selected two Brazilian government institutions not only due to convenience, also because they have heterogeneous contexts: TCU does in-house development whereas PF mostly outsources. We then implemented a tool to mine data from these instances. Overall, we mined 421,976 issues from 246 projects. We leveraged statistical techniques, including exploratory data analysis (plots and descriptive statistics) and hypothesis testing methods to analyze our mined data and then investigate the practices for fixing issues.

**Summary of the Results**

The survey results indicate that more than 80% of the developers consider ASATs warnings relevant for overall software improvement. In some cases (22%) developers postpone at least once a release or reject a pull-request based on the output of ASATs. Moreover, more than 60% of the developers consider program transformation tools to fix issues automatically. However, 66.6% of the respondents never or rarely use a tool for this purpose. This reinforces the need for tools that focus on fixing ASATs violations.

Our quantitative analysis found a low resolution of issues (only 8.77% of the issues were fixed). ASATs' violations are fixed in median faster than bugs, and some issues are fixed faster than others. We found that violations classified as Code Smells (improper code) are responsible for almost 70% of all of the fixed issues. Interestingly, some kind of issues is present among the most fixed and also the most non-fixed sets. This indicates that developers consider a variety of factors when deciding whether to fix an issue. Additionally, our results show that 20% of the rules correspond to 80% of the fixes, which paves the way for selecting a subset of the rules that developers might find valuable.

### 1.5.2 Automatically providing fix suggestions for ASATs' violations

Developers frequently request tools that can fix ASATs' violations. However, implementing fixes poses several challenges. First, ASATs are infamously known for their high percentage of false-positives (issues that do not represent an actual problem). Fixing violations that developers do not perceive as a real problem might be a waste of time (in particular, to time to review the modifications, which can span multiple lines of code and files). Secondly, a common limitation in the program repair literature is to generate fixes, or patches, that developers find acceptable. Fixing a violation is often not enough. Developers must assess that the patch has enough quality. Lastly, the approach must not take a long time to run. A tool should integrate within the developer workflow, not be on his way.

### Background

Automatic Static analysis tools (ASATs), such as SonarQube and FindBugs/SpotBugs, analyze source code by checking it against several independent rules, which developers can select. Each rule describes a good practice or recommended behavior that high-quality code should follow. Whenever a piece of code violates one of the rules, the static analysis tool emits a warning, which typically indicates the violating piece of code and the rule which the violations refers to. A developer must assess whether the violation indeed represents a deviation from good practice, and in the case it is, must also come up with a fix that modifies the source-code to remove the warning. Previous work [5], which presents the tool AVATAR, has already explored automatic fixes for ASATs. However, AVATAR focuses exclusively on behavioral *semantic* bugs, which infer *behavioral* properties that can be used to perform program optimizations as well as performance problems.

### (RQ2) Is it possible to reliably aid developers with automatic fixes for static analysis tools' violations?

This research question aims to evaluate whether our approach (SpongeBugs) to fix *syntactic* design flaws (identified by ASATs) is practical on three different perspectives: a) the ratio of the identified violations, for the rules it implements, that SpongeBugs can fix; b) whether developers find the fixes generated by SpongeBugs acceptable; c) whether SpongeBugs scales reasonably in large code bases. By answering these questions, we can assess whether SpongeBugs can reliably aid developers.

### Research Method

We selected and implemented fixes for 11 rules from previous studies that explored ASATs' issues. We select 15 open-source Java projects that use SonarQube to evaluate our approach. We ran SonarQube before and after applying our fixes, calculating the number of open and fixed issues. To evaluate the acceptance for our patches, we submit 38 pull-requests containing more than 920 fixes for our selected projects. After contacting developers to find out if our proposed fixes are welcome, we end with 12 projects. These projects include the Eclipse IDE and ASATs such as SonarQube and SpotBugs (successor to FindBugs). Finally, we analyze how long SpongeBugs takes to run in projects with varying sizes.

### Summary of the Results

SpongeBugs is able to fix 81% of all the violations found by SonarQube for the rules it implements. Regarding the developers' acceptance on the generated fixes, 34 of the 38 pull-requests were accepted. Developers accepted 84% of the fixes, with 95% of them accepted without modifications. SpongeBugs approach scales on projects of realistic size, it can run in under 10 minutes on project as large as 550 thousands lines of source-code.

## 1.6 Contributions

This sections lists and summarizes the peer-reviewed contributions, and explains how they are mapped to the chapters in this dissertation. I was the primary author of these two publications.

### A Closer Look at Realistic Usage of ASAT tools

Chapter 2 answers our RQ1 by mining SonarQube data from two large open-source foundations and two Brazilian Federal Government institutions. We find that a small set of rules are responsible for a large portion of the fixes.

> Diego Marcilio, Rodrigo Bonifácio, Eduardo Monteiro, Edna Canedo, Welder Luz, and Gustavo Pinto. 2019. "Are static analysis violations really fixed?: a closer look at realistic usage of SonarQube". In Proceedings of the 27th International Conference on Program Comprehension (ICPC '19). IEEE Press, Piscataway, NJ, USA, 209-219. DOI: https://doi.org/10.1109/ICPC.2019.00040

### SpongeBugs: Automatically Generating Fix for ASATs warnings

Chapter 3 answers RQ2 by proposing a tool (SpongeBugs) that automatically fixes violations from ASATs. We found that our tool is practical as more than 84% of the fixes submitted to open-source projects were accepted. Moreover, 95% of them were accepted without modification. The results of this investigation was accepted for publication at SCAM 2019 (19th IEEE International Working Conference on Source Code Analysis and Manipulation).

> Diego Marcilio, Carlo A. Furia, Rodrigo Bonifacio, Gustavo Pinto. "Automatically Generating Fix Suggestions in Response to Static Code Analysis Warnings". 2019

IEEE 19th International Working Conference on Source Code Analysis and Manipulation (SCAM), Cleveland, 2019

Moreover, when answering RQ2 we identified that Rascal's Java 8 grammar was lacking Binary Literals[2], which allows numbers to be expressed in the binary number system. We contributed with Rascal's grammar by adding support to Binary Literals in this pull-request: https://github.com/usethesource/rascal/pull/1245

### Datasets and Tools

The study detailed in Chapter 2 produced a large dataset with 421,976 issues from 246 projects. This dataset is publicly available as:

Diego Marcilio, Rodrigo Bonifácio, Eduardo Monteiro, Edna Canedo, Welder Luz, & Gustavo Pinto. (2019). Are Static Analysis Violations Really Fixed? A Closer Look at Realistic Usage of SonarQube. Dataset for OSS organizations (Version 1) [Data set]. Zenodo. http://doi.org/10.5281/zenodo.2602039

We also make available two tools developed during this research.

- **SonarQube Issues Miner**: The source-code for our tool used to mine SonarQube in Chapter 2 is available at: https://github.com/dvmarcilio/sonar-issues-miner
- **SpongeBugs**: The source-code for our tool that fix ASATs' issues automatically and all the data produced by Chapter 3 (e.g., questions for OSS maintainers and submitted pull-requests) are available at: https://github.com/dvmarcilio/spongebugs

## 1.7 Dissertation structure

As introduced in this chapter, the following two chapters answer the main research questions (see Section 1.5) related to the primary goal of this dissertation: aid developers in reducing ASATs' violations by automatically transforming offending source-code. Chapter 4 summarizes the conclusions of these chapters and discusses future research work.

[2]https://docs.oracle.com/javase/8/docs/technotes/guides/language/binary-literals.html

# Chapter 2

# Are Static Analysis Violations Really Fixed? A Closer Look at Realistic Usage of SonarQube

The rapid growth of software development activities over the past decade has increased the focus on the reliability and quality of software systems, which also incurs in associated costs to ensure these characteristics [30]. The use of Automatic Static Analysis Tools (ASATs) is a prominent approach to improve internal quality attributes, as they reveal recurrent code violations without having the cost of running the program [1]. By mainly leveraging heuristic pattern matching approaches to scan source/binary code, these tools can be used for a variety of purposes [26, 20, 31], such as automatically identify refactoring opportunities [32], detect security vulnerabilities [33], highlight performance bottlenecks [34], and bad programming practices, such as code smells [35].

One organization can leverage the benefits of using static analysis tools when they are integrated in the development pipeline—for instance, through the adoption of Continuous Integration (CI) practices [20, 31, 18]. An important principle of CI is continuous inspection, which includes static analysis of source code, among other types of assessments, on every change of the software [18]. Even though the use of ASATs provide several benefits, developers still face challenges when using them [1, 26]. One common reason is the high number of false positive violations, which can reach the thousands as reported by Johnson et al. [1]. Another related barrier is filtering through warnings to find defects that are worth fixing, as violations are often ignored [26].

This chapter was previously published as: Diego Marcilio, Rodrigo Bonifácio, Eduardo Monteiro, Edna Canedo, Welder Luz, and Gustavo Pinto. 2019. Are static analysis violations really fixed?: a closer look at realistic usage of SonarQube. In Proceedings of the 27th International Conference on Program Comprehension (ICPC '19). IEEE Press, Piscataway, NJ, USA, 209-219. DOI: https://doi.org/10.1109/ICPC.2019.00040

Previous works have already investigated how open-source software (OSS) projects take advantage of ASATs (e.g., [6, 21, 27]). For instance, Beller et al. [6] found that ASAT tools are widely used, with most projects relying on a single ASAT tool. Recent studies [21, 27] have focused on what kind of violations developers tend to fix. We challenge this perspective as researchers had to run the static analysis tools themselves on multiple revisions of the projects. As stated by Liu et al. [27], many developers do not use ASATs as part of their development tool chain. Consequently, a piece of code flagged as a fixed issue by these studies may never have been perceived as a violation, and thus fixed unintentionally. We argue that this fact has a significant impact on how developers react to violations. Furthermore, the studies restricted their analysis to OSS projects.

In this chapter we present the results of an in-depth, multi-method study that aims to increase the comprehension of how developers respond to violations reported by ASATs. To achieve this goal, we first conduct a survey with practitioners, in order to better understand the relevance of using static analysis tools and the general procedures developers take to deal with the reported issues. We found that developers consider the use of static analysis tools relevant for improving software quality. Developers also use the outcomes of these tools to decide about postponing a release or accepting / rejecting source code contributions. We then curate and mine a dataset of issues reported from both OSS and industrial projects that actually use the SonarQube ASAT, the leading product for continuous code inspection, used by more than 85,000 organizations.

Our study comprehends 373,413 non-fixed violations and 36,974 fixed violations spanning from 246 Java projects distributed in four distinct SonarQube instances, two from Eclipse (EF) and Apache foundations (ASF)—both well-known Java ecosystems [21, 29]—and two from Brazilian government institutions, the Brazilian Federal Court of Accounts (TCU) and the Brazilian Federal Police (PF). Altogether, in this chapter we answer questions related to (a) *the perceptions of the reported issues* and (b) *the practices for fixing them*. Accordingly, we present the following contributions:

- We present how experienced practitioners use the reports of a static analysis tool.
- We report the results of an in depth analysis of issues and fixes from four different instances of SonarQube.
- We implement and make available an approach for mining issues from SonarQube.
- We make available an extensive dataset[1] of issues from open-source Java projects.

[1]https://doi.org/10.5281/zenodo.2602038

## 2.1 Study Settings

In this section we describe the settings of our study. We first state the goal of our investigation, and then we present details about the research questions we address and the procedures we take to conduct the study and collect issues from the SonarQube instances.

### 2.1.1 Research Goal

The main goal of this chapter is to build a broad comprehension about how developers use the static analysis SonarQube tool, as well as to characterize how they respond to the warnings reported by these tools. Differently from previous works [21, 27], here we focus on both open-source and private organizations.

### 2.1.2 Research Questions

We conduct a multi-method study to investigate the following research questions:

(RQ2.1) What are the practitioners' perceptions about the use of static analysis tools?

(RQ2.2) How often developers fix issues found in open-source and private SonarQube instances?

(RQ2.3) What are the SonarQube issues that developers fix more frequently?

(RQ2.4) How is the distribution of the SonarQube issues? That is, do 20% of the issues correspond to 80% of the fixes? Do 20% of the files lead to 80% of the issues?

To answer RQ2.1 we use a *survey* approach. We explore whether the use of ASATs is relevant to improve software quality, considering the perspective of practitioners. We also use the answers to RQ2.1 to support the discussion about the results of the second study.

To answer the remaining questions we use a *mining software repository* approach. The goal in this case is to comprehend the dynamics for fixing issues reported by SonarQube. The last research question might help practitioners to configure static analysis tools properly, and thus avoid a huge number of false-positives. Moreover, it might also help developers plan their activities in a more effective way, reducing the efforts to improve the internal quality of the systems.

We consider different perspectives to answer these questions, including the characteristics of the systems (e.g., legacy or greenfield systems, private or open-source systems) and the type and severity of the issues. The datasets we use in the investigation include issues from four SonarQube instances, two publicly available, and two private ones.

| ID | Question |
|---|---|
| Q1 | Do you agree that warning messages reported by ASATs are relevant for improving the design and implementation of software? |
| Q2 | How do you fix the issues reported by Automatic Static Analysis Tools? |
| Q3 | How often do you use program transformation tools to automatically fix issues reported by Automatic Static Analysis Tools? |
| Q4 | How important is the use of program transformation tools to fix issues reported by ASATs? |
| Q5 | How often do you reject pull-requests based on the issues reported by ASATs? |
| Q6 | How often do you postpone a release based on the issues reported by ASATs? |

Table 2.1: Survey questions answered by all 18 participants

### 2.1.3 Research Methods

To answer the first research question, we conduct an online survey with developers from the four organizations in which we focus our study. We asked 6 closed questions (see Table 2.1) mainly using a Likert scale [28]. For the OSS foundations we asked for participation on mailing lists, while for the private organizations we reached our personal contacts. The survey was available for approximately one month. Participation was voluntary and all the participants allowed the researcher to use and disclose the information provided while conducting the research. The estimated time to complete the survey was 12-15 minutes. 18 developers, from 81 unique visits (completion rate of 23%), answered all questions of our questionnaire.

The majority of the participants identified themselves, although it was not mandatory. Among the respondents, 50% have more than ten years of experience in software development, 27.77% have between four and ten years, and 22.23% have under four years. Regarding the time using ASATs, 33.33% have more than four years, and the remaining 66.67% have under than four years.

To investigate the practices for fixing issues (RQ2.2) – (RQ2.4), we mine four different SonarQube repositories. We focus on projects from Eclipse Foundation (EF) and Apache Software Foundation (ASF). This decision is based on the work of Izquierdo and Cabot [29], which analyzes 89 software foundations in OSS development and both EF and ASF were the largest in terms of projects they support (216 for EF and 312 for ASF). Moreover, their projects are known for high quality and wide adoption in the OSS community [21].

Our private datasets from Brazilian government institutions are selected not only due to convenience (we got permission to mine their issue databases), but also because they represent a heterogeneous context. TCU does inhouse development whereas PF mostly outsources. More important to this chapter, they both enforce conformity to SonarQube quality checks in their development processes. We restricted our analysis to the Java programming language, since it is the programming language used in the majority of projects available in the selected OSS foundations [36] and is also the primary programming language used in the private datasets. In addition, SonarQube has a very mature analysis for Java projects, with more than 525 rules.

We leverage statistical techniques during the analysis of this chapter, including exploratory data analysis (considering plots and descriptive statistics) and hypothesis testing methods. In particular, we use the non-parametric Kruskal-Wallis hypothesis testing [37] to understand whether or not the severity of a given issue influences the interval in days of the fix. We also use the Dunn test [38] to conduct a multiple comparison of the means. We chose non-parametric methods because our data does not follow a normal distribution. As such, we used the Spearman's rank correlation test [39] to investigate the correlation between variables.

### 2.1.4 SonarQube Data Collection

We implement a tool[2] that is able to extract several data from SonarQube instances. The data collection is done by querying the API provided in the instance itself. One challenge hidden in this activity is to deal with distinct versions of SonarQube, as parameters and responses differ from versions with large disparities. We found that OSS projects rely on older versions of SonarQube: EF uses 4.5.7 (major version from September, 2014) and ASF uses 5.6.3 (major version from June, 2016). Interestingly, those are Long Term Support (LTS) versions. The private instances rely on newer versions (the 7.x, released after 2018). None of them is a LTS version though, although they can be queried in the same fashion. The data collection took place during the months of November / December of 2018, though we updated the datasets also in January 2019.

For each SonarQube instance, we gather data for rules, projects, and their issues. As aforementioned, rules indicate whether instances use customized rules or not. Even though SonarQube encompasses rules from other ASATs, such as FindBugs and CheckStyle, we found that EF and TCU use a significant number of customized rules from these ASATs. We filter out 28 projects to remove branches that are considered as projects in SonarQube, a situation particular in the TCU's repository. The next step collects issues: open, fixed, won't fix, and false-positives. To filter out non-desired projects, such as toy projects, inactive and demos [40], we apply a filter to consider only projects with at least one Java fixed issue. We removed 31 projects from EF, 21 from ASF, 62 from PF and 157 from TCU when applying this filter. Table 2.2 presents an overview of the whole dataset.

Overall we collected data from 246 Java software projects. Altogether, these software projects employed 4,319 rules (2,086 distinct ones). Still, these projects had reported a total of 421,976 issues (373,413 labeled as open, 36,974 as fixed, and 11,589 as won't fix or false positive).

[2] https://github.com/dvmarcilio/sonar-issues-miner

| Org. | Rules | Projects | Filtered Projects | Open Issues | Fixed Issues | WF + FP |
|---|---|---|---|---|---|---|
| EF | 1,493 | 95 | 64 | 29,547 | 6,993 | 952 |
| ASF | 397 | 48 | 27 | 136,235 | 16,731 | 10,168 |
| TCU | 1,998 | 283 | 98 | 165,304 | 10,021 | 467 |
| PF | 530 | 119 | 57 | 42,327 | 3,229 | 2 |
| Total | 2,086[a] | 545 | 246 | 373,413 | 36,974 | 11,589 |

[a] Distinct rules

Table 2.2: Overview of the collected data. The last column indicates quantity of (W)on't (F)ix and (F)alse-(P)ositive issues.

## 2.2 Results

In this section we present the main findings of this chapter, answering the general research questions we investigate.

### 2.2.1 What are the practitioners' perceptions about the use of static analysis tools? (RQ2.1)

The findings from our survey indicate that: (a) developers consider ASATs warnings as relevant for overall software improvement (Q1), and (b) developers typically fix the issues along the implementation of bug fixes and new features (Q2), i.e., without creating specific tasks / branches for fixing the reported issues. Perhaps due to the small effort to fix part of the issues, this task does not become a first class planned activity, and thus may not require their own development branches.

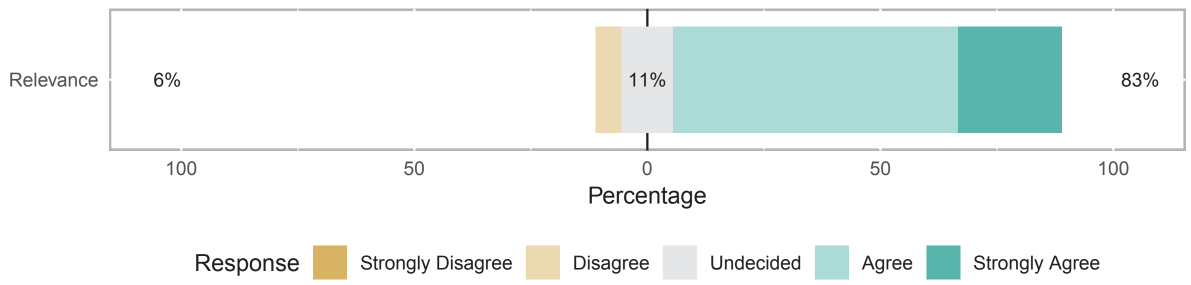

Figure 2.1: Do you agree that warning messages reported by ASAT tools are relevant for improving the design and implementation of software? (Q1)

More than 80% of the respondents said that they agree or strongly agree that the issues reported by ASAT tools are relevant for improving the design and implementation of software (Q1), as shown in Figure 2.1. Moreover, as Figure 2.2 shows, only 22.22% reject pull-requests based on the issues reported (Q5), and 50% never or rarely postpone a release based on the issues reported (Q6).

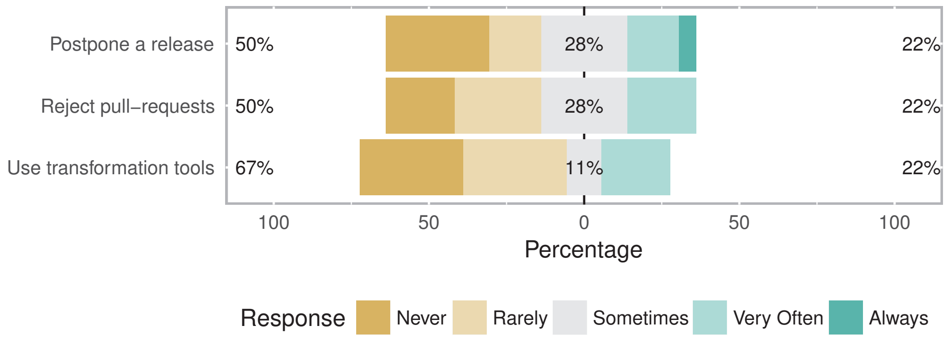

Figure 2.2: Survey responses on whether respondents postpone a release (Q6), reject pull-requests (Q5), or use transformation tools (Q3).

We find an apparent contradiction between the importance that developers claim ASATs and program transformation tools to automatically fix issues have (Q4), and how these tools are used by them during the software development process (Q3), we detail it in the next two paragraphs. Regarding the use of transformation tools to automatically fix issues reported by ASATs, more than 60% said that they consider them important or very important (see Figure 2.3). But, when asked about how often they actually use this type of tool for this purpose, only 22.22% answered very often, against 66.66% never or rarely, as shown in Figure 2.2. This might indicate that issues are not always solved in batch, which would benefit from the use of automatic program transformation tools. This also suggests that it would be worth to develop program transformation tools that address issues reported by static analysis tools and that could be integrated into continuous integration workflows.

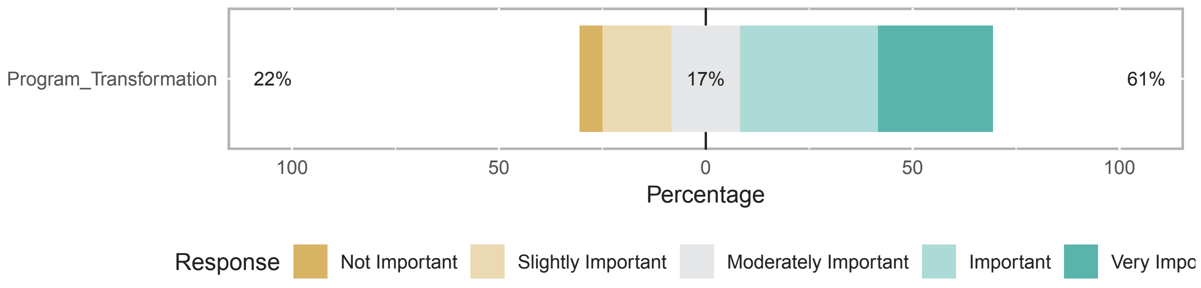

Figure 2.3: How important is the use of program transformation tools to fix issues reported by ASATs? (Q4)

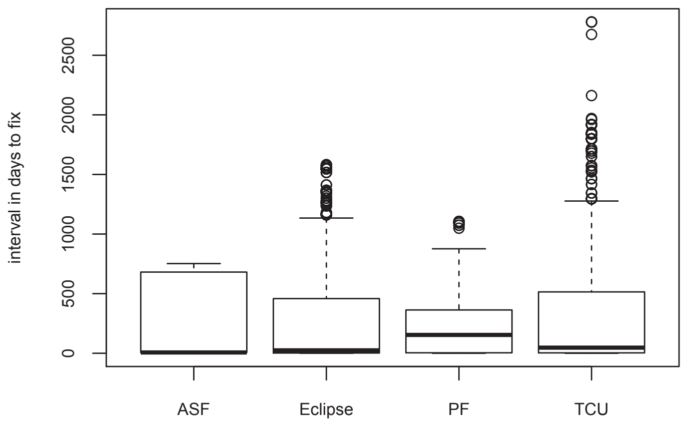

Figure 2.4: Descriptive statistics with the interval in days to fix reported issues (grouped by organization)

### 2.2.2 How often developers fix issues found by SonarQube? (RQ2.2)

To answer this research question we first investigate the interval in days between the date a given issue was created and the date it was closed. Here we only consider the 36,974 fixed issues (8.76% of all issues). The issues that developers quickly fix might represent relevant problems that should be fixed in a short period of time or could be a potential target for automation.

Figure 2.4 presents some descriptive statistics, considering the four organizations. Considering all projects, the median interval in days for fixing an issue is 18.99 days. There is a (median) central tendency of ASF developers to fix issues in 6.67 days (more details in Table 2.3). Interestingly, the industrial projects studied take much longer to fix the SonarQube reported issues, when compared to the OSS studied projects. In a previous work, Panjer and colleagues reported that Eclipse bugs are fixed in 66.2 days on average [41]. Here we found that developers spend on average more time to fix issues reported by SonarQube (see Table 2.3). In addition, we found 10,749 issues (29% of them) that were fixed after one year of the report—many of them are considered **major** issues (see Figure 2.7). These "long to be fixed" issues were found across all four organizations (ASF: 4,760, EF: 2,310, TCU: 2,954, PF: 726). It is also important to note that almost 50% of the open issues have been reported more than one year ago.

*Altogether, our first conclusion is that the number of fixes is relatively small (8.77% of the total of issues), a lower value than reported by previous research. Developers often fix the issues in a reasonable time frame (median is 18.99 days), also a shorter period than previously reported periods for both bugs and ASATs violations. In addition, we found that almost one-third of the fixes occurs after one year the issue had been reported.*

| Organization | Median | Mean | Sd |
|---|---|---|---|
| EF | 24.59 | 299.11 | 435.19 |
| ASF | 6.67 | 222.16 | 298.73 |
| TCU | 47.14 | 282.60 | 435.43 |
| PF | 153.81 | 216.60 | 241.31 |

Table 2.3: Descriptive statistics related to the number of days for fixing issues

We used the approach proposed by Giger et al. [42] for classifying the resolutions time. It has only two status: *Fast* (fix interval $\leq$ median time to fix) and *Slow* (fix interval $>$ median time to fix). Since the median interval for fixing an issue is 18.99 days, we used this information to characterize our dataset of fixed issues in the plot of Figure 2.5. As one could see, most fixes (55%) related to ASF projects present a *Fast* resolution while the PF organization presents the slowest scenario, with 66% of its resolutions being considered *Slow*.

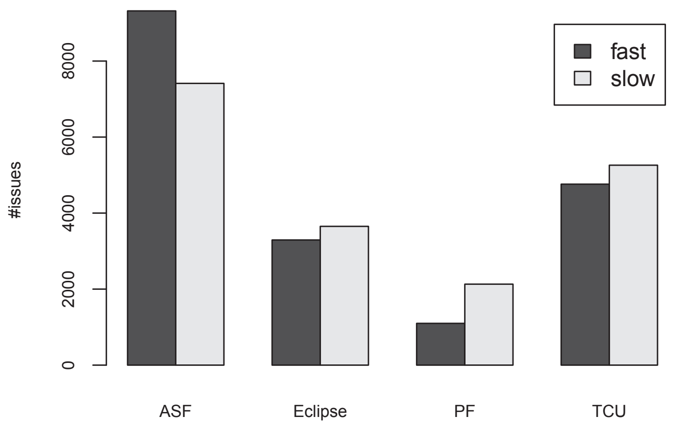

Figure 2.5: Speed resolutions of the organizations

To better understand why the reported issues are taking too long to be fixed at PF, we present a closer look at PF issues resolutions in Table 2.4.

| Projects category | Fix Class | Total |
|---|---|---|
| Greenfield[a] | Fast Resolutions | 794 (24,6%) |
| Greenfield | Slow Resolutions | 1068 (33,1%) |
| Legacy[b] | Fast Resolutions | 305 (9,4%) |
| Legacy | Slow Resolutions | 1062 (32,9%) |

[a] 26 projects
[b] 31 projects

Table 2.4: Speed resolutions for PF's different lines of software development projects

In our collaboration with PF, we found out that they work on two lines of software projects: one that maintains and evolves legacy software systems (mainly developed in Java 6), and another one that develops greenfield software systems, working with newer technologies (e.g., Java 8). This particular greenfield line of work also follows agile practices with monthly deliveries. Our analysis confirms the intuition that greenfield projects fix issues faster, and also have more issues fixed in total (1,862 for 26 greenfield projects *vs* 1,367 for 31 legacy projects). Conversely, legacy projects have a significantly larger number of open issues (27,599 *vs* 14,728 from greenfield projects).

Moreover, we also investigate whether or not the "severity" of the issues influences the interval in days for fixing the reported problems (see the boxplots in Figure 2.8). To further investigate this aspect, we executed the non-parametric *Kruskal-Wallis test* and the *Dunn test* method for comparing mean differences, which (a) reveal that the severity factor influences the time for fixing issues ($p$–value $< 2.2e^{-16}$), and (b) give evidence that the Blocker and Minor severity categories are fixed in less time than the other categories. Figure 2.6 shows the outcomes of the *Dunn test*. We actually found surprising the observation that Minor issues have been fixed faster than Major and Critical issues. A possible reason might be that Minor issues are simpler to solve than the other issues.

```
Col Mean-|
Row Mean | BLOCKER CRITICAL INFO MAJOR
---------+------------------------------------------
CRITICAL | -10.48
         | 0.0000*
         |
    INFO | -9.83 -0.32
         | 0.0000* 0.3717
         |
   MAJOR | -3.19 23.13 14.30
         | 0.0007* 0.0000* 0.0000*
         |
   MINOR | -0.69 29.67 18.60 12.48
         | 0.2445 0.0000* 0.0000* 0.0000*

alpha = 0.05
Reject Ho if p <= alpha/2
```

Figure 2.6: Mean differences of the interval in days to fix issues, considering the severity of the issues

*Blocker and Minor issues are solved faster than the other categories.*

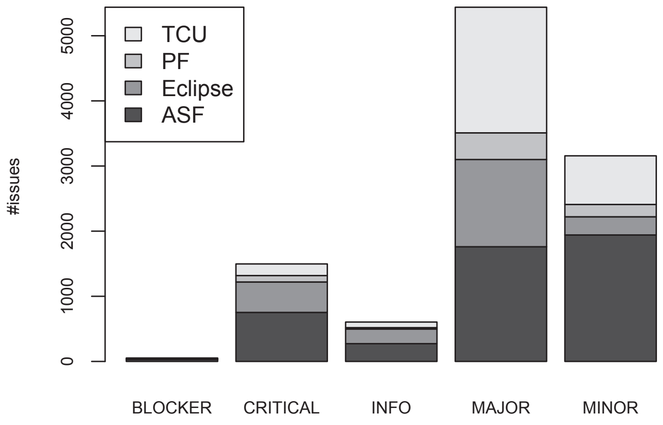

Figure 2.7: Number of issues fixed after one year they had been reported

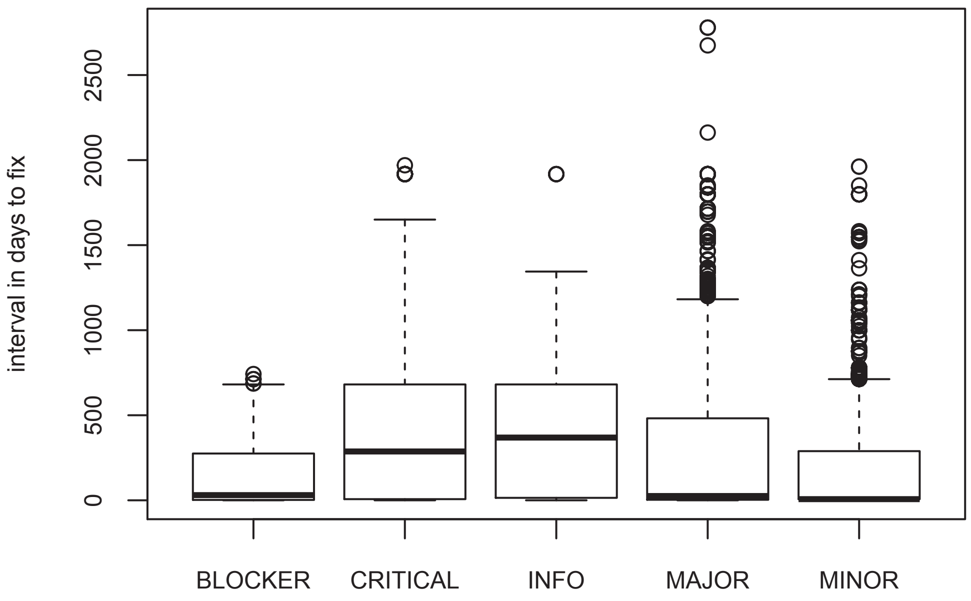

Figure 2.8: Descriptive statistics with the interval in days to fix reported issues (grouped by severity)

In order to further investigate our first research question, we also considered the frequency in which developers fix the issues reported by SonarQube. To this end, we computed a number of metrics for each project $P$.

**MinDate(P)** The first date an issue have been reported for a project $P$.

**MaxDate(P)** The last date an issue have been reported for a project $P$.

**Interval(P)** The difference between the *max* and *min* dates for a project $P$ (computed as $MaxDate(P) - MinDate(P) + 1$).

**ODD(P)** The number of distinct dates in which at least one issue has been opened in a project $P$.

**FDD(P)** The number of distinct dates in which at least one issue has appeared as fixed in a project $P$.

**OpenFreq(P)** The frequency in the interval where at least one issue have been opened in a project $P$ (computed as $ODD(P) * 100/Interval(P)$).

**FixFreq(P)** The frequency in the interval where at least one issue have been fixed in a project $P$ (computed as $FDD(P) * 100/Interval(P)$).

Table 2.5 summarizes some descriptive statistics related to these metrics. Based on **Interval(P)**, it is possible to realize that most projects in our dataset are using SonarQube for at least one year (median of **Interval(P)** is 415 days). Considering the full interval in days where the projects were using SonarQube, on average, issues have been reported in 15.47% of the days. A possible explanation is that, when a project starts using a static analysis tool (like SonarQube), several issues are reported at once. After that, while the development of a system makes progress, the frequency in which new flaws are introduced and reported becomes sparse (with seasonal peaks where many flaws are reported). Surely, this might also indicates either that SonarQube is not integrated into the development processes or that there is a lack of activity.

More interesting is that issues are less frequently fixed than they are reported—that is, on average, we found fixes in 9.91% of the days between **MinDate(P)** and **MaxDate(P)** for a given project $P$. We investigate the correlation (using the Spearman's method) between the total of distinct days the issues have been fixed (**FDD(P)**) and the total number of fixed issues of a project $P$. We find a moderate correlation within all organizations (EF with the least has 0.59, whereas ASF had the maximum of 0.69). This does not support the argument that issues are often fixed in "batch". Batch examples in the ASF ecosystem are projects *LDAP_API*, with 15 distinct dates and 6,832 fixed issues, and *Myfaces*, with 7 distinct dates and 3,856 fixed issues.

| Metric | Median | Mean | SD |
|---|---|---|---|
| **Interval(P)** | 415 days | 667 days | 772.71 |
| **OpenFreq(P)** | 4.71% | 15.47% | 26.07% |
| **FixFreq(P)** | 0.67% | 9.91% | 25.61% |

Table 2.5: Some descriptive statistics related to the frequency based metrics

*Therefore, the frequency in which new issues are either reported or fixed is relatively sparse, which might indicates that (1) SonarQube is not part of continuous integration workflows or that (2) developers do not act immediately when a new issue arises. Based on our findings, we can conclude that developers rarely fix issues reported by SonarQube.*

Finally, we also investigate if there are specific days in which developers fix more issues. To this end, we collect the total number of issues fixed in each day of the week for each organization. Table 2.6 reports the results. We found many fixes of TCU appearing on Saturdays (22.5% of them) and many fixes of the Eclipse Foundation appearing on Sundays (22.8%). Overall, 12.4% of the fixes occurred during the weekends (EF: 40%, ASF: 3.3%, TCU: 25.9%, and PF: 0%). Contrasting to the other organizations whose fixes appear more frequently during the weekdays. Actually, PF does not have any issues fixed on the weekends.

| Org. | Sun | Mon | Tue | Wed | Thu | Fri | Sat |
|---|---|---|---|---|---|---|---|
| EF | 1,294 | 1,284 | 605 | 1,132 | 923 | 1,031 | 699 |
| ASF | 383 | 724 | 7,470 | 1,520 | 5,515 | 961 | 158 |
| TCU | 218 | 1,015 | 1,390 | 1,874 | 2,253 | 1,428 | 1,843 |
| PF | 0 | 949 | 78 | 1,400 | 219 | 583 | 0 |
| Total | 1,895 | 3,972 | 9,543 | 5,926 | 8,910 | 4,003 | 2,700 |

Table 2.6: Issues fixed per day of the week

### 2.2.3 What are the SonarQube issues that developers fix more frequently? (RQ2.3)

In our study we mined 36,974 fixed issues from all organizations. Even though the number of fixed violations is significantly low in comparison to the number of open violations, we still find some open issues that are worth fixing. To answer this research question, we studied the rules associated to the reported issues. Eclipse employs a deprecated rule by the SonarQube team which says that *Cycles between packages should be removed.* Since this rule is frequently fixed on EF's dataset, we introduce the type *Deprecated* to classify it. We found that both EF and ASF modify configurations for rules, either activating/deactivating rules, or changing rules severities. For instance, EF deactivates the rule *Useless imports shoud be removed*, which is the 10th most fixed rule in our dataset. As a result, this rule is not related to any violations in EF.

| Org. | Code Smell | Vulnerability | Bug | Deprecated |
|---|---|---|---|---|
| EF | 2,533 | 399 | 116 | 3,945 |
| ASF | 15,077 | 322 | 1,332 | – |
| TCU | 8,837 | 677 | 507 | – |
| PF | 2,968 | 60 | 201 | – |
| Total | 29,415 (79.6%) | 1,458 (3.9%) | 2,156 (5.8%) | 3,945 (10.7%) |

Table 2.7: Most frequently fixed issues by type in each organization

At first, we quantify all fixed issues from all organizations' projects. We then classify them by their type, as shown in Table 2.7. It is possible to see that *Code Smells* are responsible for a high percentage (almost 80%) of all fixed issues. As we show in Table 2.8, *Minor* issues are responsible for 21% of the fixed issues, and *Info* issues for 2.5%. We also find contrasting results regarding vulnerabilities as in our study: they represent approximately 4% of the total number of fixes, when compared to 0.2% and 0.5% reported in [21] and [27], respectively.

| Severity | Code Smell | Vulnerability | Bug | Deprecated | Total |
|---|---|---|---|---|---|
| Major | **19,732** | 496 | **972** | **3,945** | 25,145 (68%) |
| Minor | 7,683 | 53 | 91 | – | 7,827 (21.2%) |
| Critical | 943 | **883** | 697 | – | 2,523 (6.8%) |
| Info | 944 | 6 | – | – | 950 (2.6%) |
| Blocker | 113 | 20 | 396 | – | 529 (1.4%) |

Table 2.8: Most frequently fixed issues classifying severity to type

Table 2.9 presents the ten most frequently fixed issues. It is worthy noting that the five most fixed *Minor* issues correspond to almost 17% of the total fixed issues. Not surprisingly, *Code Smells* and *Major* issues are prevalent in the selection. Although code smells is the most fixed issues type, it is also responsible for the ten most frequent open issues, with six of them having a Major severity. Since Major issues can highly impact developers of a system (see **??**) and also represent a predominantly large portion (68%) of the fixed issues, we question whether ASATs issue prioritization is as ineffective as reported in related works [27, 43].

We find that a frequently fixed issue does not incur in high fixing rate. We found that two issues, *Sections of code should not be commented out* and *Generic exceptions should never be thrown*, are also present in the ten most common opened issues. If we consider 20 issues for both most fixed and most opened, there are 9 common issues between the two lists.

Finally, we investigate the occurrence of *Won't fix* and *False-positive* issues. We argue that marking an issue as one of these resolutions is similar to the process of fixing a violation. The developer has to filter the specific issue among all others, assess if it truly represents a quality flaw worthy of fixing, and then she must take an action. With that in mind, developers do tend to flag issues as won't fix and/or false-positive. Apache's projects flagged a total of 10.168 issues with these resolutions. We encounter similar findings when comparing ASF's ten most issues flagged as won't fix/false-positive and the foundation's ten most opened issues. There is a common subset of 5 issues among the two. These findings suggests that no rule is always fixed, regardless of context. Developers seem to consider other factors to decide whether to fix an issue or not.

*Code smells and Major issues are highly prevalent among most of all issues' types and severities. We found common issues among the top ten most fixed, wont't fix / false-positive, and opened issues. This suggests that developers consider a variety of factors when deciding whether to fix an issue.*

EF and TCU SonarQube instances have a large number of customized rules. When comparing those rules to rules that are available in a fresh SonarQube installation, EF has 1.163 additional rules, and TCU has 1.533. Nonetheless, we found that just a subset of these rules actually lead to issues' reports. Overall, 122 unique rules are associated to fixed issues in EF, with 104 custom rules, which represents 7% of the total of custom rules. In TCU 250 unique rules are associated to fixed issues, with 141 custom rules, or 9% of TCU's custom rules.

Table 2.9: Most frequently fixed issues in all organizations

| Issue | Type | Severity | Count | EF | ASF | PF | TCU |
|---|---|---|---|---|---|---|---|
| Cycles between packages should be removed | D[a] | Major | 3,945 (10.7%) | 3,945 (100%) | 0 (0%) | 0 (0%) | 0 (0%) |
| Checked exceptions should not be thrown | CS[b] | Major | 2,053 (5.5%) | 0 (0%) | 0 (0%) | 0 (0%) | 2,053 (100%) |
| Sections of code should not be commented out | CS | Major | 1,903 (5.1%) | 182 (9.6%) | 1,014 (53.3%) | 364 (19.1%) | 343 (18%) |
| The diamond operator should be used | CS | Minor | 1,871 (5%) | 0 (0%) | 1,716 (91.7%) | 155 (8.3%) | 0 (0%) |
| Nested code blocks should not be used | CS | Minor | 1,380 (3.7%) | 0 (0%) | 1,374 (99.6%) | 6 (0.4%) | 0 (0%) |
| throws declarations should not be superfluous | CS | Minor | 1,352 (3.6%) | 0 (0%) | 623 (46.1%) | 77 (5.7%) | 652 (48.2%) |
| Generic exceptions should never be thrown | CS | Major | 1,334 (3.6%) | 59 (4.4%) | 129 (9.7%) | 20 (1.5%) | 1,126 (84.4%) |
| Redundant pairs of parentheses should be removed | CS | Major | 961 (2.6%) | 0 (0%) | 741 (77.1%) | 109 (11.3%) | 111 (11.6%) |
| Local variable and method parameter names should comply with a naming convention | CS | Minor | 823 (2.2%) | 16 (2%) | 774 (94%) | 33 (4%) | 0 (0%) |
| Useless imports should be removed | CS | Minor | 784 (2.1%) | 0 (0%) | 517 (66%) | 180 (23%) | 87 (11%) |

[a] Deprecated
[b] Code Smell

## 2.2.4 How is the distribution of the SonarQube issues? (RQ2.4)

Taking into account the results of the previous section, here we answer our fourth research question, which investigates the concentration of the rules (20% of the rules correspond to 80% of the fixes) and the concentration of the files (20% of the files concentrate 80% of the issues). Answering to this question might help practitioners (a) to select a subset of rules that should be fixed (for instance, due to its relevance for source code improvement or easiness of fixing) or (b) to concentrate quality assurance activities in certain files of a project.

Considering all projects, we found a total of 412 rules having at least one fix. In this way, we consider the 82 most frequent fixed rules to answer RQ2.4—where 82 corresponds to 20% of the 412 rules. These 82 rules are related to 32,717 fixes. Since our dataset comprises 36,959, the 20% most frequent fixed rules correspond to 88.52% of all fixed issues. We publish this list of most frequent fixed rules in the paper's website (omitted here due to the blind review process).

We further analyse our dataset to verify which projects follow the distribution *20% of the rules correspond to 80% of the fixes).* To avoid bias due to a small number of fixes, we constrain our analysis to projects having at least 16 fixes and 190 files (the median number of fixes and files per project, respectively), leading to a total of 80 projects. We found 62 projects (77.5%) in the rule 20% of the rules correspond to 80% of the fixes, which suggests that it would be possible to reduce the number of reported issues (and avoid false-positives and issues that would not be fixed) by correctly configuring SonarQube to report a relatively small subset of all rules—those issues that are more likely to be fixed.

Another recurrent question that arises in the literature [44, 45, 46] is whether or not 20% of the modules (files) are responsible for 80% of the issues (bugs in the existing literature). Investigating this issue might not only help managers to concentrate quality assurance activities on a subset of the modules of a project, but also might open new research directions to predict which files are more expected to present design flaws. More precisely, here we investigate if 20% of the files of each project (with at least 16 fixes and 190 files in our dataset) concentrate at least 80% of the issues. Interesting, we did not find any project satisfying this distribution. Considering the median statistic, the top 20% of files containing more issues represent 35.79% of all issues of a project (mean: 37.23 and max: 63.37). Comparing with the literature aforementioned, which suggests a higher concentration of bugs, we can conclude that static analysis issues are more widespread throughout the modules of a system than bugs.

*We found that 20% of the rules correspond to 80% of the fixes, and that the issues reported by static analysis tools are not localized in a relatively small subset of the files of the projects.*

## 2.3 Discussion

Our findings show contrasting results at first. Practitioners find ASATs reports relevant to the software development process, and, in some situations, reject pull-requests or even postpone the release of a software based on the outcomes of these tools.

Our investigation also reveals that the resolution time for fixing issues is faster than the time previously reported for fixing bugs. Although these results strongly support that developers indeed use ASATs and take their warnings in consideration, we find that fixed issues only represent 8.76% of the 421,976 mined issues, which suggests that not all issues are relevant to developers, as supported by the finding that 20% of the rules correspond to 80% of the fixes.

Our results also indicate that practitioners can greatly benefit from the usage of ASATs if they properly configure them to mostly consider rules that they find relevant and are more likely to fix. This might help to control the pressure related to the technical debt of the systems, often calculated using ASAT reports. Developers could also benefit from tool support to fix ASATs issues, since most of them consider important the use of tools that provide automatic fixes, but at the same time most never or rarely use them. We envision that our findings, such as the big prevalence of fixed *Code Smells* and Major issues, can provide insights to tools developers.

Our findings still unfold several unanswered observations pertaining to the comprehension on how developers fix and perceive ASAT issues. An organization, or a particular team or project, might have a policy to fix all major issues, thus impacting on which kind of violations are fixed. In cases that ASATs are integrated in the development workflow, several reasons for developers not fixing issues are possible, such as lack of configuration, unawareness on how to perform fixes, value of fixing an issue, or even time pressure. These open questions suggest future research focused on organizations, and/or teams / projects, that use ASATs as part of their workflow.

Finally, mining issues from SonarQube can be challenging, specially when considering different instance versions and different host organizations. To help further research aiming at mining SonarQube issues, we recommend researchers to mine rules for the chosen language(s). As an example, EF most fixed rule was a custom one, that would not be analyzed if rules were not mined or a revision approach was used.

## 2.4 Conclusion

In this chapter we reported the results of a multi-method study about how developers use SonarQube (one of the most used tools for static quality assurance). We first collected the perceptions of 18 developers from different organizations, regarding the use of static analysis tools. Most respondents of the survey agree that these tools are relevant for the overall improvement of software quality. By mining four instances of SonarQube, we built a general comprehension of the practices for fixing issues that this tool reports. We found a low rate of fixed issues and that one-third of the fixes occurs after one year of the issue's report. In addition, we showed evidences that 20% of the violation rules correspond to 80% of the fixes, which can assist practitioners to properly select a subset of rules that are relevant, and discard rules that are rarely fixed.

# Chapter 3

# Automatically Generating Fix Suggestions in Response to Static Code Analysis Warnings

Static code analysis tools (SATs) are becoming increasingly popular as a way of detecting possible sources of defects earlier in the development process [4]. By working *statically* on the source or byte code of a project, these tools are applicable to large code bases [47, 5], where they quickly search for patterns that may indicate problems—bugs, questionable design choices, or failures to follow stylistic conventions [48, 49]—and report them to users. There is evidence [6] that using these tools can help developers monitor and improve software code quality; indeed, static code analysis tools are widely used for both commercial and open-source software development [50, 4, 5]. Some projects' development rules even require that code has to clear the checks of a certain SAT before it can be released [50, 6, 3].

At the same time, some features of SATs limit their wider applicability in practice. One key problem is that SATs are necessarily *imprecise* in checking for rule violations; in other words, they report *warnings* that may or may not correspond to an actual mistake. As a result, the first time a static analysis tool is run on a project, it is likely to report thousands of warnings [4, 47], which saturates the developers' capability of sifting through them to select those that are more relevant and should be fixed [50]. Another, related issue with using SATs in practice is that understanding the problem highlighted by a warning and coming up with a suitable fix is often nontrivial [50, 47].

This chapter was accepted for publication at SCAM 2019 as: Diego Marcilio, Carlo A. Furia, Rodrigo Bonifacio, Gustavo Pinto, "Automatically Generating Fix Suggestions in Response to Static Code Analysis Warnings" 2019 IEEE 19th International Working Conference on Source Code Analysis and Manipulation (SCAM), Cleveland, 2019

This chapter aims at improving the practical usability of SATs by automatically providing *fix suggestions*: modifications to the source code that make it compliant with the rules checked by the analysis tools. We developed an approach, called SpongeBugs and described in 3.1, whose current implementation works on Java code. SpongeBugs detects violations of 11 different rules checked by SonarQube and SpotBugs (successor to FindBugs [4])—two well-known static code analysis tools, routinely used by very many software companies and consortia, including large ones such as the Apache Software Foundation and the Eclipse Foundation. The rules checked by SpongeBugs are among the most widely used in these two tools, and cover different kinds of code issues (ranging from performance, to correct behavior, style, and other aspects). For each violation it detects, SpongeBugs automatically generates a fix suggestion and presents it to the user.

By construction, SpongeBugs's suggestions remove the origin of a rule's violation, but the maintainer still has to decide—based on their overall knowledge of the project—whether to accept and merge each suggestion. To assess whether developers are indeed willing to accept SpongeBugs's suggestions, section 3.3 describes the results of an empirical evaluation where we applied SpongeBugs to 12 Java projects, and submitted 920 fix suggestions as pull requests to the projects. At the time of writing, project maintainers accepted 775 (84%) fix suggestions—95% of them without any modifications. This high acceptance rate suggests that SpongeBugs often generates patches of high quality, which developers find adequate and useful. The empirical evaluation also indicates that SpongeBugs is applicable with good performance to large code bases; and reports (in 3.3.4) several qualitative findings that can inform further progress in this line of work.

The work reported in this chapter is part of a large body of research (see **??**) that deals with helping developers detecting and fixing bugs and code smells. SpongeBugs' approach is characterized by the following features:

*i*) it targets static rules that correspond to frequent mistakes that are often fixable *syntactically*;

*ii*) it builds fix suggestions that remove the source of warning *by construction*;

*iii*) it scales to large code bases because it is based on lightweight program transformation techniques.

Despite the focus on conceptually simple rule violations, SpongeBugs can generate non-trivial patches, including some that modify multiple hunks of code at once. In summary, SpongeBugs's focus privileges generating a large number of practically useful fixes over being as broadly applicable as possible.

## 3.1 SpongeBugs: Approach and Implementation

SpongeBugs provides fix suggestions for violations of selected rules that are checked by SonarQube and SpotBugs. 3.1.1 discusses how we selected the rules to check and suggest fixes for. SpongeBugs works by means of source-to-source transformations, implemented as we outline in 3.1.2.

### 3.1.1 Rule Selection

One key design decision for SpongeBugs is which static code analysis rules it should target. Crucially, SATs are prone to generating a high number of false positives [51]. To avoid creating fixes to spurious warnings, we base our choice on the assumption that rules whose violations are frequently fixed by developers are more likely to correspond to real issues of practical relevance [50, 27].

We collected and analyzed the publicly available datasets from three previous studies that explored developer behavior in response to output from SonarQube [50, 21] and FindBugs [27]. Based on this data, we initially selected the top 50% most frequently fixed rules, corresponding to 156 rules, extended with another 10 rules whose usage was not studied in the literature but appear to be widely applicable.

Then, we went sequentially through each rule, starting from the most frequently fixed ones, and manually selected those that are more amenable to automatic fix generation. The main criterion to select a rule is that it should be possible to define a syntactic fix template that is guaranteed to remove the source of warning without obviously changing the behavior. This led to discarding all rules that are not *modular*, that is, that require changes that affect clients in any files. An example is the rule *Method may return* `null`*, but its return type is* `@Nonnull`.[1] Although conceptually simple, the fix for a violation of this rule entails a change in a method's signature that weakens the guarantees on its return type. This is both impractical, since we would need to identify and check every call of this method, and potentially introduce a breaking change [52]. We also discarded rules when automatically generating a syntactic fix would be cumbersome or would require additional design decisions. An example is the rule *Code should not contain a hard coded reference to an absolute pathname*, whose recommended solution involves introducing an environment variable. To provide an automated fix for this violation, our tool would need an input for the developer, since a pathname is very context specific; it would also need to have access to the application's execution environment, which is clearly beyond the scope of the source code it has access to.

[1] https://spotbugs.readthedocs.io/en/latest/bugDescriptions.html#np-method-may-return-null-but-is-declared-nonnull-np-nonnull-return-violation

We selected the top rules (in order of how often developers fix the corresponding warnings) that satisfy these feasibility criteria. This corresponds to the 11 rules listed in 3.1. Note that SonarQube and SpotBugs rules largely overlap, but the same rule may be expressed in slightly different terms in each tool. Since SonarQube includes all 11 rules we selected, whereas SpotBug only includes 7 of them, we use SonarQube rule names[2] for uniformity throughout the paper.

| ID | RULE DESCRIPTION |
|---|---|
| B1 | Strings and boxed types should be compared using `equals()` |
| B2 | `BigDecimal(double)` should not be used |
| C1 | String literals should not be duplicated |
| C2 | String functions use should be optimized for single characters |
| C3 | Strings should not be concatenated using `+` in a loop |
| C4 | Parsing should be used to convert strings to primitive types |
| C5 | Strings literals should be placed on the left-hand side when checking for equality |
| C6 | Constructors should not be used to instantiate `String`, `BigInteger`, `BigDecimal`, and primitive wrapper classes |
| C7 | `entrySet()` should be iterated when both key and value are are needed |
| C8 | `Collection.isEmpty()` should be used to test for emptiness |
| C9 | `Collections.EMPTY_LIST`, `EMPTY_MAP`, and `EMPTY_SET` should not be used |

Table 3.1: The 11 static code analysis rules that SpongeBugs can provide fix suggestions for. The rule descriptions are based on SonarQube's, which classifies rules in (B)ugs and (C)ode smells.

Rule C1 was not chosen from the dataset but it is very meaningful for our purpose—since it appears frequently in open issues in the datasets. The fix for this C1 rule involves multiple lines, along with the insertion of a constant. Rule C5 was also not present as one of the most fixed rules, but we choose it as it can be applied in conjunction with rule B1 (see Listing 3.1), making the code shorter while also avoiding `NullPointerException` from being thrown.

```
- if (render != null && render != "")
+ if (!"".equals(render))
```

Listing 3.1: Fixes for rules B1 (Strings and Boxed types should be compared using "equals()") and C5 (Strings literals should be placed on the left side when checking for equality) applied in conjunction.

[2] https://rules.sonarsource.com/java

Consistently with SonarQube's classification of rules, we assign an identifier to each rule according to whether it represents a bug (B1 and B2) or a code smell (C1–C9). While the classification is fuzzy and of limited practical usefulness, note that the most of our rules are code smells in accordance with the design decisions behind SpongeBugs.

### 3.1.2 How SpongeBugs Works

SpongeBugs looks for rule violations and builds fix suggestions in three steps:

1. Find textual patterns that might represent a rule violation;
2. For every match identified in step 1, perform a full search in the AST looking for rule violations;
3. For every match confirmed in step 2, instantiate the rule's fix templates—producing the actual fix for the rule violation.

We implemented SpongeBugs using Rascal [53], a domain-specific language for source code analysis and manipulation. Rascal facilitates several common meta-programming tasks, including a first-class visitor language constructor, advanced pattern matching based on concrete syntax, and defining templates for code generation. We used Rascal's Java 8 grammar [54], which entails that our evaluation (3.3) is limited to Java projects that can be built using this version of the language.

We illustrate how SpongeBugs's three steps work for rule C8 (`Collection.isEmpty()` *should be used to test for emptiness*). Step 1 performs a fast, but potentially imprecise, search that is based on some textual necessary conditions for a rule. For rule C8, step 1 looks for files that import some package in `java.util` *and* include textual patterns that indicate a comparison of `size()` with zero—as shown in Listing 3.2.

```
bool shouldContinueWithASTAnalysis(loc fileLoc) {
  javaFileContent = readFile(fileLoc);
  return findFirst(javaFileContent, "import java.util.") != -1 &&
    hasSizeComparison(javaFileContent);
}

bool hasSizeComparison(str javaFileContent) {
  return findFirst(javaFileContent, ".size() \> 0") != -1 ||
    findFirst(javaFileContent, ".size() \>= 1") != -1 ||
    findFirst(javaFileContent, ".size() != 0") != -1 ||
    findFirst(javaFileContent, ".size() == 0") != -1;
}
```

Listing 3.2: Implementation of step 1 for rule C8: find textual patterns that might represent a violation of rule C8.

Step 1 may report false positives: for rule C8, the comparison involving `size()` may not actually involve an instance of a collection, which may not offer a method `isEmpty()`. Therefore, step 2 is more computationally expensive since it performs a full AST matching, but is only applied after step 1 identifies code that has a high likelihood of being rule violations. In our example of rule C8, step 2 checks that the target of the possibly offending call to `size()` is indeed of type `Collection`–as shown in Listing 3.3.

```
case (EqualityExpression)
  ‘<ExpressionName beforeFunc>.size() == 0‘: {
  if (isBeforeFuncReferencingACollection(beforeFunc,
    mdl, unit)) {
```

Listing 3.3: Partial implementation of step 2 for rule C8: full AST search for rule violations.

Whenever step 2 returns a positive match, step 3 executes and finally generate a patch to fix the rule violation. Step 3's generation is entirely based on *code-transformation templates* that modify the AST matched in step 2 as appropriate according to the rule's semantics. For rule C8 step 3's template is straightforward: replace the comparison of `size() == 0` with a call to `isEmpty()`—its implementation is in Listing 3.4. Note that other patterns are transformed, but we only show one for brevity purposes.

```
refactoredExp = parse(#Expression, "<beforeFunc>.isEmpty()");
```

Listing 3.4: Implementation of step 3 for rule C8: instantiate the fix templates corresponding to the violated rule.

## 3.2 Empirical Evaluation of SpongeBugs: Experimental Design

The general goal of this chapter is to investigate the use of techniques for fixing suggestions to address the warnings generated by static code analysis tools. 3.2.1 presents the research questions answered by SpongeBugs's empirical evaluation, which is based on 15 open-source projects selected using the criteria we present in 3.2.2. We submitted the fix suggestions built by SpongeBugs as pull requests according to the protocol discussed in 3.2.3.

### 3.2.1 Research Questions

The empirical evaluation of SpongeBugs, whose results are described in 3.3, addresses the following research questions, which are based on the original motivation behind this chapter: automatically providing fix suggestions that helps improve the practical usability of SATs.

**RQ3.1.** How widely applicable is SpongeBugs?
The first research question looks into how many rule violations SpongeBugs can detect and build a fix suggestion for.

**RQ3.2.** Does SpongeBugs generate fixes that are acceptable?
The second research question evaluates SpongeBugs's effectiveness by looking into how many of its fix suggestions were accepted by project maintainers.

**RQ3.3.** How efficient is SpongeBugs?
The third research question evaluates SpongeBugs's scalability in terms of running time on large code bases.

### 3.2.2 Selecting Projects for the Evaluation

In order to evaluate SpongeBugs in a realistic context, we selected 15 well-established open-source Java projects that can be analyzed with SonarQube or SpotBugs. Three projects were natural choices: the SonarQube and SpotBugs projects are obviously relevant for applying their own tools; and the Eclipse IDE project is a long-standing Java project one of whose lead maintainers recently request help on Twitter[3] for fixing SonarQube issues. We selected the other twelve projects, following accepted best practices [40], among those that satisfy all of the following:

1. the project is registered with SonarCloud (a cloud service that can be used to run SonarQube on GitHub projects);
2. the project has at least 10 open issues related to violations of at least one of the 11 rules handled by SpongeBugs (see 3.1);
3. the project has at least one fixed issue;
4. the project has at least 10 contributors;
5. the project has commit activity in the last three months.

[3] https://twitter.com/vogella/status/1096088933144952832

| PROJECT | DOMAIN | STARS | FORKS | CONTRIBUTORS | LOC[a] |
|---|---|---|---|---|---|
| Eclipse IDE | IDE | 72 | 94 | 218 | 743 K |
| SonarQube | Tool | 3,700 | 1,045 | 91 | 500 K |
| SpotBugs | Tool | 1,324 | 204 | 80 | 280 K |
| atomix | Framework | 1,650 | 282 | 30 | 550 K |
| Ant-Media Server | Server | 682 | 878 | 16 | 43 K |
| cassandra-reaper | Tool | 278 | 125 | 48 | 88.5 K |
| database-rider | Test | 182 | 45 | 14 | 21 K |
| db-preservation-toolkit | Tool | 26 | 8 | 10 | 377 K |
| ddf | Framework | 95 | 170 | 131 | 2.5 M |
| DependencyCheck | Security | 1,697 | 464 | 117 | 182 K |
| keanu | Math | 136 | 31 | 22 | 145 K |
| matrix-android-sdk | Framework | 170 | 91 | 96 | 61 K |
| mssql-jdbc | Driver | 617 | 231 | 40 | 79 K |
| Payara | Server | 680 | 206 | 66 | 1.95 M |
| primefaces | Framework | 1,043 | 512 | 110 | 310 K |

[a] Non-comment non-blank lines of code calculated from Java source files using `cloc` (https://github.com/AlDanial/cloc)

Table 3.2: The 15 projects we selected for evaluating SpongeBugs. For each project, the table report its DOMAIN, and data from its GitHub repository: the number of STARS, FORKS, CONTRIBUTORS, and the size in lines of code. Since Eclipse's GitHub repository is a secondary mirror of the main repository, the corresponding data may not reflect the project's latest state.

### 3.2.3 Submitting Pull Requests With Fixes Made by SpongeBugs

After running SpongeBugs on the 15 projects we selected, we tried to submit the fix suggestions it generated as pull requests (PRs) in the project repositories. Following suggestions to increase patch acceptability [55], before submitting any pull requests we approached the maintainers of each project through online channels (GitHub, Slack, maintainers' lists, or email) asking whether pull requests were welcome. (The only exception was SonarQube itself, since we did not think it was necessary to check that they are OK with addressing issues raised by their own tool.) When the circumstances allowed so, we were more specific about the content of our potential PRs. For example, in the case of `mssql-jdbc`, we also asked: *"We noticed on the Coding Guidelines that new code should pass SonarQube rules. What about already committed code?"*, and mentioned that we found the project's dashboard on SonarCloud. However, we never mentioned that our fixes were generated automatically—but if the maintainers asked us whether a fix was automatic generated, we openly confirmed it. Interestingly, some developers also asked for a possible IDE integration of SpongeBugs as a plugin, which may indicate potential interest. We only submitted pull requests to the projects that replied with an answer that was not openly negative.

While the actual code patches in each pull request were generated automatically by SpongeBugs, we manually added information to present them in a way that was accessible by human developers—following good practices that facilitate code reviews [56]. We paid special attention to four aspects:

1. change description,
2. change scope,
3. composite changes, and
4. nature of the change.

To provide a good change description and clarify the scope of the change, we always mentioned which rule a patch is fixing—also providing a link to a textual description of the rule. In a few cases we wrote a more detailed description to better explain why the fix made sense, and how it followed recommendations issued by the project maintainers. For example, `mssql-jdbc` recommends to *"try to create small alike changes that are easy to review"*; we tried to follow this guideline in all projects. To keep our changes within a small scope, we separated fixes to violations of different rules into different pull requests; in case of fixes touching several different modules or files, we further partitioned them into separate pull requests per module or per file. This was straightforward thanks to the nature of the fix suggestions built by SpongeBugs: fixes are mostly independent, and one fix never spans multiple classes.

We consider a pull request *approved* when reviewers indicate so in the GitHub interface. Although the vast majority of the approved PRs were merged, two of them were approved but not merged yet at the time of writing. Since merging depends on other aspects of the development process[4] that are independent of the correctness of a fix, we do not distinguish between pull requests that were approved and those that were approved but not merged yet.

The reviewing process may approve a patch with or without modifications. For each patch generated by SpongeBugs and approved we record whether it was approved with or without modifications.

[4]One case is a pull request to Ant-Media-Server, which was approved but violates a project constraint that new code must be covered by tests.

| PROJECT | OK TO SUBMIT? | PULL REQUESTS SUBMITTED | APPROVED |
|---|---|---|---|
| `Eclipse IDE` | Positive | 9 | 9 |
| `SonarQube` | – | 1 | 1 |
| `SpotBugs` | Neutral | 1 | 1 |
| `atomix` | Positive | 2 | 2 |
| `Ant Media Server` | Positive | 3 | 3 |
| `database-rider` | Positive | 4 | 4 |
| `ddf` | Positive | 3 | 2 |
| `DependencyCheck` | Neutral | 1 | 1 |
| `keanu` | Positive | 3 | 0 |
| `mssql-jdbc` | Positive | 1 | 1 |
| `Payara` | Positive | 6 | 6 |
| `primefaces` | Positive | 4 | 4 |
| `cassandra-reaper` | No reply | – | – |
| `db-preservation-toolkit` | No reply | – | – |
| `matrix-android-sdk` | No reply | – | – |
|  | **Total:** | 38 | 34 |

Table 3.3: Responses to our inquiries about whether it is OK to submit a pull request to each project, and how many pull requests were eventually submitted and approved.

## 3.3 Empirical Evaluation of SpongeBugs: Results and Discussion

The results of our empirical evaluation of SpongeBugs answer the three research questions presented in 3.2.1.

### 3.3.1 RQ3.1: Applicability

To answer **RQ3.1** ("How widely applicable is SpongeBugs?"), we ran SonarQube on each project, counting the warnings triggering a violations of any of the 11 rules SpongeBugs handles. Then, we ran SpongeBugs and applied all its fix suggestions. Finally, we ran SonarQube again on the fixed project, counting how many warnings have disappeared. To speed up the analysis we configured SonarQube to only consider the set of rules that SpongeBugs supports (see Table 3.1); and we excluded test files (often located under *src/test/java*)—SonarQube ignores then by default anyway.

Table 3.4 shows the results of these experiments. Overall, SpongeBugs's fix suggestions remove the source of 81% of the all warnings violating the rules we considered in this research.

| PROJECT | B1 | B2 | C1 | C2 | C3 | C4 | C5 | C6 | C7 | C8 | C9 | TOTAL | FIXED % |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| `Eclipse IDE` | 44/5 | 13/13 | 214/199 | 4/4 | 11/8 | 18/3 | 189/176 | 15/11 | – | 159/97 | 102/32 | 769/548 | 71% |
| `SonarQube` | – | – | 104/81 | – | – | – | 7/7 | – | – | – | – | 111/88 | 79% |
| `SpotBugs` | 12/8 | 1/0 | 289/247 | 2/1 | 1/0 | 11/1 | 141/125 | – | – | 30/20 | – | 487/402 | 82% |
| `atomix` | 1/1 | – | 57/55 | – | – | – | 9/9 | – | – | 1/0 | 2/0 | 70/65 | 93% |
| `Ant Media Server` | – | – | 30/30 | 1/0 | 1/0 | 2/1 | 3/3 | 3/0 | – | 4/2 | 4/2 | 48/38 | 79% |
| `database-rider` | – | – | 5/5 | 5/5 | – | – | 2/2 | – | 1/1 | 1/1 | – | 14/14 | 100% |
| `ddf` | 1/0 | – | 104/97 | – | 1/0 | – | 88/86 | – | 1/1 | 45/33 | 8/1 | 248/218 | 88% |
| `DependencyCheck` | – | – | 61/51 | 10/4 | – | – | 3/3 | – | – | 4/2 | – | 78/60 | 77% |
| `keanu` | 1/1 | – | – | – | – | – | 4/4 | – | 12/11 | 5/3 | – | 22/19 | 86% |
| `mssql-jdbc` | 4/1 | – | 314/274 | 14/1 | – | 7/0 | 58/58 | 2/0 | – | 14/11 | – | 413/345 | 83% |
| `Payara` | 39/36 | – | 1,413/1,305 | 213/163 | 66/14 | 114/10 | 1,830/1,620 | 200/88 | 50/44 | 438/265 | 58/20 | 4,421/3,565 | 81% |
| `primefaces` | – | – | 336/286 | 2/0 | 9/6 | 3/3 | 336/329 | – | 1/1 | 1/0 | 4/1 | 692/626 | 90% |
| TOTAL | 102/52 | 14/13 | 2,927/2,630 | 251/178 | 89/28 | 155/18 | 2,670/2,422 | 220/99 | 65/58 | 702/434 | 178/56 | 7,373/5,988 | – |
| FIXED % | 51% | 93% | 90% | 71% | 31% | 12% | 91% | 45% | 89% | 62% | 31% | 81% | – |

Table 3.4: For each project and each rule checked by SonarQube, the table reports two numbers $x/y$: $x$ is the number of warnings violating that rule found by SonarQube on the original project; $y$ is the number of warnings that have disappeared after running SpongeBugs on the project and applying all its fix suggestions for the rule. The two rightmost columns summarize the data per project (TOTAL), and report the percentage of warnings that SpongeBugs successfully fixed (FIXED %). The two bottom rows summarize the data per rule in the same way.

These results justify our decision of focusing on a limited number of rules. In particular, the three rules (C3, C4, C9) with the lowest percentages of fixing are responsible for less than 6% of the triggered violations. In contrast, a small number of rules triggers the vast majority of violations, and SpongeBugs is extremely effective on these rules.

To elaborate, consider rule C9: (`Collections.EMPTY_LIST`, `EMPTY_MAP`, *and* `EMPTY_SET` *should not be used*). SpongeBugs only looks for return statements that violate this rule, since it is simpler to check whether the return type of a method is a generic collection, rather than to a variable declaration—which might depend on other local variables and constants. Listing 3.5 shows a fix for rule C9. Note that if the return type of the method were the raw type `Collection` (without type parameters), a violation would not be raised.

A widely applicable kind of suggestion are those for violations of rule C1 (*String literals should not be duplicated*), which SpongeBugs can successfully fix in 90% of the cases in our experiments. Generating automatically these suggestions is quite challenging. First, fixes to violations of rule C1 change multiple lines of code, and add a new constant. This requires to automatically come up with a descriptive name for the constant, based on the content of the string literal. The name must comply with Java's rules for identifiers (e.g., it cannot start with a digit). The name must also not clash with other constant and variable names that are in scope. SpongeBugs's fix suggestions can also detect whether there is already another string constant with the same value—reusing that instead of introducing a redundant new one.

```
public Collection<Binding> getSequencesFor(ParameterizedCommand command)
    ↪ {
    ArrayList<Binding> triggers = bindingsByCommand.get(command);
- return (Collection<Binding>) (triggers == null ?
    ↪ Collections.EMPTY_LIST : triggers.clone());
+ return (Collection<Binding>) (triggers == null ?
    ↪ Collections.emptyList() : triggers.clone());
}
```

Listing 3.5: Fix suggestion for a violation of rule C9 (`Collections.EMPTY_LIST`, `EMPTY_MAP`, *and* `EMPTY_SET` *should not be used*) in project Eclipse IDE.

```
public class AccordionPanelRenderer extends CoreRenderer {

+ private static final String FUNCTION_PANEL = "function(panel)";

@@ -130,13 +133,13 @@ public class AccordionPanelRenderer extends
    ↪ CoreRenderer {
         if (acco.isDynamic()) {
             wb.attr("dynamic", true).attr("cache", acco.isCache());
         }

         wb.attr("multiple", multiple, false)
- .callback("onTabChange", "function(panel)", acco.getOnTabChange())
- .callback("onTabShow", "function(panel)", acco.getOnTabShow())
- .callback("onTabClose", "function(panel)", acco.getOnTabClose());
+ .callback("onTabChange", FUNCTION_PANEL, acco.getOnTabChange())
+ .callback("onTabShow", FUNCTION_PANEL, acco.getOnTabShow())
+ .callback("onTabClose", FUNCTION_PANEL, acco.getOnTabClose());
```

Listing 3.6: Fix suggestion for a violation of rule C1 (*String literals should not be duplicated*) in project primefaces.

We also highlight that our approach is able to perform distinct transformations in the same file and statement. Listing 3.7 shows the combination of a fix for rule C1 (*String literals should not be duplicated*) applied in conjunction with a fix for rule C5 (*Strings literals should be placed on the left side when checking for equality*).

```
public class DataTableRenderer extends DataRenderer {

+ private static final String BOTTOM = "bottom";
```

```
- if (hasPaginator && !paginatorPosition.equalsIgnoreCase("bottom")) {
+ if (hasPaginator && !BOTTOM.equalsIgnoreCase(paginatorPosition)) {
```

Listing 3.7: Fix suggestion for a violation of rules C1 and C5 in the same file and statement found in project primefaces.

Another encouraging result is the negligible number of fix suggestions that failed to compile: only two among all those generated by SpongeBugs. We attribute this low number to our approach of refining SpongeBugs's implementation with the support of a curated and growing suite of examples to test against. We also note that one of the two fix suggestions that didn't compile is likely a false positive (reported by SonarQube). On line 6 of Listing 3.8, the String Literal. `"format"` is replaced by the constant `OUTPUT_FORMAT` which is only accessible within class `CliParser` using its qualified name `ARGUMENT.OUTPUT_FORMAT`. However, SonarQube's warning does not have this information, as it just says: "Use already-defined constant `OUTPUT_FORMAT` instead of duplicating its value here".

```
public final class CliParser {

- final Option outputFormat =
    ↪ Option.builder(ARGUMENT.OUTPUT_FORMAT_SHORT)
- .argName("format").hasArg().longOpt(ARGUMENT.OUTPUT_FORMAT)
+ final Option outputFormat =
    ↪ Option.builder(ARGUMENT.OUTPUT_FORMAT_SHORT)
+ .argName(OUTPUT_FORMAT).hasArg().longOpt(ARGUMENT.OUTPUT_FORMAT)

    public static class ARGUMENT {
        public static final String OUTPUT_FORMAT = "format";
    }
}
```

Listing 3.8: Example of a false-positive match of rule C1. Line 6 references constant `OUTPUT_FORMAT` which is not available as an unqualified name.

### 3.3.2 RQ3.2: Effectiveness and Acceptability

As discussed in Section sec:prs, we only submitted pull requests after informally contacting project maintainers asking to express their interest in receiving fix suggestions for warnings reported by SATs. As shown in 3.3, project maintainers were often quite welcoming of contributions with fixes for SATs violations, with 9 projects giving clearly positive answers to our informal inquiries.

For example an `Ant Media Server` maintainer replied “Absolutely, you’re welcome to contribute. Please make your pull requests”. A couple of projects were not as enthusiastic but still available, such as a maintainer of `DependencyCheck` who answered “I’ll be honest that I obviously haven’t spent a lot of time looking at SonarCloud since it was setup… That being said – PRs are always welcome”. Even those that indicated less interest in pull requests ended up accepting most fix suggestions. This indicates that projects and maintainers that do use SATs are also inclined to find valuable the fix suggestions in response to their warnings. We received no reply from 3 projects, and hence we did not submit any pull request to them (and we excluded them from the rest of the evaluation).

In order to answer **RQ3.2** (“Does SpongeBugs generate fixes that are acceptable?”), we submitted 38 pull requests containing 920 fixes for the 12 projects that responded our question on whether fixes were of interest for the project. We did not submit pull requests with all fix suggestions (more than 5,000) since we did not want to overwhelm the maintainers. Instead, we sampled broadly (randomly in each project) while trying to select a diverse collection of fixes.

Overall, 34 pull requests were accepted, some after discussion and with some modifications. 3.3 breaks down this data by project. The non-accepted pull requests were: 3 in project `keanu` that were ignored; and 1 in project `ddf` where maintainers argued that the fixes were mostly stylistic. In terms of fixes, 775 (84%) of all 920 submitted fixes were accepted; 740 (95%) of them were accepted without modifications.

How to turn these measures into a precision measure depends on what we consider a *correct* fix: one that removes the source of warnings (precision nearly 100%, as only two fix suggestions were not working), one that was accepted in a pull request (precision: 84%), or one what was accepted without modifications (precision: 740/920 = 80%). Similarly, measures of recall depend on what we consider the total amount of relevant fixes.

An aspect that we did not anticipate is how policies about code coverage of *newly added* code may impact whether fix suggestions are accepted. At first we assumed our transformations would not trigger test coverage differences. While this holds true for single-line changes, it may not be the case for fixes that introduce a new statement, such as those for rule C1 (*String literals should not be duplicated*), rule C3 (*Strings should not be concatenated using* `+` *in a loop*), and some cases of rule C7 (`entrySet()` *should be iterated when both key and value are needed*). For example, the patch shown in Listing 3.9 was not accepted because the 2 added lines were not covered by any test. One pull request to `Ant Media Server` which included 97 fixes in 20 files was not accepted due to insufficient test coverage of some added statements.

```
public class TokenServiceTest {

+ private static final String STREAMID = "streamId";

- token.setStreamId("streamId");
+ token.setStreamId(STREAMID);
```

Listing 3.9: The added lines were flagged as not covered by any existing tests.

Sometimes a fix's context affected whether it was readily accepted. In particular, developers tend to insist that changes be applied so that the overall stylistic consistency of the whole codebase is preserved. Let's see two examples of this.

Listing 3.10 fixes three violations of rule C2; a reviewer asked if line 3 should be modified as well to use a character `'*'` instead of the single-character string `"*"`:

> Do you think that for consistency (and maybe another slight performance enhancement) this line should be changed as well?

```
1  - if (pattern.indexOf("*") != 0 && pattern.indexOf("?") != 0 &&
       ↪ pattern.indexOf(".") != 0) {
2  + if (pattern.indexOf('*') != 0 && pattern.indexOf('?') != 0 &&
       ↪ pattern.indexOf('.') != 0) {
3        pattern = "*" + pattern;
4    }
```

Listing 3.10: Fix suggestion for a violation of rule C2 that introduces a stylistic inconsistency.

The pull request was accepted after a manual modification. Note that we do not count this as a modification to one of our pull requests, as the modification was in another line of code other than the one we fixed.

Commenting on the suggested fix in Listing 3.11, a reviewer asked:

> Although I got the idea and see the advantages on refactoring I think it makes the code less readable and in some cases look like the code lacks a standard, e.g one may ask why only this map entry is a constant?

```
+ private static final String CASE_SENSITIVE_TABLE_NAMES =
    ↪ "caseSensitiveTableNames";

putIfAbsent(properties, "batchedStatements", false);
putIfAbsent(properties, "qualifiedTableNames", false);
```

```
- putIfAbsent(properties, "caseSensitiveTableNames", false)
+ putIfAbsent(properties, CASE_SENSITIVE_TABLE_NAMES, false);
putIfAbsent(properties, "batchSize", 100);
putIfAbsent(properties, "fetchSize", 100);
putIfAbsent(properties, "allowEmptyFields", false);
```

Listing 3.11: Fix suggestion for a violation of rule C3 that introduces a stylistic inconsistency.

This fix was declined in project `database-rider`, even though similar ones were accepted in other projects (such as `Eclipse`) after the other string literals were extracted as constants in a similar way.

Sometimes reviewers disagree on their opinion about pull requests. For instance, we received four diverging reviews from four distinct reviewers about one pull request containing two fixes for violations of rule C3 in project `primefaces`. One developer argued for rejecting the change, others for accepting the change with modifications (with each reviewer suggesting a different modification), and others still arguing against other reviewers' opinions. These are interesting cases that may deserve further research, especially because several projects require at least two reviewers to agree to approve a change.

Sometimes fixing a violation is not enough [48]. Developers may not be completely satisfied with the fix we generate, and may request changes. In some initial experiments, we received several similar modification requests for fix suggestions to violations of rule C7 (`entrySet()` *should be iterated when both key and value are needed*); in the end, we changed the way the fix is generated to accommodate the requests. For example, the fix in Listing 3.12 received the following feedback from maintainers of `Eclipse`:

> For readability, please assign `entry.getKey()` to the `menuElement` variable

```
- for (MMenuElement menuElement : new
    ↪ HashSet<>(modelToContribution.keySet())) {
-   if (menuElement instanceof MDynamicMenuContribution) {
+ for (Entry<MMenuElement, IContributionItem> entry :
    ↪ modelToContribution.entrySet()) {
+   if (entry.getKey() instanceof MDynamicMenuContribution) {
```

Listing 3.12: Fix suggestion for a violation of rule C7 generated in a preliminary version of SpongeBugs.

We received practically the same feedback from developers of `Payara`, which prompted us to modify how SpongeBugs generates fix suggestions for violations of rule C7. Listing 3.13 shows the fixed suggestion with the new template. All fixes generated using this refined fix template, which we used in the experiments reported in this paper, were accepted by the developers without modifications.

```
- for (MMenuElement menuElement : new
   ↪ HashSet<>(modelToContribution.keySet())) {
+ for (Entry<MMenuElement, IContributionItem> entry :
   ↪ modelToContribution.entrySet()) {
+ MMenuElement menuElement = entry.getKey();
      if (menuElement instanceof MDynamicMenuContribution) {
```

Listing 3.13: Fix suggestion for a violation of rule C7 generated in the final version of SpongeBugs.

Overall, SpongeBugs's fix suggestions were often found of high enough quality perceived to be accepted—many times without modifications. At the same time, developers may evaluate the acceptability of a fix suggestions within a broader context, which includes information and conventions that are not directly available to SpongeBugs or any other static code analyzer. Whether to enforce some rules may also depend on a developer's individual preferences; for example one developer remarked that fixes for rule C5 (*Strings literals should be placed on the left side when checking for equality*) are "*style preferences*". The fact that many of such fix suggestions were still accepted is additional evidence that SpongeBugs's approach was generally successful.

### 3.3.3 RQ3.3: Performance

To answer **RQ3.3** ("How efficient is SpongeBugs?"), we report some runtime performance measures of SpongeBugs on the projects. All experiments ran on a Windows 10 laptop with an Intel-i7 processor and 16 GB of RAM. We used Rascal's native benchmark library[5] to measure how long our transformations take to run on the projects considered in Table 3.4. Table 3.5 show the performance outcomes. For each of the measurements in this section, we follow recommendations on measuring performance [57]: we restart the laptop after each measurement, to avoid any startup performance bias (i.e., classes already loaded); and also provide summary descriptive statistics on 5 repeated runs of SpongeBugs.

Project `mssql-jdbc` is an outlier due to its relatively low count of files analyzed with a long measured time. This is because its files tend to be large—multiple files with more than 1K lines. Larger files might imply more complex code, and therefore more complex ASTs, which consequently leads to more rule applications. To explore this hypothesis we

[5] http://tutor.rascal-mpl.org/Rascal/Rascal.html#/Rascal/Libraries/util/Benchmark/benchmark/benchmark.html

| PROJECT | FILES ANALYZED | RUNNING TIME MEAN | RUNNING TIME ST. DEV. |
|---|---:|---:|---:|
| Eclipse IDE | 5,282 | 63.9 m | 3.31 m |
| SonarQube | 3,876 | 25 m | 3.23 m |
| SpotBugs | 2,564 | 26.4 m | 2.05 m |
| Ant Media Server | 228 | 3.8 m | 0.15 m |
| atomix | 1,228 | 8.1 m | 0.23 m |
| database-rider | 109 | 0.8 m | 0.04 m |
| ddf | 2,316 | 29.7 m | 3.98 m |
| DependencyCheck | 245 | 4.9 m | 0.13 m |
| keanu | 445 | 2.6 m | 0.1 m |
| mssql-jdbc | 158 | 14.2 m | 0.27 m |
| Payara | 8,156 | 141.5 m | 5 m |
| primefaces | 1,080 | 11.8 m | 0.15 m |

Table 3.5: Descriptive statistics summarizing 5 repeated runs of SpongeBugs. Time is measured in minutes.

ran our transformations on a subset of these larger files. As seen in Table 3.6, five larger files are responsible for more than 5 minutes of running time. Additionally, file `dtv` takes, on average, almost 40 seconds (56%) longer than `SQLServerBulkCopy`; even though they have roughly the same size, file `dtv` has numerous class declarations and methods with more than 300 lines, containing multiple `switch`, `if`, and `try/catch` statements.

| FILE | LOC | RUNNING TIME MEAN | RUNNING TIME ST. DEV. |
|---|---:|---:|---:|
| SQLServerConnection | 4,428 | 116 s | 2.4 s |
| SQLServerResultSet | 3,858 | 99 s | 3.8 s |
| dtv | 2,823 | 106 s | 3.1 s |
| SQLServerBulkCopy | 2,529 | 68 s | 5 s |
| SQLServerPreparedStatement | 2,285 | 64 s | 4.8 s |

Table 3.6: Descriptive statistics summarizing 5 repeated runs of SpongeBugs on the 5 largest files in projects mssql-jdbc. Time is measured in seconds.

Generating some fix suggestions takes longer than others. We investigated this aspect more closely in SpotBugs, which is the largest project among those we analyzed. SpotBugs includes more than a thousand files containing multiple test cases for the rules it implements. Excluding test files in `src/test/java` does not work for SpotBugs, which puts tests in another location, thus greatly increasing the amount of code that SpongeBugs analyzes.

We also observed that SpongeBugs takes considerably longer to run on rules B1, B2/C6, and C1. The main reason is that step 1 in these rules raises several false positives, which are then filtered out by the more computationally expensive step 2 (see 3.1.2). For example, step 1's filtering for rule B1 (*Strings and boxed types should be compared using* `equals()`), shown in Listing 3.14, is not very restrictive. One can imagine that several files have a reference to a `String` (covered by `hasWrapper()`) and also use `==` or `!=` for comparison operators. Contrast this to step 1's filtering for rule C9 (`Collections.EMPTY_LIST` ... *should not be used*), shown in Listing 3.15, which is much more restrictive; as a result SpongeBugs runs in under 5 seconds for rule C9.

```
return hasWrapper(javaFileContent) &&
    ↪ hasEqualityOperator(javaFileContent);
```

Listing 3.14: Violation textual pattern in the implementation of rule B1

```
return findFirst(javaFileContent, "Collections.EMPTY") != -1;
```

Listing 3.15: Violation textual pattern in the implementation of rule C9

Overall, we found that SpongeBugs's approach to fix warnings of SATs is scalable on projects of realistic size. SpongeBugs could be reimplemented to run much faster if it directly used the output of static code analysis tools, which indicate precise locations of violations. While we preferred to make SpongeBugs's implementation self contained to decouple from the details of each specific SAT, we plan to explore other optimizations in future work.

### 3.3.4 Discussion

In this section we summarize findings we collected based on the feedback given by reviews of our pull requests.

**Some fixes are accepted without modifications.** Some fixes are uniformly accepted without modifications. For example those for rule C2 (*String function use should be optimized for single characters*), which bring performance benefits and only involve minor modifications (as shown in Listing 3.16: change string to character).

```
- int otherPos = myStr.lastIndexOf("r");
+ int otherPos = myStr.lastIndexOf('r');
```

Listing 3.16: Example of a fix for a violation of rule C2.

**SAT adherence is stricter in new code.** Some projects require SAT compliance only on new pull requests. This means that previously committed code represent accepted technical debt. For instance, `mssql-jdbc`'s contribution rules state that "New developed code should pass SonarQube rules". A `SpotBugs` maintainer also said "I personally don't check it so seriously. I use SonarCloud to prevent from adding more problems in new PR". Some use SonarCloud not only for identifying violations, but for test coverage checks.

**Fixing violations as a contribution to open source.** Almost all the responses to our questions about submitting fixes were welcoming—along the lines of *help is always welcome.* Since one does not need a deep understanding of a project domain to fix several SATs' rules, and the corresponding fixes are generally easy to review, submitting patches to fix violations is an approachable way of contributing to open source development.

**Fixing violations induce other clean-code activities.** Sometimes developers requested modifications that were not the target of our fixes. While our transformations strictly resolved the issue raised by static analysis, developers were aware of the code as a whole and requested modifications to preserve and improve code quality.

**Fixing issues promotes discussion.** While some fixes were accepted "as is", others required substantial discussion. We already mentioned a pull request for `primefaces` that was intensely debated by four maintainers. A maintainer even drilled down on some Java Virtual Machine details that were relevant to the same discussion. Developers are much more inclined to give feedback when it is about code they write and maintain.

## 3.4 Conclusions

In this chapter we introduced a new approach and a tool (SpongeBugs) that finds and repairs violations from static code analysis tools such as SonarQube, FindBugs, and SpotBugs. We designed SpongeBugs to deal with relevant violations, those that are frequently fixed in both private and open-source projects. We assessed SpongeBugs by running it on 12 popular open source projects, and submitting a large portion (total of 920) of the fixes it generated as pull requests in the projects. Overall, project maintainers accepted 775 (84%) of those fixes—most of them without any modifications. We also assessed SpongeBugs's performance, showing that it scales to large projects (under 10 minutes on projects as large as 550 KLOC). These results suggest that SpongeBugs, which uses source-to-source transformations based on standard templates for fixing warnings from bug finding tools, can be an effective approach to fix warnings issued by static code analysis tools, contributing to increasing the usability of these tools and, in turn, the overall quality of a software system.

For *future work*, we envision using SpongeBugs to prevent violations to static code analysis rules from happening in the first place. One way to achieve this is by making its functionality available as an IDE plugin, which would help developers in real time. Another approach is to integrate SpongeBugs as a tool in a continuous integration toolchain. We plan to pursue these directions in future work.

# Chapter 4

# Conclusions

This research thesis have discussed whether or not the usage of ASATs (Automated Static Analysis Tools) is useful and detailed what types of violations developers tend to fix. To this end, we first surveyed developers and then mined data from four SonarQube instances to explore common practices to fix issues. The survey results reinforce the need for automatic fixes and we also discussed the results of a quantitative analysis that show that 20% of the rules are responsible for 80% of the fixes. These findings might help practitioners to better configure their tools, and also has the potential to help tool developers to better select rules to fix.

To build on the reports of several studies, which points out that developers' need for tools that aid them in fixing ASATs' issues, we also explored whether providing automatic fixes is feasible. We designed and implemented SpongeBugs, which leverages program transformations to automatically fix 11 rules from SonarQube and SpotBugs. Although, previous works (e.g., [5, 27]) have already explored automatic fixes for ASATs' issues, they focus on behavioral issues that might yield a bug. We argue that our approach is complementary to existing ones. While SpongeBugs mainly fixes violations that do not affect the program behavior, it represents an effective approach as project maintainers accepted 84% of the fixes it generated, and its performance scaled well in our experiments (under 10 minutes on projects as large as 550 KLOC). Moreover, SpongeBugs was designed to fix rules that are frequently fixed in both open-source and private projects. We also highlighted that our fixes were accepted by widely used projects in the Java ecosystem, such as the Eclipse IDE.

Altogether, we can conclude that even though developers face several barriers when using and adopting ASATs, they still find them useful. By providing automatic fixes for commonly violated rules, this research not only aids developers on fixing issues they find valuable, thus, reducing the overall effort of using an ASAT, but also addresses a well-reported need of developers: tools that provide fixes.

## 4.1 Related Work

Beller et al. [6] performed a large-scale evaluation on how nine different ASATs are used. They investigate how ASATs are configured by analyzing 168,214 OSS projects, for Java and other three popular programming languages, and reported that the default configurations of most tools fits the needs of the majority of projects–custom rules are used in less than 5% of the cases. We find diverging results in Chapter 2 as we argue that developers should choose more carefully the rules ASATs check, and a big portion of EF and TCU fixed rules are related to custom rules. Regarding ASAT usage in a CI context, Rausch et al. [58] performed an in-depth analysis of the build failures of 14 projects, and found that 10 of these projects present a history of build failures related to violations reported by ASATs. Zampetti et al. [31] analyze 20 Java OSS projects and report that build breakages are mainly related to adherence to coding guidelines, while build failures originated by potential bugs or vulnerabilities do not occur frequently. Our results confirm that a large portion of fixed issues are related to code smells, which cover coding standards and other aspects.

Vassalo et al. [18] conducted a study on 119 OSS projects, mined from SonarCloud (cloud service based on SonarQube), and concluded that developers check code quality only at the end of a sprint, contrary to CI principles. In this study we find that developers tend to fix issues from 216.60 days to 299.12 on average, after they had been reported. However, we find in ASF's projects a central tendency to fix issues in 6.67 days after the report. Kim and Ernest [43] observed warnings by three distinct ASATs, finding that no more than 9% are removed during fix changes. They suggest that issues' prioritization given by ASATs are ineffective. We question ASAT's ineffectiveness on prioritizing issues, largely due to developers mainly fixing Major issues, which are supposed to highly impact developers' productivity.

Recent studies focused on the kinds of rule violations developers are more likely to fix. Liu et al. [27] compared a large number of *fixed* and *not fixed* FindBugs rule violations across revisions of 730 Java projects. They characterized the categories of violations that are often fixed, and reported several situations in which the violations are systematically ignored. They also concluded that developers disregard most of FindBugs violations as not being severe enough to be fixed during development process. Digkas et al. [21] performed a similar analysis for SonarQube rules, revealing that a small number of all rules account for the majority of the programmer-written fixes.

Several researchers have developed techniques that propose fix suggestions for rules of the popular FindBugs in different ways: interactively, with the user exploring different alternative fixes for the same warning [48]; and automatically, by systematically generating fixes by mining patterns of programmer-written fixes [5, 27]. These studies focus on *behavioral* bugs e.g., the ones collected in Defects4J [59]—a curated collection of 395 real bugs of open-source Java projects. In contrast, SpongeBugs mainly targets rules that characterize so-called *code smells*—patterns that may be indicative of poor programming practices, and mainly encompass design and stylistic aspects. We focus on these because they are simpler to characterize and fix "syntactically" but are also the categories of warnings that developers using SATs are more likely to address and fix [50, 27, 21]. This focus helps SpongeBugs achieve high precision and scalability, as well as be practically useful.

The work closest to ours is probably Liu et al.'s study [5], which presents the AVATAR automatic program repair system. AVATAR recommends code changes based on the output of SAT tools. In its experimental evaluation, AVATAR generated correct fixes for 34 bugs among those of the Defects4J benchmark. This suggests that responding to warnings of SATs can be an effective way to fix some behavioral "semantic" bugs. However, AVATAR's and SpongeBugs' scopes are mostly complementary, since our approach focuses on "syntactic" design flaws that often admit simple yet effective fixes.

Other approaches learn transformations from examples, using sets of bug fixes [60], bug reports [61], or source-code editing patterns [62]. We directly implemented SpongeBugs' transformations based on our expertise and the standard recommendations for fixing warnings from static analysis tools. Even though SpongeBugs cannot learn new fixing rules, this remains an interesting direction for further improving its capabilities.

Behavioral bugs are also the primary focus of techniques for "automated program repair", a research area that has grown considerably in the last decade. The most popular approaches to automated program repair are mainly driven by dynamic analysis (i.e., tests) [63, 64] and targets generic bugs. In contrast, SpongeBugs' approach is based on static code-transformation techniques, which makes it of much more limited scope but also more easily and widely applicable.

## 4.2 Threats to Validity and Limitations

### 4.2.1 Identifying what kind of violations developers tend to fix

Identifying if an issue is fixed intentionally is a commonly reported threat among studies that analyze ASATs [21, 27, 43]. Common mitigation strategies involve mining source code management repositories to look for patterns in commit messages[31, 27, 43], or to find references for bug reports identifiers, such as #4223, that are hosted on issue management platforms [43]. Although in our study we did not mine commits, we minimize the threat on our private dataset, as we know for sure, by means of our collaborations, that PF and TCU developers use SonarQube. Regarding OSS projects, we have indication that SonarQube is used, as for ASF we had respondents in our survey, and for EF, SonarQube is mandatory for projects that aim to achieve a higher maturity assessment. We believe that the projects we studied in this chapter are well suited for our analyses. Our results might be partially generalized to companies and OSS projects. We study 246 projects, 155 from large Brazilian government institutions, and 91 OSS projects from two well-known open-source foundations. However, since Eclipse Foundation and Apache Software Foundation are well matured foundations, with well defined standards and practices, they may not represent the general OSS community. Our choice of SonarQube as the targeted ASAT for this chapter might not properly contextualize general usage of ASATs. We believe that this threat is minimized as SonarQube is used by more than 85,000 organizations, and also encompasses rules from other ASATs, such as FindBugs, and PMD [18]. Also, our study restricts its analysis only on Java projects. Even though Java projects are among the most prevalent on the organizations we investigated, practices on fixing issues might differ for other languages.

Our survey design was targeted for project leaders and more experienced developers in regards to ASAT usage. This decision might have diminished the participation of either experienced developers that do not use ASATs in the first place, or less experienced developers in general.

Our technical decisions might also introduce some threats to internal validity. That is, since we mined data from different versions of SonarQube instances, with different APIs, our approach for data extraction and filtering might have errors. However, we manually verified parts of our data, and in some cases we verified both our data and findings with collaborators from TCU and PF. Another major threat is the reliance on measuring issues' open and creations dates only from the data extracted from SonarQube. It is possible that an issue fix, for example, may have happened in a different moment than the tool was run, and thus the date reported by SonarQube might not reflect a precise date/time on when the issue was fixed. As we observed in PF, this limitation may be minimized by *nightly builds* (tools are executed automatically at the end of each day).

### 4.2.2 Automatically providing fix suggestions for ASATs' violations

Some of SpongeBugs's transformations may violate a project's stylistic guidelines [27]. As an example, project `primefaces` uses a rule[1] about the order of variable declarations within a class that requires that private constants (`private static final`) be defined

[1]http://checkstyle.sourceforge.net/apidocs/com/puppycrawl/tools/checkstyle/checks/coding/DeclarationOrderCheck.html

after public constants. SpongeBugs's fixes for rule C1 (*String literals should not be duplicated*) may violate this stylistic rule, since constants are added as the first declaration in the class. Another example of stylistic rule that SpongeBugs may violate is one about empty lines between statements[2]. Overall, these limitations appear minor, and it should not be difficult to tweak SpongeBugs's implementation so that it fixes comply with additional stylistic rules.

Static code analysis tools are a natural target for fix suggestion generation, as one can automatically check whether a transformation removes the source of violation by rerunning the static analyzer [64]. In the case of SonarCloud, which runs in the cloud, the appeal of automatically generating fixes is even greater, as any technique can be easily scaled to benefit a huge numbers of users. We checked the applicability of SpongeBugs on hundreds of different examples, but there remain cases where our approach fails to generate a suitable fix suggestions. There are two reasons when this happens:

1. *Implementation limitations.* One current limitation of SpongeBugs is that its code analysis is restricted to a single file at a time, so it cannot generate fixes that depend on information in other files. Another limitation is that SpongeBugs does not not analyze methods' return types.

2. *Restricted fix templates.* While manually designed templates can be effective, the effort to implement them can be prohibitive [5]. With this in mind, we deliberately avoided implementing templates that were too hard to implement relative to how often they would have been useful.

SpongeBugs's current implementation does not rely on the output of SATs. This introduces some occasional inconsistencies. as well as cases where SpongeBugs cannot process a violation reported by a SAT. An example, discussed above, is rule C9: SpongeBugs only considers violation of the rule that involve a return statement. These limitations of SpongeBugs are not fundamental, but reflect trade-offs between efficiency of its implementation and generality of the technique it implements. We only ran SpongeBugs on projects that normally used SonarQube or SpotBugs. Even though SpongeBugs is likely to be useful also on general projects, we leave a more extensive experimental evaluation to future work.

## 4.3 Future Work

Based on the results of the research presented in this dissertation, we highlight the following directions for future research as open questions:

[2]http://checkstyle.sourceforge.net/config_whitespace.html#EmptyLineSeparator

**How do organization policies or practices impact on the practice of fixing issues?**

We found a low resolution of static analysis violations (8.76%). Some organization policies or practices might influence this rate, such as fix all violations classified as bugs, or do not fix violations with lower severity. Another reported practice is to fix violations only close to a release date. Identifying these practices and how they impact the fixing of issues might help not only researchers but practitioners too, to increase the practical use of static analysis tools.

**What are the reasons for developers not fixing issues?** Organization policies or practices are not the only reasons that issues are not fixed.

Personal and team preferences might also influence the practice of fixing issues. Although the study of Johnson et al. [1] already identified several reasons for the lack of usage of ASATs (e.g., a high number of identified issues and false-positives), a complementary and interesting question remains: Why developers that already use ASATs do not fix issues identified by them? One way to approach this is to revisit our survey done with practitioners that use ASATs and ask more open questions on when and why the fix, or do not fix, issues.

**Are fixes from ASATs generally accepted by projects that do not use them?**

We submitted 38 pull-requests to 12 different well established open-source projects. In total, those pull requests contained 920, and maintainers accepted 775 of them. A commonality between all these projects is that they already used SonarQube. We did not explore whether projects that do not explicitly use SonarQube or any other ASAT would welcome fixes for offending code. Even though some rules might indicate personal developer opinion, a portion of them bring unarguable benefits, such as increased performance. We conjecture that some fixes, especially those that involve minor modifications, might be generally accepted.

**Can SpongeBugs be integrated with ASATs' outputs (instead of identifying a violation by itself)?**

SpongeBugs currently reimplements the violation identification done by tools such as SonarQube and SpotBugs. This approach brings the benefit of making SpongeBugs self-contained, as in this way no ASAT is needed to be set to run against a project, moreover, this setup is not always practical or possible. However, SpongeBugs could also take in consideration ASATs' violations reports. This would make it possible to increase the precision of how many issues SpongeBugs can fix, also, would also drastically increase the runtime performance.

**Can SpongeBugs prevent violations from happening in the first place?**

When interacting with developers and submitting pull requests to fix ASATs' issues, we identified that static analysis tools adherence is stricter in new code. SpongeBugs could be used to prevent the violations from happening (i.e., before code is committed to the version control system). One way to explore this is to integrate SpongeBugs as an IDE plugin.

# References

[1] Johnson, Brittany, 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*, pages 672–681. IEEE Press, 2013. 1, 3, 6, 11, 55

[2] Ayewah, Nathaniel and William Pugh: *The google FindBugs fixit.* In *Proceedings of the 19th international symposium on Software testing and analysis.* ACM Press, 2010. https://doi.org/10.1145/1831708.1831738. 1

[3] Ayewah, Nathaniel, David Hovemeyer, J. David Morgenthaler, John Penix, and William Pugh: *Using static analysis to find bugs.* IEEE Software, 25(5):22–29, sep 2008. https://doi.org/10.1109/ms.2008.130. 1, 29

[4] Habib, Andrew and Michael Pradel: *How many of all bugs do we find? a study of static bug detectors.* In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ASE 2018, pages 317–328, New York, NY, USA, 2018. ACM, ISBN 978-1-4503-5937-5. http://doi.acm.org/10.1145/3238147.3238213. 1, 29, 30

[5] Liu, K., A. Koyuncu, D. Kim, and T. F. Bissyandè: *Avatar: Fixing semantic bugs with fix patterns of static analysis violations.* In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 1–12, Feb 2019. 1, 8, 29, 50, 51, 52, 54

[6] Beller, M., R. Bholanath, S. McIntosh, and A. Zaidman: *Analyzing the state of static analysis: A large-scale evaluation in open source software.* In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, volume 1, pages 470–481, March 2016. 1, 6, 12, 29, 51

[7] Fowler, Martin, Kent Beck, John Brant, William Opdyke, and Don Roberts: *Refactoring: improving the design of existing code.* Addison-Wesley Professional, 1999. 1

[8] Vakilian, Mohsen, Nicholas Chen, Stas Negara, Balaji Ambresh Rajkumar, Brian P Bailey, and Ralph E Johnson: *Use, disuse, and misuse of automated refactorings.* In *Proceedings of the 34th International Conference on Software Engineering*, pages 233–243. IEEE Press, 2012. 1

[9] *Iso/iec/ieee international standard - systems and software engineering–vocabulary.* ISO/IEC/IEEE 24765:2017(E), pages 1–541, Aug 2017. 2

[10] Pressman, Roger S: *Software engineering: a practitioner's approach.* Palgrave Macmillan, 2005. 2

[11] Ghezzi, Carlo, Mehdi Jazayeri, and Dino Mandrioli: *Fundamentals of Software Engineering.* Prentice Hall PTR, Upper Saddle River, NJ, USA, 2nd edition, 2002, ISBN 0133056996. 2

[12] Meirelles, Paulo Roberto Miranda: *Monitoramento de métricas de código-fonte em projetos de software livre.* PhD thesis, Universidade de São Paulo. 2

[13] Dallal, Jehad Al: *Object-oriented class maintainability prediction using internal quality attributes.* Information and Software Technology, 55(11):2028–2048, nov 2013. https://doi.org/10.1016/j.infsof.2013.07.005. 2

[14] Santos, Danilo, Antônio Resende, Paulo Afonso Junior, and Heitor Costa: *Attributes and metrics of internal quality that impact the external quality of object-oriented software: A systematic literature review.* In *Computing Conference (CLEI), 2016 XLII Latin American*, pages 1–12. IEEE, 2016. 2

[15] ISO/IEC: *Iso/iec 25010 system and software quality models.* Technical report, 2010. 2

[16] Emanuelsson, Pär and Ulf Nilsson: *A comparative study of industrial static analysis tools.* Electronic Notes in Theoretical Computer Science, 217:5–21, jul 2008. https://doi.org/10.1016/j.entcs.2008.06.039. 3

[17] Hovemeyer, David and William Pugh: *Finding bugs is easy.* ACM SIGPLAN Notices, 39(12):92, dec 2004. https://doi.org/10.1145/1052883.1052895. 3

[18] Vassallo, Carmine, Fabio Palomba, Alberto Bacchelli, and Harald C. Gall: *Continuous code quality: are we (really) doing that?* In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering - ASE 2018.* ACM Press, 2018. https://doi.org/10.1145/3238147.3240729. 3, 11, 51, 53

[19] SonarSource S.A.: *Sonarqube*, 2019. https://www.sonarqube.org, [accessed 18-January-2019]. 3

[20] Vassallo, Carmine, 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).* IEEE, mar 2018. https://doi.org/10.1109/saner.2018.8330195. 3, 11

[21] Digkas, G., M. Lungu, P. Avgeriou, A. Chatzigeorgiou, and A. Ampatzoglou: *How do developers fix issues and pay back technical debt in the apache ecosystem?* In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 153–163, March 2018. 3, 6, 12, 13, 14, 24, 31, 51, 52

[22] Visser, Eelco: *A survey of rewriting strategies in program transformation systems.* Electronic Notes in Theoretical Computer Science, 57:109–143, dec 2001. https://doi.org/10.1016/s1571-0661(04)00270-1. 4

[23] Cooper, Keith and Linda Torczon: *Engineering a compiler*. Elsevier, 2011. 4

[24] Klint, Paul, Tijs van der Storm, and Jurgen Vinju: *RASCAL: A domain specific language for source code analysis and manipulation*. In *2009 Ninth IEEE International Working Conference on Source Code Analysis and Manipulation*. IEEE, 2009. https://doi.org/10.1109/scam.2009.28. 5

[25] Wohlin, Claes, Per Runeson, Martin Hst, Magnus C. Ohlsson, Bjrn Regnell, and Anders Wessln: *Experimentation in Software Engineering*. Springer Publishing Company, Incorporated, 2012, ISBN 3642290434, 9783642290435. 5

[26] Wang, Junjie, Song Wang, and Qing Wang: *Is there a "golden" feature set for static warning identification?* In *Proceedings of the 12th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement - ESEM'18*. ACM Press, 2018. https://doi.org/10.1145/3239235.3239523. 6, 11

[27] Liu, K., D. Kim, T. F. Bissyande, S. Yoo, and Y. Le Traon: *Mining fix patterns for findbugs violations*. IEEE Transactions on Software Engineering, pages 1–1, 2018, ISSN 0098-5589. 6, 12, 13, 24, 31, 50, 51, 52, 53

[28] Allen, I Elaine and Christopher A Seaman: *Likert scales and data analyses*. Quality progress, 40(7):64, 2007. 7, 14

[29] Izquierdo, Javier Luis Cánovas and Jordi Cabot: *The role of foundations in open source projects*. In *Proceedings of the 40th International Conference on Software Engineering Software Engineering in Society - ICSE-SEIS'18*. ACM Press, 2018. https://doi.org/10.1145/3183428.3183438. 7, 12, 14

[30] Saha, Ripon K., Yingjun Lyu, Wing Lam, Hiroaki Yoshida, and Mukul R. Prasad: *Bugs.jar*. In *Proceedings of the 15th International Conference on Mining Software Repositories - MSR'18*. ACM Press, 2018. https://doi.org/10.1145/3196398.3196473. 11

[31] Zampetti, Fiorella, Simone Scalabrino, Rocco Oliveto, Gerardo Canfora, and Massimiliano Di Penta: *How open source projects use static code analysis tools in continuous integration pipelines*. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. IEEE, may 2017. https://doi.org/10.1109/msr.2017.2. 11, 51, 52

[32] Lin, Yu, Semih Okur, and Danny Dig: *Study and refactoring of android asynchronous programming (T)*. In *30th IEEE/ACM International Conference on Automated Software Engineering, ASE 2015, Lincoln, NE, USA, November 9-13, 2015*, pages 224–235, 2015. 11

[33] Decan, Alexandre, Tom Mens, and Eleni Constantinou: *On the impact of security vulnerabilities in the npm package dependency network*. In *Proceedings of the 15th International Conference on Mining Software Repositories, MSR 2018, Gothenburg, Sweden, May 28-29, 2018*, pages 181–191, 2018. 11

[34] Pinto, Gustavo, Anthony Canino, Fernando Castor, Guoqing (Harry) Xu, and Yu David Liu: *Understanding and overcoming parallelism bottlenecks in forkjoin applications*. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, ASE 2017, Urbana, IL, USA, October 30 - November 03, 2017*, pages 765–775, 2017. 11

[35] Aniche, Mauricio Finavaro, Gabriele Bavota, Christoph Treude, Marco Aurélio Gerosa, and Arie van Deursen: *Code smells for model-view-controller architectures*. Empirical Software Engineering, 23(4):2121–2157, 2018. 11

[36] Bavota, Gabriele, Gerardo Canfora, Massimiliano Di Penta, Rocco Oliveto, and Sebastiano Panichella: *The evolution of project inter-dependencies in a software ecosystem: The case of apache*. In *2013 IEEE International Conference on Software Maintenance*. IEEE, sep 2013. https://doi.org/10.1109/icsm.2013.39. 14

[37] Hollander, M. and D.A. Wolfe: *Nonparametric Statistical Methods*. Wiley Series in Probability and Statistics. Wiley, 1999, ISBN 9780471190455. 15

[38] Dunn, Olive Jean: *Multiple comparisons among means*. Journal of the American statistical association, 56(293):52–64, 1961. 15

[39] Mukaka, Mavuto M: *A guide to appropriate use of correlation coefficient in medical research*. Malawi Medical Journal, 24(3):69–71, 2012. 15

[40] Kalliamvakou, Eirini, Georgios Gousios, Kelly Blincoe, Leif Singer, Daniel M. German, and Daniela Damian: *An in-depth study of the promises and perils of mining GitHub*. Empirical Software Engineering, 21(5):2035–2071, sep 2015. https://doi.org/10.1007/s10664-015-9393-5. 15, 35

[41] Panjer, L. D.: *Predicting eclipse bug lifetimes*. In *Fourth International Workshop on Mining Software Repositories (MSR'07:ICSE Workshops 2007)*, pages 29–29, May 2007. 18

[42] Giger, Emanuel, Martin Pinzger, and Harald Gall: *Predicting the fix time of bugs*. In *Proceedings of the 2Nd International Workshop on Recommendation Systems for Software Engineering*, RSSE '10, pages 52–56, New York, NY, USA, 2010. ACM, ISBN 978-1-60558-974-9. http://doi.acm.org/10.1145/1808920.1808933. 19

[43] Kim, Sunghun and Michael D. Ernst: *Which warnings should i fix first?* In *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering - ESEC-FSE'07*. ACM Press, 2007. https://doi.org/10.1145/1287624.1287633. 24, 51, 52

[44] Fenton, N. E. and N. Ohlsson: *Quantitative analysis of faults and failures in a complex software system*. IEEE Transactions on Software Engineering, 26(8):797–814, Aug 2000, ISSN 0098-5589. 26

[45] Runeson, P. and C. Andersson: *A replicated quantitative analysis of fault distributions in complex software systems*. IEEE Transactions on Software Engineering, 33:273–286, 2007. 26

[46] Walkinshaw, Neil and Leandro L. Minku: *Are 20% of files responsible for 80% of defects?* In *Proceedings of the 12th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, ESEM 2018, Oulu, Finland, October 11-12, 2018*, pages 2:1–2:10. ACM, 2018. 26

[47] Johnson, Brittany, 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. http://dl.acm.org/citation.cfm?id=2486788.2486877. 29

[48] Barik, T., Y. Song, B. Johnson, and E. Murphy-Hill: *From quick fixes to slow fixes: Reimagining static analysis resolutions to enable design space exploration.* In *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 211–221, Oct 2016. 29, 44, 51

[49] Tómasdóttir, K. F., M. Aniche, and A. van Deursen: *Why and how javascript developers use linters.* In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 578–589, Oct 2017. 29

[50] Marcilio, Diego, Rodrigo Bonifácio, Eduardo Monteiro, Edna Canedo, Welder Luz, and Gustavo Pinto: *Are static analysis violations really fixed?: A closer look at realistic usage of sonarqube.* In *Proceedings of the 27th International Conference on Program Comprehension*, ICPC '19, pages 209–219, Piscataway, NJ, USA, 2019. IEEE Press. https://doi.org/10.1109/ICPC.2019.00040. 29, 31, 52

[51] Johnson, Brittany, Yoonki Song, Emerson R. Murphy-Hill, and Robert W. Bowdidge: *Why don't software developers use static analysis tools to find bugs?* In *35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013*, pages 672–681, 2013. 31

[52] Brito, Aline, Laerte Xavier, Andre Hora, and Marco Tulio Valente: *Why and how Java developers break APIs.* In *25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 255–265, 2018. 31

[53] Klint, Paul, Tijs Van Der Storm, and Jurgen Vinju: *Rascal: A domain specific language for source code analysis and manipulation.* In *2009 Ninth IEEE International Working Conference on Source Code Analysis and Manipulation*, pages 168–177. IEEE, 2009. 33

[54] Dantas, Reno, Antonio Carvalho, Diego Marcilio, Luisa Fantin, Uriel Silva, Walter Lucas, and Rodrigo Bonifácio: *Reconciling the past and the present: An empirical study on the application of source code transformations to automatically rejuvenate java programs.* In Oliveto, Rocco, Massimiliano Di Penta, and David C. Shepherd (editors): *25th International Conference on Software Analysis, Evolution and Reengineering, SANER 2018, Campobasso, Italy, March 20-23, 2018*, pages 497–501. IEEE Computer Society, 2018. https://doi.org/10.1109/SANER.2018.8330247. 33

[55] Tao, Y., D. Han, and S. Kim: *Writing acceptable patches: An empirical study of open source project patches.* In *2014 IEEE International Conference on Software Maintenance and Evolution*, pages 271–280, Sep. 2014. 36

[56] Ram, Achyudh, Anand Ashok Sawant, Marco Castelluccio, and Alberto Bacchelli: *What makes a code change easier to review: an empirical investigation on code change reviewability.* In *Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2018, Lake Buena Vista, FL, USA, November 04-09, 2018*, pages 201–212, 2018. https://doi.org/10.1145/3236024.3236080. 37

[57] Georges, Andy, Dries Buytaert, and Lieven Eeckhout: *Statistically rigorous java performance evaluation.* In *Proceedings of the 22Nd Annual ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications*, OOPSLA '07, pages 57–76, New York, NY, USA, 2007. ACM, ISBN 978-1-59593-786-5. http://doi.acm.org/10.1145/1297027.1297033. 45

[58] Rausch, Thomas, Waldemar Hummer, Philipp Leitner, and Stefan Schulte: *An empirical analysis of build failures in the continuous integration workflows of java-based open-source software.* In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. IEEE, may 2017. https://doi.org/10.1109/msr.2017.54. 51

[59] Just, René, Darioush Jalali, and Michael D. Ernst: *Defects4j: A database of existing faults to enable controlled testing studies for java programs.* In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ISSTA 2014, pages 437–440, New York, NY, USA, 2014. ACM, ISBN 978-1-4503-2645-2. http://doi.acm.org/10.1145/2610384.2628055. 52

[60] Nguyen, Tung Thanh, Hoan Anh Nguyen, Nam H. Pham, Jafar Al-Kofahi, and Tien N. Nguyen: *Recurring bug fixes in object-oriented programs.* In *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, pages 315–324, New York, NY, USA, 2010. ACM, ISBN 978-1-60558-719-6. http://doi.acm.org/10.1145/1806799.1806847. 52

[61] Liu, C., J. Yang, L. Tan, and M. Hafiz: *R2fix: Automatically generating bug fixes from bug reports.* In *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*, pages 282–291, March 2013. 52

[62] Rolim, Reudismam, Gustavo Soares, Rohit Gheyi, and Loris D'Antoni: *Learning quick fixes from code repositories.* CoRR, abs/1803.03806, 2018. http://arxiv.org/abs/1803.03806. 52

[63] Gazzola, Luca, Daniela Micucci, and Leonardo Mariani: *Automatic software repair: A survey.* IEEE Trans. Software Eng., 45(1):34–67, 2019. https://doi.org/10.1109/TSE.2017.2755013. 52

[64] Monperrus, Martin: *Automatic software repair: A bibliography.* ACM Comput. Surv., 51(1):17:1–17:24, January 2018, ISSN 0360-0300. http://doi.acm.org/10.1145/3105906. 52, 54