Università
della
Svizzera
italiana

Practical Automated Program Analysis for Improving Java Software

Repairing Static Analysis Violations and Analyzing Exception Behavior

presented by

Diego Marcilio

under the supervision of

Prof. Carlo Alberto Furia

November 2023

Dissertation Committee

Dissertation accepted on 23 November 2023

Research Advisor
Prof. Carlo Alberto Furia

Ph.D. Program Co-Director	Ph.D. Program Co-Director
Prof. Walter Binder	**Prof. Stefan Wolf**

I certify that except where due acknowledgement has been given, the work presented in this thesis is that of the author alone; the work has not been submitted previously, in whole or in part, to qualify for any other academic award; and the content of the thesis is the result of work which has been carried out since the official commencement date of the approved research program.

Diego Marcilio
Lugano, 23 November 2023

Abstract

Finding and fixing bugs are among the most time-consuming activities of the software development process. This thesis presents work that increases the level of automation in finding and fixing bugs in Java software: by automatically repairing static analysis warnings and by analyzing exception behavior. In both directions, we aim to provide actionable feedback to developers and to demonstrate practical applicability.

Developers widely use static analysis tools (SATs) to identify bugs early in the development process. However, using SATs comes with challenges, such as too many reported warnings, false positives, and limitations in detecting issues that relate to libraries and external project dependencies.

To improve the usability of SATs when they report a high number of violations, we propose to automatically address some of the violations by synthesizing source-code fixes. We designed a technique, SpongeBugs, to produce fixes for violations of simple, widely used rules detected by popular static analyzers (SonarQube and SpotBugs). Our technique can often generate fixes quickly and that are similar to those developers would write. In an experimental evaluation, maintainers of popular Java open-source projects accepted 87% of 946 fixes generated automatically by SpongeBugs.

To widen the scope of static analysis to issues involving external libraries, we focus on exception behavior, which is notoriously often poorly documented, associated with antipatterns, and a frequent source of software failures. We first examined how Java developers test exception behavior and identified the most frequently tested exceptions. Building on these insights, we introduced the WIT technique, which automatically extracts precise exception preconditions in Java methods. We demonstrated several practical applications of using WIT on realistic programs. First, we used WIT's extracted preconditions to add to and improve the Javadoc documentation of popular Apache Commons projects: Lang, IO, and Text. We then repurposed WIT so that it could analyze client code to detect calls that violate the exception preconditions of library calls. We applied this approach to 1,523 open-source Java projects in 21 widely used open-source Java libraries, including the Java Development Kit (JDK); we found 4,115 cases of calls to library methods that may result in an exception. To our knowledge, this kind of analysis of exceptions that originate in calls to external libraries is beyond the capabilities of most commercial static analyzers.

Overall, our contributions were designed so that they can work with limited requirements on the analyzed codebases. This emphasizes providing practical tools and actionable and reliable feedback, which can help developers be more productive when finding and fixing bugs.

Contents

Figures

Tables

Part I

Prologue

1

Introduction

Debugging—finding and fixing bugs—is a recurring and time-consuming task for developers: it can take from 50% [70] to 80% [106] of a software product's budget. Finding bugs by itself is a complex task, and programmers employ several tactics for identifying them [106]. Some widespread development practices involve writing different kinds of tests (e.g., unit and integration) and using static analysis tools (SATs) [125]. By working *statically* on the source or byte code of a project, these tools search for patterns that may indicate problems—bugs, vulnerabilities, or failures to follow formatting conventions. SATs are used on large code bases both in commercial and open-source settings.

Despite providing several benefits, using SATs also comes with disadvantages [89]. A key problem is the high number of false positive warnings—those that do not correspond to actual mistakes. It is likely for a developer to find thousands of warnings when executing a static analysis tool for the first time in a project; it is unlikely that a developer wants to act on all of them. Another related issue is that understanding a problem indicated by a warning and coming up with a suitable fix is often nontrivial. Finally, the set of warnings SATs can detect is limited; one under-represented case is when the code interacts with third-party libraries—an ubiquitous practice in modern software development. A library method may throw an exception, for example, to signal one of its arguments should not be `null`. SATs do not generally report exceptions that originate in an external library.

To alleviate these problems, our research explores two related directions: i) automatic repair of static analysis warnings; and ii) analysis of exception behavior. By automatically repairing static analysis warnings, we can aid developers make the most out of tools that are already commonly integrated into developers workflows. By analyzing exception behavior, we can address a common source of failures when using libraries. In our work, we often trade-off generality in favor of *practicality*; focusing on these specific topics allows our techniques to *automatically* yield *actionable* feedback while executing on large projects with low requirements (e.g., directly on source-code, without requiring to build a whole project with all its dependencies).

We adopt the paradigm of Automated Program Repair (APR) to automatically fix SATs warnings. APR's ultimate promise is to automate the detection and fixing of bugs, ultimately improving developers' productivity. It generally works by identifying incorrect behavior in a program against a given specification and then generating fixes to the source code that conform to that specification. Most APR techniques rely on tests to identify buggy behavior and to validate generated fixes [70, 142]. More concretely, a test-based technique identifies bugs that are revealed by some failing tests and generates changes to the program that make all tests pass. Recent studies [92, 115] show that the over-reliance on tests severely hinders APR's applicability in real world scenarios. Test-based APR approaches are mostly evaluated on benchmarks of bugs, which curate reproducible bugs from popular open-source projects, where each bug is tied to one or more failing tests that exposes it. However, these bug-exposing tests are almost entirely *future tests* [115]: they were written after discovering a bug (e.g., following a user bug report). In Defects4J [91], a widely used Java benchmark for APR, 381 (96%) of all tests are future tests [115]. Future tests are not representative of real-world scenarios [92]. The over-reliance on such tests to pin-point a bug's location, fostered by the high usage of benchmarks of bugs, may partially explain why APR techniques are mostly not applicable to realistic development scenarios. Indeed, Winter et al. [197] found that only 17 (6%) APR papers out of 264 engage with developers in their evaluations.[1]

Repairing Static Analysis Violations. To achieve greater APR applicability, we focus on fixing SATs warnings, which allows us to circumvent the use of tests in the APR pipeline. A warning pinpoints the bug's location, therefore there is no need for a failing test. If a SAT does not report a warning after a fix is applied, the fix can be considered valid. With this in mind, we developed SpongeBugs, a technique that can produce fixes for 11 Java violations of rules frequently detected by widely used Java static analyzers (SonarQube and SpotBugs). The technique scales to real projects and generates fixes similar to what developers would write, as suggested by our experimental evaluation: maintainers of popular Java open-source projects accepted 87% of 946 fixes generated by SpongeBugs.

In programming languages like Java, a method's implementation may throw an exception to signal that a call violates its precondition. Ideally, a method's exception behavior should be described in the method's documentation and thoroughly tested. However, a method's documentation can become incomplete or inconsistent with its implementation [150, 210], and, compounding the problem, a project's test suite may insufficiently exercise exceptional behavior [126]. Unsurprisingly, exception behavior is a frequent source of failures, and is often implicated in anti-patterns (e.g., empty catch blocks, no informative messages). As APIs typically throw exceptions to signal invalid preconditions [126], uncaught exceptions are often symptoms of API misuses—a common cause of bugs [4]. To our knowledge, static analyzers typically cannot generally report exceptions that may be thrown by an external library.

Analyzing Exception Behavior. To address these exception related difficulties, we increasingly explored Java exception behavior. First, we investigated how Java developers test exceptions with JUnit. We found that unchecked exceptions such as `IllegalArgumentException` and `NullPointerException` figured among the most tested exceptions. These same exceptions are most frequently implicated in API misuses [196] and Android app bugs [37], and are among those often insufficiently documented [97, 173, 207]. Given that these frequently thrown exceptions may be used to signal precondition violations, we developed WIT, a tech-

[1]Our work on SpongeBugs (Chapter 3) is among those few 6% that engage with developers.

nique to automatically extract exception preconditions in a precise manner. WIT performs a lightweight static analysis of Java classes for checking which exception program paths are feasible by symbolically executing them. WIT only needs the source code of the classes under analysis, and it employs several heuristics to remain practical. We confirmed WIT' extracted preconditions to be 100% precise in our evaluation on 46 large real world Java open-source projects. Automatically extracted precise preconditions can directly help developers writing documentation (and tests). We demonstrated several practical applications of using WIT on realistic programs; we used WIT's extracted preconditions to add to and improve the Javadoc documentation of highly popular Apache Commons projects (i.e., Lang, IO, and Text). These projects figure among the most mature, popular, documented, and tested Java libraries [150]. Apache maintainers accepted and merged 7 pull requests containing 170 additions and improvements to the projects' Javadoc. Detecting calls that may throw exceptions is yet another practical application of having precise exception preconditions. We repurposed WIT to store libraries' exceptions preconditions in a database so that it can run on client code to detect calls that violate these preconditions. We applied this approach in 1,523 open-source Java projects to automatically detect calls that can potentially throw exceptions in 21 widely used open-source Java libraries, including the Java Development Kit (JDK). We found 4,115 cases of calls to library methods that may result in an exception. To our knowledge, exceptions that originate in calls to external libraries are not generally covered by Java static analyzers.

1.1 Thesis Statement

We formulate our thesis statement as follows:

> *Maximizing practicality for source code tools in non local environments involves minimizing assumptions and requirements, aiming for high precision, and targeting amenable problems—even if they seem uncomplicated.*

We can see "minimizing assumptions and requirements" under a *practicality* lens. Ideally, we want to lower the barriers for adopting a tool and to reduce the constraints for its applicability. For instance, asking developers to add seemingly simple Java annotations (e.g., `@NonNull`) may deter practitioners to use compile-time tools [52, 69]. And a tool that does not scale to realistic-size codebases is unlikely to be of usage outside academic settings.

Designing for high *precision* is a well-accepted guideline to increase practitioners engagement with a tool. Christakis and Bird [34] suggest a precision no lower than 75–80%: "Developers care much more about too many false positives than about too many false negatives". The authors also add that "high false positive rates lead to disuse".

Distinguishing between the contexts developers use tools is essential for maximizing practicality and minimizing assumptions. Vassallo et al. [190] identify three contexts on how developers engage with static analysis tools: local programming, continuous integration, and code review. Local programming happens when developers write code in integrated development editors (IDEs) and text editors. The other two contexts occur *after* local programming; they may happen after a long time since the code was written, and may involve a developer different than the original author. We posit that, in a local context, developers may tolerate warnings and fixes with *some* uncertainty degree of correctness. Barik et al. [13] support our conjecture by proposing an idea of “slow fixes”: developers, while in the IDE, may explore different solutions for a warning. On the other hand, fixes and warnings in a non local context should strive to be as correct as possible, even if that incurs targeting unambiguous and seemingly simple problems. Another source of motivation for our work is SapFix [130], the first APR approach deployed into production on industrial software of millions of lines of code, whose “repairs aim to tackle the most prevalent (yet arguably also the most simple) bugs, fixable by small patches, comparatively easily checked by the final human gate-keeper.” [130].

In all, we aim to minimize assumptions and requirements for source code tools in order to maximize practicality. By targeting amenable problems with high precision (i.e., low false positive rate), our tools outputs can generally be trusted without requiring manual validation.

1.2 Research Contributions

The contributions of our research can be grouped in two high-level categories: i) using static analysis for Automated Program Repair; and ii) targeting exception behavior for static analysis.

1.2.1 Automated Program Repair of Static Analysis Warnings

Static analysis tools are widely used by many software companies and consortia to identify bugs and for software quality assurance. As static analysis works statically on the source code (i.e., without running the program or tests), using it for bug detection and fixing does not depend on whether tests are available. For this goal, we developed SpongeBugs, a technique that can produce fixes for warnings of Java rules detected by static analyzers SonarQube and SpotBugs. The technique is fast and generates fixes similar to what developers would write, as suggested by our experimental evaluation: maintainers of popular Java open-source projects accepted 87% of 946 fixes generated by SpongeBugs. This work [129] was published as follows:

> **Automatically Generating Fix Suggestions in Response to Static Code Analysis Warnings**
> **Diego Marcilio**, Carlo A. Furia, Rodrigo Bonifácio, Gustavo Pinto. In *Journal of Systems and Software (JSS 2020)*, Volume 168, 2020

C# is a popular object-oriented language which shares similarities with Java (e.g., similar syntax, top types, garbage collection) but also interesting differences (e.g., no checked exceptions in C#, no operator overloading in Java).[2]{2} We replicated [155] our work of automatically fixing Java static analysis warnings for the C# language:

[2]Throughout the text, we use plain numbers for standard footnotes (shown in the same page, as usual). For URL references (e.g., GitHub source code and pull requests), we use superscript numeric marks, and list them at the end of the thesis after the bibliographic references. These superscripts are in blue between curly braces{1} so that they can be easily distinguished from regular footnotes.

> **Static Analysis Warnings and Automatic Fixing: A Replication for C# Projects**
> Martin Oddermatt, **Diego Marcilio**, Carlo A. Furia. In *Proceedings of the 29th IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER 2022) – Reproducibility Studies and Negative Results (RENE) Track*, pp. 805–816, 2022

Automatically integrating the generated fixes into a codebase is a desirable feature that increases the level of automation. However, this integration has its own challenges; we report our experience on implementing and deploying an automated bot that submits pull requests with fixes for SATs warnings [27]:

> **C-3PR: A Bot for Fixing Static Analysis Violations via Pull Requests**
> Antonio Carvalho, Welder Pinheiro Luz, **Diego Marcilio**, Rodrigo Bonifácio, Gustavo Pinto, Edna Dias Canedo. In *Proceedings of the 27th IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER 2020) – Technical Research Track*, pp. 161–171, 2020

1.2.2 Actionable Static Analysis of Exception Behavior

Exception behavior in Java is frequently linked to poor programming practices; it's often undocumented, tied to anti-patterns, and implicated in bugs. We incrementally investigated and analyzed Java exception behavior to ultimately suggest improvements to client code when its interaction with an external library could potentially throw an exception.

We gained valuable insights of frequently thrown exceptions by looking into how Java developers test exceptions [126]:

> **How Java Programmers Test Exceptional Behavior**
> **Diego Marcilio**, Carlo A. Furia. In *Proceedings of the IEEE/ACM 18th International Conference on Mining Software Repositories (MSR 2021) – Research Technical Track*, pp. 207–218, 2021

We then developed a static analysis technique that automatically extracts exception precondition by analyzing the Java source code of public methods. The work resulted in the following publication, awarded with an IEEE TCSE Distinguished Paper Award [127]:

> **What Is Thrown? Precise Automatic Extraction of Exception Preconditions in Java Methods**
> **Diego Marcilio**, Carlo A. Furia. In *Proceedings of the IEEE International Conference on Software Maintenance and Evolution (ICSME 2022) – Research Technical Track*, pp. 340–351, 2022

In a follow-up work, we extended the technique to support modular analysis: the reuse of preconditions previously extracted when analyzing different projects. Modular analysis allowed us to not only extract more preconditions, but to also extract preconditions occurring in a project's dependencies, which are often not immediately identifiable. The results of this work has been accepted by the Spring Journal of Empirical Software Engineering as an extension of the previous publication:

> **Lightweight Precise Automatic Extraction of Exception Preconditions in Java Methods**
> **Diego Marcilio**, Carlo A. Furia. In *Empirical Software Engineering (EMSE 2023)*, To appear, 2023

Building on the modular analysis of the previous work, we analyzed client code using several popular libraries, in order to find possible violations of the libraries' exception preconditions. Exception behavior in clients often naturally suggests improvements to the documentation, tests, runtime checks, and annotations of the clients. The work has been accepted in the “New Ideas and Emerging Results Track” (NIER) of the 2023 International Conference on Software Maintenance and Evolution (ICSME):

> **Towards Code Improvements Suggestions from Client Exception Analysis**
> **Diego Marcilio**, Carlo A. Furia. In *Accepted at 30th IEEE International Conference on Software Analysis, Evolution and Reengineering (ICSME 2023) – New Ideas and Emerging Results Track (NIER) Track*, To appear, 2023

1.3 Outline

This dissertation is structured in the following chapters:

Chapter 2 presents an overview of the state of the art, including the most relevant literature to frame the two main topics covered in this thesis: i) using static analysis for Automated Program Repair; and ii) targeting exception behavior for static analysis.

Chapter 3 describes our work on using static analysis for automated program repair. The chapter describes the SpongeBugs work, our technique to fix static analysis warnings.

Chapter 4 presents our empirical investigation into how Java programmers test exception behavior. Looking into exception tests can unveil commonly thrown exceptions and provide additional sources of information and documentation on how these exceptions are triggered.

Chapter 5 describes our technique to automatically infer exception preconditions from Java methods and constructors. Exception preconditions can be used to add or improve documentation and outline precise conditions when an exception is thrown.

Chapter 6 discusses experiments applying the idea of suggesting code improvements for client code when an exception may be thrown in calls of library APIs. This work builds on the technique of Chapter 5, as we first analyze the exception preconditions from popular Java libraries to detect possible thrown exceptions in client code using these libraries.

Chapter 7 concludes this dissertation by summarizing our work and indicating open research directions based on the results we achieved.

2

State of the Art

2.1 Introduction

In this chapter, we overview the literature related to the two topics covered in this thesis: (i) repairing static analysis warnings; and (ii) exception behavior analysis. First, in Section 2.2, we introduce Automated Program Repair (APR) and summarize its main limitations in terms of practical applicability. We mostly focus on studies targeting the Java language, which dominates the current APR literature [56, 70, 116, 142]. We discuss some practical APR approaches and the trade-offs they make to achieve some practicality. Then, in Section 2.3, we review the literature on static analysis tools and link them to APR. Finally, we discuss the research on Java exception behavior in Section 2.4. We focus our research, and state of the art discussion, on the Java programming language as it has been one of the most popular languages during recent years.{3},{4} Moreover, Java dominates the current APR literature [56, 70, 116, 142].

2.2 Automated Program Repair

2.2.1 General-Purpose Automated Program Repair

The research area of Automated Program Repair (APR) has been gaining increased attention in the last decade [142]. APR aims to generate fixes at the source code level to remove a bug in a way that conforms to a specification [70]. As described by Gazzola et al. [70], the APR process can be typically decomposed into three steps. First, *fault localization* identifies locations that may be responsible for a bug. Then *fix generation* modifies the locations of the source code indicated by the localization step. Several techniques can drive the generation of fixes, ranging from predefined templates [99], random mutation operators [76], constraint solving [135], to machine learning [10, 16]. The third and final step is *validation*, which checks if the generated fix actually repairs the program.

Gazzola et al. [70] classify APR approaches into two categories when considering the strategies for generating fixes and validating them. *Correct-by-construction* approaches [135, 136, 153, 200] formally encode the expected correct behavior, and produce fixes guaranteed to remove the bug. *Generate-and-validate* approaches [55, 76, 99] generate fixes by exploring a space of potential fixes for a given bug and then validate the correctness of the potential fixes. Test-based approaches are the most common "flavor" of the generate-and-validate category; they leverage test suite executions for both the localization and the verification steps [113, 115, 116].

In practice, tests guide most of the APR approaches [56]. Fault localization relies on *failing* test cases to identify potential buggy code locations. Spectrum-based fault localization—the most popular fault-localization technique used in APR [70, 113]—leverages the execution traces of test cases to compute the fault likelihood of a given code location. Its underlying idea is that code locations executed by many failing test cases and few passing test cases are likely to be faulty, and vice versa [70]. On the other hand, the validation step of APR uses the test cases to validate generated patches. Since formal specifications are mostly unavailable [14, 106], while tests are generally more available, test suites are used as (partial) specifications [116]. A given patch passes the verification step if all available tests are passing [70].

The over-reliance on test suites—with the requirement of failing test cases—may significantly hinder the practical usability of APR techniques. The empirical evaluations of a large portion of APR techniques rely on *future tests*: failing bug-triggering tests that are only added or updated *after* the bug has been detected and manually fixed [115]. Liu et al. [115] found that state-of-the-art APR techniques only fix at most 6 (1.5%) bugs out of 395 when dropping future tests from the test suites used in their evaluations. Such a low performance is explained by Ginelli et al. [73]: "it *is hard or even impossible* to repair a fault without selecting a good location for the fix". These future tests are often present in the benchmarks of bugs used in the evaluation of APR techniques.

Benchmarks of bugs facilitate leveraging tests for APR activities; they systematically collect reproducible bugs, where each bug is a pair of a buggy program version and a failing test that exposes it [121]. Benchmarks provide reliable ways of ensuring the generality of research results and enable the comparison of different techniques on a common ground [121]. However, the danger of over-fitting a benchmark is real. Indeed, recent studies [56, 115, 116] highlight that Java APR approaches are biased towards Defects4J [91], a widely used[1] Java benchmark of bugs that contains bugs from popular open-source projects. Researchers found [56, 115] that many tools not only fix 10-30% more bugs when using Defects4J instead of other benchmarks but are also hard-coded in non-trivial ways to work only with Defects4J.

Liu et al. [115] provide further criticism of this Defects4J bias in APR research. The Defects4J authors dedicated significant effort to curate the collection to facilitate fault localization and validation. This effort includes adding future tests, rewriting previous test cases to expose buggy locations, and inserting assertions in program source code. There is a staggering total of 381 (96%) future test cases in the 395 bugs in Defects4J. Two other benchmarks, Bugs.jar [171] and Bears [121], share similar rates with 94% and 92% of future

[1]Duriex et al. [56] analyzed 24 Java test-based APR tools and found that 22 of those were evaluated on Defects4J.

tests, respectively.

Soremekun et al. [177] took a deep dive into fault localization techniques and how "in the lab" evaluation assumptions fare in the real world. The authors identified 3 assumptions that researchers often make and investigated how they affect fault localization evaluations. In summary, the assumptions are:

- **Perfect Bug Understanding**: occur when assuming that locating and examining the first faulty statement (out of several) is sufficient to explain and fix the bug;
- **Fix Location**: occur when assuming that a fix location coincide with a fault location (i.e., the root cause and the fix location may be different);
- **Single Fault Location**: occur when assuming that a single contiguous location is sufficient for locating a bug (i.e., a bug location may be spread e.g., lines 6 and 10).

They performed a controlled experiment with 18 automated fault localization, 2 APR techniques (GenProg and Angelix), and 76 professional developers. Although they studied C programs, the authors' findings are likely to generalize to fault localization in other languages as the techniques follow the same idea. In an analysis of 63 papers from 7 high-ranked Software Engineering venues,[2] the authors found that the Fix Location assumption (the most prevalent one) is present in 76% of the evaluations. According to the authors, this assumption is also present in Defects4J [91]. When assessing developers' thoughts about the assumptions, they write a concerning statement: "fix location assumption is the least sound and most severe debugging assumption". In all, 60% of the developers believe the assumptions are unrealistic in practice, as they lead to "imprecise, wrong or inadequate bug diagnoses".

Kabadi et al. [92] follow up on Liu et al.'s work [115] to build a dataset of bugs *without* any future tests. The authors collected 102 bugs by manually inspecting continuous integration (CI) failures from 40 large programs. The idea of considering CI failures is to leverage regression errors: existing tests, that were previously passing, fail after the latest source code modification. The comparison between bugs exclusively exposed by regression tests against bugs exposed by future tests shed light on how the latter hinders APR's practicality. The number of bug-exposing tests from the CI failures are more than six times larger than the Defects4J ones and they are three times harder to localize (i.e., Defects4J bug-exposing tests have a much higher chance to lead to the location of a bug). Kabadi et al. mention that the higher the number of bug-exposing tests, the larger the search space for repair will be, as larger parts of code may be deemed suspicious. Less localizable means that the identified regression tests have a smaller chance to lead to the root causes of the bugs; there is a higher number of methods that need to be repaired among the ones explicitly called by bug-exposing test cases. The authors aptly summarize the conceptual problem—beyond the technical fault-localization one—by highlighting the following:

> [...] using future test cases may not be able to accurately estimate how well APR would work in practice. Since developers already know where and how to fix

[2]The authors used the CORE Conference Ranking[5] to select four conferences: ICSE, FSE, ASE, and MSR. The three chosen journals were: TSE, EMSE, and TOSEM.

the bugs, tests created in this case may encapsulate knowledge gained *only after* bugs are fixed.

2.2.2 Large-Language Models and Automated Program Repair

As of 2023, large pre-trained language models (LLMs) are in the limelight. General-purpose models (e.g., GPT3 [23]) have been fine-tuned to work with code (e.g., Codex). Due to these capabilities, researchers have been exploring how LLMs perform on several tasks, including APR. Despite showing a lot of promise, LLMs adoption also bring challenges [61] regarding bias (e.g., training mostly done in the English language), legal compliance (e.g., copyright and licensing), and security vulnerabilities (e.g., suggesting previously learned vulnerable code). This section aim to discuss both LLMs promises and limitations and how they fit in APR.

Two 2022 studies evaluated how Codex [33] fares in APR. In the first study, Prenner et al. [164] setup an experiment on the QuixBugs benchmark [110], a dataset of 40 Java and Python algorithm implementations with one-line bugs. The authors feed a buggy function to Codex and ask the model (i.e., prompt-engineering) to generate a complete non-buggy version. The authors found that Codex performed similarly to state-of-the-art learning based APR. Moreover, Codex had better results with Python programs than Java; it fixed 50% more bugs in Python. Kolak et al. [104] uses three versions of open-source models, alongside with Codex, to evaluate the models also on QuixBugs. Instead, the authors ask the model to fix a single line at a time, as opposed to a full function in [164]. In all, Prenner et al. found that Codex fixed around 45-57% of the bugs in Python and 35–45% in Java; Kolak et al. found better results when the model is applied to a single line at a time; Codex fixed 88.9% of the bugs in Python and 50% in Java. Note that, in both studies, fault localization was not taken in consideration; the authors fed the exact bug location to the model.

More recently, two 2023 studies conducted a more thorough evaluation of LLMs in APR. Fan et al. [65] evaluated Codex on 113 Leetcode{6} Java problems. Leetcode is an online platform with thousands of programming tasks ranging from easy to hard difficulty; it is commonly used for software engineering interview preparation. The authors experimented with three types of fault localization to provide to Codex for generating patches: i) just telling Codex “that a bug exists”; ii) providing candidate fix line numbers (by using Ochiai [1] as fault localization); iii) using the statement itself as part of the instruction. The number of correct patches was as following: i) 15 patches; ii) 11 patches; iii) 16 patches. Fan et al. provide insightful directions for software development, leveraging the interplay of APR and LLMs. They suggest a test-driven development (TDD) workflow where developers specify requirements in natural language along with a few test cases. LLMs generate the program and APR techniques could be used to fix small mistakes in the program by validating it via test cases. Xia et al. [199] investigate how 9 LLMs perform on 5 different repair datasets in Java, Python, and C. The authors experimented with different settings, including in particular: a) complete function generation and b) single line generation. Similar to the previous study, for a), the prompt does not provide a bug location. Instead for b), the prompt provides a bug location. Codex was the best performing model. When analyzing rates of syntactic and semantic errors of the generated patches, Codex had close to 40% errors when performing complete function generation and close to 80% on single line generation. In the case of Defects4J, the authors found that Codex could fix 32 more bugs than 20 Java existing APR techniques; 12 traditional APR tools (e.g., template-based) and 8 learning-based tools. For the comparison, the tools were given perfect fault localization: the ground-truth fix location is known. The authors also share useful findings and insights: "for real-world software systems, it is still more cost-effective to first use traditional fault localization techniques to pinpoint the precise bug locations and then leverage LLMs for more targeted patch generation". The authors looked into whether the models were generating fixes that may have been present in the training data (i.e., leakage). They found that for 93 Defects4J bugs, all LLMs combined generated at least one correct patch that is different than the original developer patch.

Bertrand Meyer, in his June 2023 article entitled "AI Does Not Help Programmers", experimented with ChatGPT 4 and concluded that: "AI in its modern form, does not generate correct programs. [...] These programs look correct but have no guarantee of correctness.". Meyer recognizes that ChatGPT provides value in assisting one writing code from scratch, but it does not work for a professional programmer hoping for an effective pair-programmer. The author envisions that for Generative AI for programming to work in serious professional programming, "it will have to spark a wonderful renaissance of studies and tools in formal specification and verification".

In all, LLMs currently shine on assisting developers on "spending more time on the fun part of their job—solving hard problems" by reducing the time spent on boilerplate code.{7} Indeed, two works [19, 159] by Microsoft and GitHub researchers highlight Copilot's impact on productivity. The authors devised programming tasks (e.g., implement a "send email" feature, implement an HTTP server in JavaScript) measuring completion time; they additionally surveyed the developers after the task. The researchers saw that participants spent more time reviewing code than writing the code itself, and they could complete tasks 55.8% faster than developers that were not using Copilot. In August 2023, Zhong and Wang [209] analyzed whether API misuses are present when recent LLMs (GPT-3.5, GPT-4, Llama-2, and Vicuna-1.5) are prompted to answer 1,208 StackOverflow Java questions studied in a previous work by Zhang et al. [206]. The questions pertain to JDK and Android APIs, including string processing, data structures, and IO. They found that among all the answers that contain executable code, 57-70% of the code snippets contain some API misuse. An interesting direction may be to give extra prompts to the model (e.g., ask it to circumvent the misuse) or provide different and/or equivalent prompts. On the latter, Mastropaolo et al. [132], found that Copilot can generate different code recommendations given semantically equivalent but different Javadoc descriptions. Some correct recommendations could only be achieved using a semantically equivalent description. Another promising direction is what Ciniselli et al. [35] call "adaptive recommendations": recommended code that is automatically adapted to the code under development; a LLM should e.g., reuse identifiers and preserve the coding style. As of October 2023, Amazon CodeWhisperer provides a customization that allows its model to analyze internal codebases.{8},{9} The end result is that the model "understand the intent, determines which internal and public APIs are best suited to the task, and generates code recommendations".

2.2.3 Practical Automated Program Repair Approaches

Most of APR evaluation has an over-reliance on benchmarks of bugs. As summarized by an August 2023 blog post from Uber,{10} "Current automatic program repair focuses on standard benchmarks, and neglects evaluation on real production code". In this section, we summarize actionable approaches, which may work with minimal requirements and assumptions, and have been evaluated in real-world scenarios.

Industry Reports and Deployment

Perhaps the most practical APR approaches are those studied, devised, and deployed in industry. An early experience report from 2018 by Naitou et al. [146] describes the authors experience on applying GenProg [76] and NOPOL [200] in a software development company. For the 22 bugs considered by the authors, none of them had failing bug-exposing tests; despite the authors mentioning that the software was well tested. Naitou et al. [146] state the developers were aware of test coverage tools, and aimed at maximizing coverage. Nonetheless, the existent tests did not expose any of the considered bugs. This observation—while anecdotal—may explain why most of the approaches we discuss in this "Practical APR" chapter *do not* rely on traditional fault localization, enabled by bug-exposing failing tests.

Sapienz [124] is a search-based test generator of test cases that trigger crashes in Android apps. SapFix [130] is an end-to-end repair approach that relies on Sapienz. It was successfully deployed at Meta (formerly Facebook) targeting apps with millions of lines of codes and hundred millions of users. When Sapienz identifies a crash, it feeds it to SapFix for a repair attempt. SapFix uses GetAFix [10] to generate template fixes learned from previous successful fixes. In order for a fix to be valid, it must pass all tests (including those generated by Sapienz) and also clear any violations detected by the Infer static analyzer. The interplay between Infer and Sapienz is very effective: bugs reported by both tools have a 98% fix rate. The authors explain a likely reason for the high fix rate: "developers have a localisation of both the likely root causing fault (from Infer) and a consequent failure (from Sapienz)". In other words, Infer seems to be doing the heavy-lifting of the *fault* localization, whether Sapienz exposes the *failure*. As evidence supporting our goal of practicality, which includes fixing seemingly simple problems, the authors summarize the approach: "Our repairs aim to tackle the most prevalent (yet arguably also the most simple) bugs, fixable by small patches, comparatively easily checked...".

Also deployed at Meta, GetAFix [10][3] fixes instances of commons bugs by learning from past fixes. It target fixes for warnings detected by Infer and Error Prone, which includes null dereferences, incorrect API calls and misuses of Java constructs. Our tool SpongeBugs (presented in Chapter 3) and GetAFix share two common rules. The static analyzers are used not only to pinpoint a bug, but also to validate the fix: a fix is valid if it removes the warning.

[3] GetAFix is integrated into SapFix but described in a different publication.

GetAFix synthesizes different suggestions for each fix; its top-5 suggestions contained fixes for 526 of the 1,268 collected bug fixes.

Bloomberg is another large company that shared its experience with APR [102, 198]. The authors are very pragmatic about what they expect from APR: "Academics are often more interested in the novel aspects of research and finding the next new solution; industry, on the other hand, is more interested in the value they get from the technique, irrespective of whether it is novel or not". They provide further criticism of APR research: "Academia's view of APR is that it should strive to repair complex bugs automatically because this increases the applicability of an APR tool and demonstrates the wider viability of the approach". "Academic research will often claim that the work motivates new research for the improvement of APR application to important and hard to solve bugs. The reality in our case is that, counterintuitively, this makes an APR tool less applicable and less attractive to industry". Given all that, one can clearly understand their APR philosophy: "Bloomberg's approach was based much more around *easy wins* that nonetheless are seen to offer significant benefit to developers, removing manual bug-fixing tasks and freeing up developer time.".

Google's DeepDelta [138] focuses on fixes for compilation errors that "follow a pattern and are highly mechanical". Their deep neural network approach learns patterns from previous fixes to suggest fixes for the two most costly errors in their dataset collected from 300 million lines of Java code. DeepDelta generated the correct repair change for 50% of 38,788 unseen compilation errors; correct changes are in the top three suggested fixes 86% of the time. A repair is deemed correct if at least one of the 10 suggested repairs compiles and exactly matches the fix the developer performed. The goal is to "help developers to repair errors", which may accommodate the not-so-high precision of 50% (lower than the 75-80% range suggested by Christakis and Bird [34]).

Another deep learning-based APR approach [101] was deployed in Samsung to support the migration of mobile applications from Java to Kotlin. The authors' goal was to fix warnings detected by SonarQube. They experimented with different models and report that the best model could fix 19.5% of the violations. The paper does not seem to report any measure of precision.

InferFix [88] is a transformer-based program repair framework paired with the Infer [26] static analyzer. InferFix uses Infer to to detect, localize, and classify a bug; it considers three kinds of bugs detected by Infer (null pointer dereference, resource leak, and thread safety violation). Given a bug of these kinds detected by Infer, it retrieves semantically-similar source code present in a database of historic bugs and fixes. The final step is to generate a fix by prompting a large language model (Codex) finetuned on a dataset of prompts enriched with the information provided by Infer and the retrieved semantically-similar source code. Codex is finetuned with the goal of teaching the model to generate a fix for the given buggy code. Its input is the buggy code augmented with the bug's location and category, and similar fixes (retrieved from the historic database). The model finetuning required extremely powerful hardware (sixty four 32 GB V100 GPU). When evaluated on bugs for Java and C#, InferFix could fix (top-1 prediction) between 57% and 82% of the three categories of bugs. The authors deployed InferFix internally at Microsoft as a GitHub action and as an Azure DevOps plugin operating as part of the continuous integration pipeline of Microsoft's

Developer Division. Such a tight integration adds to what the authors call an end-to-end solution: detection and localization of bugs, and fixing and validation of patches. When a pull request is created for an analyzed project, InferFix proposes fixes for any of the considered bugs detected by Infer. Each candidate patch is packaged as separate pull request, which is individually validated.

Open-source Evaluation

Liu et al. [112] collected changes that fixed FindBugs violations in 730 projects by mining 291,615 commits. They devised an approach that uses convolutional neural networks to learn from these fixes. Their evaluation included submitting pull requests to open-source projects as well as checking whether the approach could fix bugs in Defects4J. The authors submitted 116 patches to open-source projects; 69 were accepted. The authors evaluated 500 of the generated fixes (at most 10 unfixed violations for 50 categories) in terms of correctness. A fix is correct if it compiles, passes all tests, and removes the FindBugs warning. Moreover, the authors manually checked the fix to confirm that semantics is preserved. They found that 25.4% of violations were fixed by the top-1 suggestion and 40.6% were fixed by the top-10 suggestions.

Repairnator [143] is a program repair bot that monitors continuous integration builds of GitHub Java projects. When it finds a failing build, it attempts to generate a patch using several integrated APR tools. At the time of publication it used Nopol [200], Astor [131], and NPEFix [55]. In all, it could generate 12 patches (5 accepted as pull requests) for 6,173 build failures.

Sorald [63], like SpongeBugs (Chapter 3), is an approach to fix Java violations detected by SonarQube. It can fix 10 rules of type “bug”, as classified by SonarQube. Differently from our approach, SpongeBugs, it uses SonarQube’s output to pinpoint a fix location. Moreover, Sorald is integrated into SoraldBot, a bot that monitors changes on GitHub repositories and can generate pull request with fixes. The authors submitted 29 pull requests (17 accepted) to 21 projects.

Styler [120] is another learning approach that mines fixes by traversing commits. However, Styler’s goal is to fix only violations of Checkstyle,{11} a tool that enforces formatting standards for Java. In its evaluation, Styler repaired 41% of almost 27K Checkstyle violations from 104 GitHub projects. It could fix 24 different out of 25 types of rules. The authors analyzed a sample of the 59% violations that were not fixed and found that the generated fixes could introduce new violations or even render the file not parsable.

TADAF [11] is a prototype that aims to detect, fix, and verify TensorFlow (Python) API calls. It statically analyzes Python programs to identify 11 common API misuses patterns extracted from StackOverflow. TADAF generates fixes based on templates for each rule it detects. The authors evaluated TADAF on 5 GitHub projects where it fixed 7 misuses.

2.2.4 Summing Up

When evaluated against a benchmark of bugs, the typical patch of a Java APR technique breaks more functionality than it repairs [144]. *As of now*, most Java APR techniques are not that practical, but they may very well be *in the future*, given we see advances in the literature. As stated from from Marginean et al. [130]: "Automated oracles, and testing and verification will hopefully advance in the years to come, thereby widening the remit of automated repair.".

More practical APR approaches, deployed in industry or evaluated on open-source projects, do not rely on the traditional fault localization approach (i.e., a failing test to pin-point a bug's location); instead, they rely on static analyzers to detect and locate bugs. For this reason, to reach our goal of practicality, we focus on fixing issues detected by static analyzers.

2.3 Static Analysis Tools

Static analysis tools (SATs) are widely used by open-source and commercial software projects to detect possible sources of defects as early as possible in the development process [129]; some organizations even have strict policies requiring the code not to go above a threshold of warnings before being released [125]. SATs work by raising warnings when a piece of code violates a given rule. A rule specifies correct behavior, best practices, or stylistic standards; while rule violations correspond to bugs, deviations from best practices, or breaking formatting standards. Since SATs work statically on the source or bytecode, they can scale to large projects without requiring any form of tests [142].

2.3.1 Static Analysis Warnings

SATs like SonarQube [176] offer a range of *rules* that users may choose for checking. Each rule corresponds to a pattern whose violations the SATs can identify. For example, SonarQube's rule 11863 requires that "Methods should not be empty"; whenever SonarQube finds a method with an empty body, it will report a violation of rule 11863. A SAT's rule violation report is also called a warning (or an issue); therefore, each warning corresponds to the violation of a specific rule at a certain location in the source code. Fixing a warning means modifying the source code so that the SAT no longer triggers that warning. For example, a fix for a violation of rule 1186 consists of adding some executable code or some comments to fill the empty method body.

SATs notoriously report spurious warnings, usually in the form of false positives—warnings that do not represent an actual mistake. It is common to find thousands of warnings reported when a static analysis tool runs for the first time for a project [89]. Unsurprisingly, we have found that developers tend to fix at most 10% of the reported warnings [125]. Despite the low warning fix rates, developers consider SAT warnings important when measuring code quality. Therefore, developers frequently complain about the lack of fix suggestions when using SATs [89].

2.3.2 Static Analysis and Automated Program Repair

Static analysis tools are an attractive target for automatic fix generation, not only due to the high number of unfixed violations but because they also support verification: a fix is verified if the static analyzer does not report a warning anymore after applying the fix for that warning [142]. However, this notion of verification is often not enough. A fix can clear a warning in a way that developers would hardly accept, or, even worse, by deleting functionality. Generating fixes acceptable by developers is a general challenge for APR [70].

Previous research has explored automatic fix suggestions through static analysis. Padioleau et al. [105] proposed Coccinelle to support the evolution of large codebases, such as the Linux kernel. Logozzo et al. [118] fixed simple yet frequently occurring bugs in C#, such as improper initialization of loop variables. Aftandilian et al. [2] extended Java's OpenJDK compiler for error checking by proposing the tool error prone [62]. More recent works specifically target static analysis tools and their violations in different ways: interactively, through the exploration of different fixes [13]; and automatically by mining patterns from programmer-written fixes [16].

SATs can also guide the bug detection of more general bug fixing APR approaches. Three recent studies [10, 16, 114] leverage the mining of past programmer-written fixes for specific SAT warning categories; the mined fixes are used as templates to fix violations of the same warning category. AVATAR [114] recommends code changes based on the output of SATs. It uses the output of FindBugs to guide its fault localization process and its mining process. For its validation step, AVATAR executes the available test suite. AVATAR could fix 34 bugs in the Defects4J benchmark. Getafix [10] mines its fix patterns by exploring commits with messages mentioning SATs violations. Differently than AVATAR, Getafix uses SATs in its fault localization and validation steps. For fault localization, Getafix leverages SATs' detailed reports, which generally contains the exact line of code of a warning. The SAT is also used for the validation: a fix is valid if it makes the warning disappear. Getafix considers six categories of warnings (e.g., null dereference, return should not be null) that often require a change in logic or control flow. The warnings are detected either by Error Prone [62] or Facebook's Infer [85]. Getafix was deployed for 3 months on Facebook, where developers addressed fixes for 250 null dereferences pointed by the approach. PHOENIX [16] is another approach that considers SATs and developers' past fixes for warnings. For its mining process, PHOENIX executed FindBugs on all commits of 517 projects to build a large number of fix suggestions for 234 warning categories. Given a pair of consecutive commits, PHOENIX considers a change as a fix if a violation from a previous commit disappears in the following one. Similarly, a fix is valid if FindBugs no longer reports a violation. When evaluated in a set of $\sim$5.4K violations from 5 popular open-source projects, PHOENIX had a recall of 85% and a precision of 54%. An open question is whether PHOENIX is suitable for being directly used by developers. Although PHOENIX produced 19 patches that were accepted by the open-source projects, the paper's authors carefully selected the patches. Some categories of warnings were completely discarded as "a large fraction were indicated as false positives". Its relatively low precision may make developers skeptical of the usability of static code analysis tools [34, 187].

2.3.3 Summing Up

Static analysis tools are widely used in open-source and commercial software. Despite their several benefits, developers face several difficulties when using them; we highlight two: i) large number of reported violations; and ii) lack of automated fixes. To address these two difficulties, we propose to automatically fix Java SATs' violations that developers already tend to fix manually. We recall that SATs are also widely used in *practical* Automated Program Repair approaches (as seen in Section 2.2) to pinpoint a bug's location. Therefore, targeting SATs' violations can i) help developers on using tools already integrated in their workflows; and ii) lessen the requirements of an APR approach.

2.4 Exception Behavior in Java

Exceptions are often used to signal faulty or undesired behavior [30]. A program may include exception handling code, which executes when an exception is raised to try to recover from the error or at least mitigate it. Java exceptions are a frequent cause of bugs; thus, they represent an interesting domain with practical implications for software developers.

Researchers have investigated how exception handling is done in practice both by mining codebases [7, 48, 86, 97, 98, 149, 173] and by looking at programmers' habits and guidelines [79, 137]. This line of research has revealed that exception-handling code is often complex [170] and among the most poorly understood and scarcely documented parts of a system [58]. Thus, it is not surprising that exceptions are commonly implicated in bugs in a variety of software including Java libraries [151, 196, 205, 206, 207], Android apps [64], and cloud systems [32].

2.4.1 Testing and Debugging

Given that exception handling is often associated with bugs, the literature has investigated how developers test and debug exception bugs.

Researchers have studied the testing practices of developers [50], the characteristics of the tests they write [6, 185], and how they relate to the bugs that are commonly found [188]. Considering technological contributions, a lot of effort has been devoted to bringing more automation to testing. Frameworks such as the popular JUnit [90] and TestNG [181] automate test-case execution by providing syntactic means of defining test inputs and the expected outputs. Other tools can automate the generation of test inputs [67, 156] as well as of oracles—for example in the form of assertions [60, 193, 194].

Even though plenty of research studied testing, only little looked at the intersection of *tests* and *exceptions* beyond test-case generation [49, 74, 94]; as Ebert et al. [59] put it "perhaps surprisingly, there is comparatively little work studying how exception handling code is tested or debugged in practice, with few exceptions [147, 205]". Dalton et al. [43] analyzed tests and exception tests in 417 Java projects. They surveyed 66 developers about their perception of exception behavior testing and reported that respondents agree that developers often neglect exception behavior tests. They also found that 61% of projects with tests also include some exception tests. The Android mobile-programming framework defines and uses numerous framework-specific exception classes. Due to the nature of mobile apps (which are event-driven and use several external resources), writing proper exception handling code in Android is not easy [37, 64], which is why test amplification is a promising technique [205] to *validate* such exception handling code.

Detecting potential exception bugs is among the numerous exception debugging related lines of research. MAESTRO [122] automatically detects potential exception triggering code locations and suggests a relevant StackOverflow post to assist in solving the problem. Its static detection approach mines StackOverflow exception related questions and answers to parse code snippets. MAESTRO then builds a control flow graph that abstracts low-level syntactic details to generalize from the concrete code snippets. For instance, it can detect the same exception bug in both `for` and `while` loops. MAESTRO was evaluated only in situations where it could detect an exception (i.e., no measure of recall); it provided a relevant StackOverflow post for 71% of 78 cases. DREX [108] applies deep learning to identify runtime exceptions that a method might throw. It statically builds a graph-based representation to capture code syntax and semantics and "an attention-based graph neural network that learns fine-grained statement embedding based on weighed syntactic and semantics information". DREX learned from methods that contain a `try`/`catch` from thousands of GitHub projects; code inside a `try` block is considered as a potential exception triggering statement. In their manual evaluation of 50 methods with uncaught exceptions, DREX's authors could successfully write tests to trigger 20 cases (i.e., 40% precision). The authors highlighted that none of the detected unchecked exceptions were detected by static analyzers such as SpotBugs and PMD. Both MAESTRO and DREX use JavaParser [87], a source code parser that builds abstract syntax trees with advanced capabilities, such as resolving references to class and methods. JavaParser enables tools to work on a diverse set of projects as it does not require building or compiling projects. Fully automatically compiling a broad and general set of Java projects is considered an unfeasible task [80]. CATCHER combines static exception propagation analysis with automatic test case generation to detect potential exception throwing code locations. It uses SOOT [189] to identify method calls that propagate runtime exceptions that are effectively uncaught (i.e., not handled by a try/catch). These method calls are used for generating tests by restricting the search space of EVOSUITE [67]. CATCHER was able to generate 77 tests for 21 Java Projects that EVOSUITE could not generate by itself.

2.4.2 Precondition Inference

Automatically inferring preconditions and other specification elements from implementations is a long-standing problem in computer science, which has been tackled with a variety of different approaches. One key motivation is that most developers are reluctant to explicitly express preconditions as assertions or other forms of documentation [163].

Historically, the first approaches used *static analysis* and thus were typically sound (the inferred specification is guaranteed to be correct, that is, 100% precision) but incomplete (not all specifications can be inferred, that is low recall), and may not apply to all features of a realistic programming language [39, 40, 41, 117, 172]. Daikon [60] was the first, widely successful approach that used *dynamic analysis*, which offers a different trade-off: it is unsound (the "inferred" specifications are only "likely" to be correct) but it is applicable to any program that can be executed. Daikon produces these assertions by observing properties that hold during executions of the system [163].

Dietrich et al. [52] looked at how Java developers implement lightweight precondition checking by analyzing 176 projects hosted on Maven central. The authors do a thorough job to define different strategies that developer use to check and enforce preconditions. The main strategies include: i) Conditional Runtime Exceptions (CRE) which includes exceptions like `IllegalArgumentException`, `NullPointerException`, and `IllegalStateException`; ii) the use of APIs (e.g., Apache Commons Validate, Guava Preconditions); iii) assertions (i.e., `assert`); iv) the use of annotations (e.g., `@Nullable` and `@NotNull`). The first three are runtime strategies, while annotations are checked during compile-time. The authors found that 160 (91%) of the projects use some kind of precondition checking, when looking at the latest versions of the analyzed projects. API usage was surprisingly low (at most 15%), according to the authors. CRE was the most used strategy; used by 155 (97%) of the 160 projects. The OpenJDK project was the project that employed CRE the most. The authors conclude that precondition checking may be explained by the high level reuse of library code in open-source programs. "Modern libraries have to provide defensive API surfaces to deal with unknown clients". Thus, by throwing exceptions such as `IllegalArgumentException`, a library shifts the responsibility of complying to a precondition to the client. Dietrich et al. lists several practical advantages to this strategy, including clear debugging (i.e., exception in the stack trace) and less workload for the libraries maintainers.

In general, inference techniques usually analyze *a*) client code; *b*) API documentation; *c*) API code [207]. Several approaches infer preconditions from the client code by analyzing code that invokes a given API [151, 167, 168, 174, 182, 192, 203]. The rationale is that patterns used by many clients of the same API are likely to indicate suitable ways of using that API's methods. More recently, approaches based on natural language processing (NLP) have gained traction [20, 157, 179, 191, 191, 208]. NLP can analyze artifacts other than program code (e.g., comments and other documentation); on the other hand, machine learning is usually based on statistical models, and hence it cannot guarantee correctness and may be subject to overfitting [84, 162]. The work on Toradocu [75] and its later extension Jdoctor [20] is a relevant representative of the capabilities of natural language processing techniques to extract (exception) preconditions of Java methods. Toradocu/Jdoctor's preconditions are Java Boolean expressions; thus, they can be directly used to generate test oracles or other kinds of executable specification. In its experimental evaluation on widely used Java libraries, Jdoctor achieved a recall of 83% and a precision of 92%.

Techniques that work on the API code target the implementation of classes and their methods [207]. Some approaches have explored Java exception preconditions. Buse and Weimer's work [24] targets the documentation of exception behavior by instrumenting the bytecode of a Java application. Their approach outputs exception preconditions that could often improve or complement human-written documentation. However, their exception preconditions are not guaranteed to be correct, nor were they evaluated quantitatively in precision and recall. SnuggleBug [29] infers preconditions that characterize the reachability of a goal state from an entry location. It works on bytecode to analyze general preconditions, including exception preconditions (i.e., having a goal of a null dereference expression). SnuggleBug can handle method calls and can scale to real-world Java projects. Its evaluation was done exclusively on exceptions thrown by the JVM. PaRu [207] is an automated technique that analyzes source code and Javadoc documentation to link method parameters to exception behavior. Its goal is to "identify as many links as possible", so PaRu does not interpret any rules nor infer preconditions. PaRu outputs a mapping between parameters and throw statements that depend on them. Drone [210] compares the exception behavior of source code to that described in Javadoc in order to find inconsistencies. It analyzes a program's control flow statically and uses constraint solving to find inconsistencies between the code and documentation. Drone is primarily designed to run on projects with *some* documentation, in order to detect inconsistencies and omissions.

The idea of using techniques to first extract preconditions from libraries, and then to analyze client code of these libraries has not been systematically explored. Zeng et al.'s recent work [201] experiments with the idea of combining the results of library and client analysis. Their work is not specific to exception behavior but targets different kinds of API misuses and information sources (including Javadoc natural language documentation, call graphs, method names, and annotations); thus, they can potentially report a broader variety of API misuses, but with weaker guarantees of precision compared to our experiments. In addition, there is a natural trade off between breadth of detected misuses and how easily addressable they are; our focus on exception preconditions can lead to actionable (and possibly even automatic) code improvement suggestions.

2.4.3 Repairing Exception Behavior

Researchers have explored strategies for mitigating or avoiding errors from exceptions, both dynamically and statically. Fixes at runtime on the deployed application—often related to the concept of software healing [70]—aim at assuring program availability in the case of failures [70]. Runtime repairs techniques targeting exceptions generally involve manually writing patches first and then dynamically applying them when a crash occurs [25, 28, 31]. An approach targeting Android apps [9], can automatically avoid crashes by dynamically disabling crash inducing functionalities.

APR tools that focus on exceptions localize potential faulty statements statically [51, 54], by analyzing and modifying test executions [38], or by relying on failing tests [55, 99, 119]. Most tools [54, 55, 99] target Java null pointer exceptions or other exceptions commonly thrown by the Java virtual machine (e.g., index out of bounds and class cast). Their fixes include surrounding the offending code with an if statement that checks whether a reference is null and either skips the exception-throwing statement or provides alternative/default values for the null object. CLOTHO [51] focuses on exceptions caused by improper string manipulation; it statically detects vulnerable statements and surrounds them with try/catch blocks where the catch block aims at preserving the original code intent. A more general approach [38] aims at increasing a program's resilience and avoiding crashing when an uncaught exception occurs. It works by modifying the exception handling code exercised by tests to add generic throw statements at the beginning of a try block. If the newly thrown exception fails the tests, the catch clause is modified to capture a more generic exception. While this approach may prevent a program from crashing, using catch clauses that capture generic exception types is usually considered an anti-pattern [137, 173].

In the last couple of years, researchers proposed APR approaches focusing on exceptions with novel contributions to the fault localization and validation steps. EXCEPT [73] is a technique that enhances fault localization by focusing on the semantics of exceptions rather than on the correlation between executed statements and failed tests. It returns a ranked list of repair targets by analyzing the stack trace of an exception thrown by a failing test. A stack trace is the list of methods in the call stack up to the moment an exception is thrown.{12} EXCEPT's key insight is to differentiate the ranking of likely fault locations based on the type of 4 frequently thrown exceptions. It analyzes different expressions and locations depending on the exception. In its evaluation of exception bugs in Defects4J, EXCEPT outperformed Ochiai [1], a popular spectrum-based fault localization approach. An other approach, NPEX [107] fixes `NullPointerException` thrown implicitly by the JVM without needing tests. It automatically infers the repair specification of the buggy program to validate its generated patches. NPEX takes a stack trace as input and then uses static symbolic execution to infer the specification of the method that throws the exception. It could correctly fix 51% of 119 bugs taking on average ∼3 minutes per bug. EXCEPT and NPEX both work on exception stack traces, which assume that an exception has been thrown in the first place.

2.4.4 Exception API Misuses

APIs typically throw exceptions to signal incorrect calls (for instance, invalid parameters). Therefore, API misuses can often be linked to exceptions [196] (an invalid argument like `null`) or to missing exception-handling code [109, 206] (omitting a try-catch block). Static API misuse detectors are often limited with respect to exception behavior [4]; and Automated Program Repair approaches could benefit from better API-misuse detection capabilities [96].

Static analysis tools such as Infer,{13} Coverity Scan,{14} SpotBugs,{15} and SonarQube all have rules that check for null dereferences: a null dereference occurs in a piece of code like `s.length()` when `s` is `null`. To our knowledge, these static analyzers do not generally report exceptions that may be thrown from calls to external libraries. SonarQube does have rules that check API misuses of common Java libraries (e.g., JUnit, Spring, and Mockito), as well as the JDK core APIs (e.g., use an overloaded signature of `String.indexOf()` when looking for a single character); however, it lacks rules that look for general exceptions thrown in external libraries.

2.4.5 Summing Up

Exceptions are frequent cause of bugs in Java applications of several domains. Although they have been studied from multiple perspectives, the literature shows that static API misuse detectors do not perform well on misuses related to exceptions. Moreover, static analyzers also do not generally provide rules that check for exceptions originated from calls to external libraries. Therefore, analyzing exception behavior can help developers on a frequent occurring problem and yet insufficiently covered by current approaches.

Part II

Repairing Static Analysis Warnings

3

Automatically Generating Fix Suggestions in Response to Static Code Analysis Warnings

Static code analysis tools such as FindBugs and SonarQube are widely used on open-source and industrial projects to detect a variety of issues that may negatively affect the quality of software. Despite these tools' popularity and high level of automation, several empirical studies report that developers normally fix only a small fraction (typically, less than 10% [125]) of the reported issues—so-called "warnings". If these analysis tools could also automatically provide *suggestions* on how to fix the issues that trigger some of the warnings, their feedback would become more actionable and more directly useful to developers.

We investigate whether it is feasible to automatically generate fix suggestions for common warnings issued by static code analysis tools, and to what extent developers are willing to accept such suggestions into the codebases they're maintaining. To this end, we implemented a Java program transformation technique, called SpongeBugs, that fixes 11 distinct rules checked by two well-known static code analysis tools (SonarQube and SpotBugs). Fix suggestions are generated automatically based on templates, which are instantiated in a way that removes the source of the warnings; templates for some rules are even capable of producing multi-line patches. We submitted 38 pull requests, including 946 fixes generated automatically by our technique for various open-source Java projects, including the Eclipse UI—a core project of the Eclipse IDE—and both SonarQube and SpotBugs tools. Project maintainers accepted 87% of our fix suggestions (97% of them without any modifications).

Structure of the Chapter

- Section 3.1 provides motivation for this chapter.
- Section 3.2 describes SpongeBugs' approach and its implementation.
- Section 3.3 describes the experimental design of SpongeBugs' evaluation.
- Section 3.4 discusses our results and findings.
- Section 3.5 presents the threats that affect the validity of our work.

- Section 3.6 draws our conclusions on SpongeBugs.

3.1 Introduction

Static code analysis tools (SATs) are becoming increasingly popular as a way of detecting possible sources of defects earlier in the development process [77]. By working *statically* on the source or byte code of a project, these tools are applicable to large code bases [89, 114], where they quickly search for patterns that may indicate problems—bugs, questionable design choices, or failures to follow stylistic conventions [13, 186]—and report them to users. There is evidence [17] 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 [77, 114, 125]. Some projects' development rules even require that code has to clear the checks of a certain SAT before it can be released [8, 17, 125].

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 [77, 89], which saturates the developers' capability of sifting through them to select those that are more relevant and should be fixed [125]. 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 [89, 125].

Our research 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, whose current implementation works on Java code. SpongeBugs detects violations of 11 different rules checked by SonarQube and SpotBugs (successor to FindBugs [77])—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 suggests and presents a fix to the user.

By construction, the fixes SpongeBugs suggests remove the origin of a rule's violation, but the maintainers still have 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, we present the result of an empirical evaluation where we applied it to 12 open-source Java projects, and submitted 946 fix suggestions as pull requests to the projects. At the time of writing, project maintainers accepted 825 (87%) fix suggestions—97% 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 perfor-*

mance to large code bases. SpongeBugs is also *accurate*, as it rarely generates false positives (spurious rule violations). We actually found several cases where SpongeBugs correctly detected cases of rule violations that were missed by SonarQube.

To further demonstrate SpongeBugs's versatility, Section 3.4 also discusses how SpongeBugs complements program repair tools (e.g., AVATAR [114]) and how it performs on software whose main contributors are non-professionals (i.e., students).

With few exceptions—which we discuss throughout our evaluation to inform further progress in this line of work—SpongeBugs worked as intended by providing sound, easy to apply suggestions to fix static rule violations.

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 nontrivial 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.

3.2 SpongeBugs: Approach and Implementation

SpongeBugs provides fix suggestions for violations of selected rules that are checked by SonarQube and SpotBugs. Section 3.2.1 discusses how we selected the rules to check and suggest fixes for. SpongeBugs works by means of source-to-source transformations.

3.2.1 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 [89]. To avoid creating fixes to spurious warnings, we base our design on the assumption that rules whose violations are frequently fixed by developers are more likely to correspond to real issues of practical relevance [112, 125].

We collected and analyzed the publicly available datasets from three previous studies that explored developer behavior in response to output from SonarQube [53, 125] and FindBugs [112]. 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`.[16] 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 impractical, since we would need to identify and check every call of this method, and potentially introduces a breaking change [22]. 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 from developers, since pathnames are context specific; it would also need access to the application's execution environment, which is clearly beyond the scope of the source code under analysis.

We selected the top rules (in order of how often developers fix the corresponding warnings) that satisfy these feasibility criteria, leading to the 11 rules listed in Table 3.1. Note that SonarQube and SpotBugs rules largely overlap, but the same rule may be expressed in slightly different terms in either tool. Since SonarQube includes all 11 rules we selected, whereas SpotBug only includes 7 of them, we use SonarQube rule identifiers.{17}

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.

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.

ID	SONARQUBE ID	RULE DESCRIPTION
B1	S4973	Strings and boxed types should be compared using `equals()`
B2	S2111	`BigDecimal(double)` should not be used
C1	S1192	String literals should not be duplicated
C2	S3027	String functions use should be optimized for single characters
C3	S1643	Strings should not be concatenated using + in a loop
C4	S2130	Parsing should be used to convert strings to primitive types
C5	S1132	Strings literals should be placed on the left-hand side when checking for equality
C6	S2129	Constructors should not be used to instantiate `String`, `BigInteger`, `BigDecimal`, and primitive wrapper classes
C7	S2864	`entrySet()` should be iterated when both key and value are are needed
C8	S1155	`Collection.isEmpty()` should be used to test for emptiness
C9	S1596	`Collections.EMPTY_LIST`, `EMPTY_MAP`, and `EMPTY_SET` should not be used

Rules C1 and C5 were selected *indirectly* on top of the feasibility criteria discussed above (which were used directly to select the other 9 rules). We selected rule C1 because it features very frequently among the open issues of many projects; its fixes are somewhat challenging since they involve multiple lines and the insertion of a constant. We selected rule C5 because it can be fixed in conjunction with fixes to 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-hand side when checking for equality) applied in conjunction.

3.2.2 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 [103], 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 the latest Rascal's Java grammar [45], which targets Java 8, thus our evaluation 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 C2 (String functions use should be optimized for single characters). Step 1 performs a fast, but potentially imprecise, search that is based on some textual necessary conditions for a rule to be triggered. For rule C2, step 1 looks for files that have a method call to either `lastIndexOf()` *or* `indexOf()`–as shown in Listing 3.2.

```
bool shouldContinueWithASTAnalysis(loc fileLoc) {
   javaFileContent = readFile(fileLoc);
   return findFirst(javaFileContent, ".lastIndexOf(\"") != -1 ||
      findFirst(javaFileContent, ".indexOf(\"") != -1;
}
```

Listing 3.2. Implementation of step 1 for rule C2: find textual patterns that might represent a violation of rule C2.

Step 1 may report false positives: for rule C2, the call to `lastIndexOf()` or `indexOf()` may not actually involve an instance of a `String`, or the argument of the function might not be a single character. Step 2 is more precise, but also more computationally expensive, as it performs a full AST matching; therefore, it is only applied after step 1 identifies code that has a high likelihood of being rule violations.

In our example of rule C2, step 2 checks that the target of the possibly offending call to `indexOf()` is indeed of type `String` *and* the argument is a single character—as shown in Listing 3.3.

```
case (MethodInvocation)
  ‘<Primary varName>.<TypeArguments? ts>indexOf(<ArgumentList? args>)‘: {
    if (isVarAString(mdl, varName) && isSingleArgOfInterest(args)) {
```

Listing 3.3. Partial implementation of step 2 for rule C2: 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 C2, step 3's template is straightforward: replace the single character String (double quotes) with a character (single quote)—its implementation is in Listing 3.4. (Steps 2 and 3 for rule C2 also handle other patterns not shown in this example for brevity.)

```
argAsChar = parseSingleCharStringToCharAsArgumentList(argsStr);
insert (MethodInvocation) ‘<Primary varName>.<TypeArguments? ts>indexOf(<ArgumentList
    ↪ argAsChar>)‘;
```

Listing 3.4. Implementation of step 3 for rule C2: instantiate the fix templates corresponding to the violated rule.

Note that step 1 is susceptible to differences in layout (e.g., a newline character between a method call and the first argument). In contrast, step 2 is not affected by layout difference due to Rascal's implementation.

3.3 Empirical Evaluation of SpongeBugs: Experimental Design

The general goal of SpongeBugs's experimental evaluation is to investigate the use of techniques that suggest fixes to warnings generated by static code analysis tools. Section 3.3.1 presents the research questions we answer in this empirical study, which targets:

- 15 open-source projects selected using the criteria we present in Section 3.3.2;
- 5 student projects developed as part of software engineering courses;
- Defects4J: a curated collection of faulty Java programs, widely used to evaluate automated program repair tools.

3.3.1 Research Questions

The empirical evaluation of SpongeBugs, whose results are described in Section 3.4, addresses the following research questions, which are based on the original motivation behind this work: automatically providing fix suggestions that helps to improve the practical usability of SATs.

RQ1. How widely *applicable* is SpongeBugs?
The first research question looks into how many rule violations SpongeBugs can detect and suggest a fix for. We also analyze cases in which the SpongeBugs's detection of rule violations are not accurate.

RQ2. 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. How *efficient* is SpongeBugs?
The third research question evaluates SpongeBugs's scalability in terms of running time on large code bases.

RQ4. How does SpongeBugs perform on code written by *non-professionals*?
The fourth research question runs SpongeBugs on projects developed by *students*, to see whether its suggestions have a different impact than those about code written by professionals.

RQ5. How does SpongeBugs work on code with *semantic* bugs?
The fifth research question runs SpongeBugs on Defects4J, a curated collection of semantic bugs in real-world Java program; while SpongeBugs is not designed to fix these kinds of behavioral bugs, it is interesting to see how its heuristics interact with code that has different kinds of errors.

3.3.2 Selecting Projects for the Evaluation

The evaluation of SpongeBugs uses different Java projects, which we describe in the following subsections.

Open-Source Projects

The bulk of the evaluation—addressing RQ1, RQ2, and RQ3—targets large open-source Java projects, which provide a real-world usage scenario. 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 requested{18} help with fixing SonarQube issues. We selected the other twelve projects, following accepted best practices [93], 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 Table 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.

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 non-blank non-comment 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.

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-blank non-comment lines of code calculated from Java source files using `cloc` ({19})

Student Projects

The effort devoted to improving code quality is likely to be different in projects developed by students as opposed to large open-source projects such as those that we identified in Section 3.3.2. SpongeBugs can still be effective on both kinds of projects, but the impact and scope of its suggestions may differ.

To investigate this aspect, and to demonstrate SpongeBugs's applicability on heterogeneous software, we considered 5 student projects from courses taught at USI (this thesis's author's affiliation)[1] in the latest two years. Table 3.3 summarizes the projects' characteristics. We selected these projects because they were (mainly) written in Java, of substantial size, and developed by students with experience (that is, non-beginners).

Projects 1 and 2 were developed by undergraduate students of "Software Atelier 4", a software engineering project course where groups of about 12 students work to develop an application following realistic best practices of code development and team coordination. The projects included a front-end written in JavaScript, which we ignored for the purpose of evaluating SpongeBugs. The projects required students to use SonarQube to spot potential problems in their code, and to monitor test coverage.

Projects 3, 4, and 5 were developed by master's students of "Software Analytics", a course

[1]However, the courses were not taught by the author.

Table 3.3. The student projects we analyzed using SpongeBugs. For each project, the table reports whether it was developed by Undergraduates or Master's students (LEVEL); the number of students working on the project (# STUDENTS); its size in non-blank non-comment lines of code (LOC); and whether it was already analyzed using SonarQube by the students (SQ?).

PROJECT	LEVEL	# STUDENTS	LOC	SQ?
Project 1	U	12	6.8 K	Yes
Project 2	U	13	5.2 K	Yes
Project 3	M	7	2.5 K	Yes
Project 4	M	8	2.4 K	Yes
Project 5	M	4	3.6 K	No

about using and building tools to monitor software development artifacts and their evolution; each project was developed by a group of 4–8 students. As for projects 1 and 2, we only considered the project modules that are written in Java. Project 5 did not require students to use SonarQube, whereas projects 3 and 4 did.

Curated Collection of Bugs

As we discussed in detail in Section 3.2, SpongeBugs's targets SAT rules that are *syntactic* and *modular*. This is a deliberate restriction of SpongeBugs's applicability, but also one that makes it very effective in its domain.

In contrast, the related work on "automated program repair" typically targets the more diverse category of *semantic* (*behavioral*) bugs, which include any program behavior that deviates from the intended one. Defects4J has become a popular benchmark to evaluate the performance of such tools for Java. It is a curated collection of (mostly semantic) bugs found in open-source Java projects. Each bug in Defects4J comes with some tests that trigger it, as well with a programmer-written patch that corrects the behavior as intended. Table 3.4 lists the basic characteristics of the code included in Defects4J (version 1.5.0).

In order to get a better idea of the difference between syntactic and semantic bugs, we ran SpongeBugs on all 438 Defects4J bugs. Precisely, we ran SpongeBugs on the *buggy* version of each program in Defects4J. While we don't expect SpongeBugs to repair the bugs (SpongeBugs targets violations of different rules), this experiment can shed light on the interaction between the syntactic modifications introduced by SpongeBugs and the semantic behavior of programs (as checked by the tests in Defects4J). In other words, we would like to ascertain that SpongeBugs's suggestions generally do not adversely interfere with the intended program behavior, and they can be applied even when the code is buggy (as it often is).

3.3.3 Submitting Pull Requests With Fixes Made by SpongeBugs

After running SpongeBugs on the 15 open-source projects listed in Table 3.2, we submitted the fix suggestions it generated as pull requests (PRs) in the project repositories. Following

Table 3.4. A summary of the code included in Defects4J, grouped by the Java project it comes from. For each group, we list the overall size in non-blank non-comment lines of code, the number of available tests, and the number of bugs triggered by the tests.

PROJECT		LOC	TESTS	BUGS
Chart	JFreechart	96 K	2,278	26
Closure	Closure Compiler	90 K	8,300	176
Lang	Apache Commons-Lang	22 K	2,341	65
Math	Apache Commons-Math	84 K	3,619	106
Mockito	Mockito Framework	11 K	1,546	38
Time	Joda-Time	30 K	4,186	27
TOTAL		330.5 K	22,270	438

suggestions to increase patch acceptability [180], 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 interest. We only submitted pull requests to the projects that replied with an answer that was not openly negative. The single negative response pointed out that the project was discontinued. However, fixes were still welcome to a newer project which involved the same team. We received this reply after we concluded our pull request submissions, so we did not consider the suggested project.

While the actual code patches in submitted pull requests 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 [165]. 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 a change, we always mentioned which rule a patch is fixing—also providing a link to SonarQube's official 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

Table 3.5. Responses to our inquiries about whether it is OK to submit a pull request to each project, how many pull requests were eventually submitted and approved, and the average time that each project took to accept or not a pull request.

		PULL REQUESTS		
PROJECT	OK TO SUBMIT?	SUBMITTED	APPROVED	AVG. TIME TO DECISION
Eclipse IDE	Positive	9	9	3.44 days
SonarQube	–	1	1	1.07 days
SpotBugs	Neutral	1	1	2.15 hours
atomix	Positive	2	2	12.47 days
Ant Media Server	Positive	3	3	18.33 hours
database-rider	Positive	4	4	14.66 hours
ddf	Positive	3	2	2.33 days
DependencyCheck	Neutral	1	1	10.18 hours
keanu	Positive	3	0	–
mssql-jdbc	Positive	1	1	27.10 days
Payara	Positive	6	6	9 hours
primefaces	Positive	4	4	16.5 minutes
cassandra-reaper	No reply	–	–	–
db-preservation-toolkit	No reply	–	–	–
matrix-android-sdk	Negative	–	–	–
	Total:	38	34	–

was straightforward thanks to the nature of the fix suggestions built by SpongeBugs: fixes are mostly independent, and one fix never spans multiple classes. Moreover, SpongeBugs can be easily configured to apply fixes for a single rule at a time.

Overall, the manual effort required to generate the pull requests was low. The most time-consuming task was related to fixing some projects' style requirements, which we discuss in more detail in Section 3.5. However, the general process of contributing to a project might involve several other tasks, not always source code related. Projects Eclipse IDE, Payara, and mssql-jdbc required the signature of a Contributor Agreement License (CLA) to handle intellectual property concerns. For project Payara, we had to physically sign and scan a printed version of the contract that was validated through e-mail. Project Eclipse IDE uses the Gerrit Code Review platform,{20} which required the creation of a web account in the platform, and also extra configurations in the Eclipse IDE itself for submitting contributions. Moreover, Eclipse IDE might require changes on the source code to modify version numbers in MANIFEST.MF files.{21} This versioning process is not intuitive and caused the reviewers and us some nuisances, as in one pull request, a reviewer suggested a version change that later needed to be reverted.{22} Finally, project Ant-Media-Server requires that all source code modifications must be done by members of the GitHub organization responsible for the project. This requirement led to the author of this thesis being added as a member of the organization.

We consider a pull request *approved* when reviewers indicate so in the GitHub interface, alternatively, for the case of project Eclipse IDE, when the code changes receive positive review points. Only one pull request was approved but not merged. Since merging depends on other aspects of the development process that are independent of the correctness of a fix, we do not distinguish the approved pull request from those that were merged.[2]

The reviewing process may approve a patch with or without modifications. For each approved patch generated by SpongeBugs we record whether it was approved with or without modifications. Table 3.6 shows a detailed overview of all pull requests submitted for each rule.

Table 3.6. Pull requests submitted for each rule with the total percentage of accepted fixes.

			ACCEPTED FIXES %		
RULE	# PRS	# FIXES	W/O MOD.	WITH MOD.	TOTAL
B1	1	1	100%	–	100%
B2/C6	2	101	75%	25%	100%
C1	6	289	97%	–	97%
C2	8	195	100%	–	100%
C3	2	16	88%	12%	100%
C4	3	35	100%	–	100%
C5	6	181	50%	–	50%
C7	6	65	74%	1%	75%
C8	2	30	83%	–	83%
C9	2	33	100%	–	100%
Total	38	946	84%	3%	87%

3.4 Empirical Evaluation of SpongeBugs: Results and Discussion

The results of our empirical evaluation of SpongeBugs answer the five research questions presented in Section 3.3.1. For uniformity, all experiments related to RQ1–3 target the 12 projects whose maintainers were accepting of pull requests fixing static analysis warnings (top portion of Table 3.5).

3.4.1 RQ1: Applicability

To answer **RQ1** ("How widely applicable is SpongeBugs?"), we ran SonarQube on each selected open-source project, counting the warnings triggering 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 had been fixed. Table 3.7 shows the results of these experiments. Overall, SpongeBugs removes 85% of all warnings violating the rules we considered in this research.

[2]The only case is a pull request to Ant Media Server that was approved but violates the project's constraint that new code must be covered by tests.

Table 3.7. 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 been fixed 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.

PROJECT	B1	B2	C1	C2	C3	C4	C5	C6	C7	C8	C9	TOTAL	FIXED %
Eclipse IDE	41/5	13/10	214/199	4/4	3/2	19/3	189/176	17/11	–	138/94	102/97	740/601	81%
SonarQube	–	–	104/94	–	–	–	7/7	–	–	–	–	111/101	91%
SpotBugs	12/8	1/0	289/247	2/1	1/0	11/1	141/141	–	–	30/26	–	486/424	87%
atomix	1/1	–	57/57	–	–	–	9/9	–	–	1/0	2/1	70/68	97%
Ant Media Server	–	–	28/28	3/2	1/0	2/1	23/23	3/0	–	4/2	4/4	68/60	88%
database-rider	–	–	5/5	5/5	–	–	2/2	–	1/1	1/1	–	14/14	100%
ddf	1/0	–	104/98	–	1/0	–	88/86	–	1/1	45/37	8/8	247/230	93%
DependencyCheck	–	–	61/51	10/9	–	–	3/3	–	–	4/2	–	78/65	83%
keanu	1/1	–	–	–	–	–	4/4	–	12/12	5/5	–	22/22	100%
mssql-jdbc	4/1	–	314/282	14/1	–	7/1	58/58	2/0	–	14/14	–	413/357	86%
Payara	39/36	–	1,413/1,305	214/169	61/14	114/10	1,830/1,627	200/88	50/44	438/301	58/50	4,417/3,644	82%
primefaces	–	–	336/286	11/9	6/6	3/3	336/329	–	1/1	1/0	4/4	698/638	91%
TOTAL	99/52	14/10	2,925/2,652	263/200	71/22	156/19	2,690/2,465	222/99	65/59	681/448	178/164	7,364/6,224	–
FIXED %	53%	71%	91%	76%	31%	12%	92%	45%	91%	71%	92%	85%	–

These results justify our decision of focusing on a limited number of rules. In particular, the two rules (C3 and C4) with the lowest percentages of fixing are responsible for approximately 3% 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.

A widely applicable kind of suggestion are those for violations of rule C1 (String literals should not be duplicated), shown in Listing 3.5, which SpongeBugs can successfully fix in 91% 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 duplicate.

```
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 3.5. 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 3.6 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 3.6. 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 3.7, 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".

```
 1  public final class CliParser {
 2
 3  - final Option outputFormat = Option.builder(ARGUMENT.OUTPUT_FORMAT_SHORT)
 4  - .argName("format").hasArg().longOpt(ARGUMENT.OUTPUT_FORMAT)
 5  + final Option outputFormat = Option.builder(ARGUMENT.OUTPUT_FORMAT_SHORT)
 6  + .argName(OUTPUT_FORMAT).hasArg().longOpt(ARGUMENT.OUTPUT_FORMAT)
 7
 8     public static class ARGUMENT {
 9        public static final String OUTPUT_FORMAT = "format";
10     }
11  }
```

Listing 3.7. Example of an incorrect fix due to a false positive violation of rule C1. Line 6 references constant `OUTPUT_FORMAT` which is not available as an unqualified name.

Detection Accuracy

To better evaluate SpongeBugs's applicability, we investigate when its analysis produces *false positives* and *false negatives*. A false positive is a rule violation that is erroneously reported; in these cases, SpongeBugs produces a suggestion that changes something that need not be changed. A false negative is a rule violation that is *not* reported; in these cases, SpongeBugs should produce some suggestion that is instead missing.

From the point of view of *usability*, false positives are those with the greater potentially negative impact. Indeed, a high false-positive rate is one of the key reasons that limit the adoption of SATs [89]: a user flooded with many false positives quickly concludes that the tool is not reliable because it points to a lot of violations that are incorrect or irrelevant [112]. In contrast, false negatives are a minor problem *as long as* a tool is still widely applicable and reports suggestions that help improve the design or other quality attributes of the program.

Since SpongeBugs provides suggestions to enforce rules that are checked by SonarQube, we use SonarQube's detected rule violations as ground truth[3] to count SpongeBugs's false positives and false negatives:

- A suggestion provided by SpongeBugs for which SonarQube reports no rule violation is a *false positive*;
- A rule violation reported by SonarQube for which SpongeBugs provides no suggestions is a *false negative*.

[3]Remember that SpongeBugs does not use SonarQube's output to identify rule violations but performs its own detection; otherwise this discussion would be moot.

As we discuss in detail in the following subsections, SpongeBugs generates *very few* false positives (a tiny fraction of all violations it detects), and many of these are actually misdetections by SonarQube. In contrast, SpongeBugs generates a more significant number of false negatives (19% of all violations SonarQube detects); but these are not so problematic for applicability since they simply reflect some design decisions that trade off some detection capabilities in exchange for soundness.

False Positives

Table 3.8 reports SpongeBugs's false-positive rate in each project and for each rule. Overall, only 0.6% of all fix suggestions provided by SpongeBugs do not correspond to a violation reported by SonarQube. Such a low rate of false positives corroborates the data about SpongeBugs's accuracy and hence practical relevance.

To better understand the few cases where SpongeBugs generates false positives, we classify all of them into categories according to their origin. We found out that the majority of false positives (23 out of 37 cases) are actually likely *not* false positives but rather false negatives of SonarQube's detection.

Table 3.8. For each project and each rule checked by SonarQube, the table reports the number of false positives (a fix suggestion provided by SpongeBugs for which SonarQube reported no rule violation). The bottom rows summarize the TOTAL number of false positives, and what percentage of the overall violations reported by SonarQube these false positives correspond to; similarly the rightmost column reports the TOTAL per project.

PROJECT	B1	B2	C1	C2	C3	C4	C5	C6	C7	C8	C9	TOTAL
Eclipse IDE	0	0	3	0	0	4	0	0	1	3	1	12
SonarQube	1	0	0	0	0	1	0	0	2	0	0	4
ddf	0	0	6	0	0	0	0	0	0	0	0	6
Payara	0	0	2	0	0	12	0	0	0	1	0	15
TOTAL	1	0	11	0	0	17	0	0	3	4	1	37
FP %	1.9%	0%	0.4%	0%	0%	47.2%	0%	0%	4.8%	0.8%	0.6%	0.6%

True Positives According to SonarLint. SonarLint is a plugin that integrates SonarQube inside the Eclipse IDE. Somewhat surprisingly, SonarLint disagrees with SonarQube's analysis on 4 rule violations that were fixed by SpongeBugs but not reported by SonarLint. It is possible this disagreement is due to different filtering rules (or warning prioritization rules) used by SonarLint and SonarQube. In any case, these 4 violations can be considered true positives (even though we classified them as false positives when using SonarQube's output as ground truth).

True Positives According to Developers. Out of all fix suggestions built by SpongeBugs that were accepted as pull requests by developers (see Section 3.3.3), 17 do not correspond to any rule violation reported by SonarQube. All but one of these fix suggestions correspond to violations of rule C4 (Parsing should be used to convert strings to primitive types); 4 of them were accepted by the maintainers of Eclipse IDE and 12 by the maintainers of Payara. This convincingly indicates that most, if not all, of these cases are true positives (and false negatives of SonarQube); at the very least, SpongeBugs's suggestions are considered not harmful by maintainers of the code.

Listing 3.8 shows an example of SpongeBugs fix suggestion that does not correspond to a rule violation according to SonarQube. In order to understand why SonarQube failed to report this as a violation, we looked for other similar violations of rule C4 that were instead reported by SonarQube as well as fixed by SpongeBugs, such as line 9 in Listing 3.9. It turns out that if we remove the outer parentheses in subexpression `(Double.valueOf(value))` SonarQube will report a violation of rule C4 in Listing 3.8 as well. The sensible conclusion is that this is a miss in SonarQube's detection.

```
public static double asDouble(String value) throws DataFormatException {
    try {
- return (Double.valueOf(value)).doubleValue();
+ return Double.parseDouble(value);
    } catch (NumberFormatException e) {
       throw new DataFormatException(e.getMessage());
    }
}
```

Listing 3.8. Fix suggestion generated by SpongeBugs for a violation of rule C4 not detected by SonarQube.

```
1  public int getInt(String key) throws NumberFormatException {
2     String setting = items.get(key);
3        if (setting == null) {
4           // Integer.valueOf(null) will throw a NumberFormatException and
5           // meet our spec, but this message is clearer.
6           throw new NumberFormatException(
7                 "There is no setting associated with the key \"" + key +
                      ↪ "\"");//$NON-NLS-1$ //$NON-NLS-2$
8        }
9        return Integer.valueOf(setting).intValue();
10 }
```

Listing 3.9. Violation of rule C4 detected by both SonarQube and SpongeBugs.

We found several other examples of fragile behavior of SonarQube detecting violations of rule C4, which betray failures of its analysis algorithm. Like every static analyzer, SonarQube sometimes limits the cases in which a rule is checked, to improve scalability and precision of detection in the other cases. SpongeBugs is no different, but it sometimes achieves different trade-offs than SonarQube—hence the discrepancies we observed in these experiments. By and large, however, SpongeBugs and SonarQube's detection results are consistent and correct.

Actual False Positives. Only 17 out of all fix suggestions produced by SpongeBugs are actual false positives: they correspond to spurious rule violations. In 5 of these cases we could find a programmer's annotation that explicitly turns off checking of certain rules; unfortunately, SpongeBugs does not process these annotations, and hence it will obliviously flag what it considers a violation.

Listing 3.10 shows examples of such annotations that should suppress detection. On line 1 a generic annotations suppresses checking all rules in the whole class; on lines 4–7 an annotation turns of two specific rules within a method; on line 15 a special comment turns off checking a specific rule on the same line where the comment appears.[4]

```
1  @SuppressWarnings("all") // prevents detection of all rules within the entire class
2  public class JavaClass {
3
4     @SuppressWarnings({ // suppress detection of two rules within aMethod()
5        "squid:S1192", // S1192 is SonarQube's identifier for rule C1
6        "squid:S106"
7     })
8     public static void aMethod() {
9        // ...
10    }
11
12    public static void anotherMethod(String s1, String s2) {
13       // the following comment suppresses detection of the rule on the line where
            ↪ NOSONAR appears
14       if (s1 == s2) { // NOSONAR false-positive: Compare Objects With Equals
15    }
16 }
```

Listing 3.10. Examples of annotations used to suppress detection in SonarQube. SpongeBugs ignores them.

We also found two fix suggestions corresponding to a violation of rule C7 (`entrySet()` should be iterated when both key and value are needed) in project SonarQube that are spuriously reported by SpongeBugs. Listing 3.11 shows the corresponding code, where there is an important difference between an iteration over `keySet()` and one over `entrySet()`. Since a `TreeSet` is created passing `Map prop` as keys, the map is directly used as underlying implementation of the set of keys. An enumeration over `keySet()` will then follow the ordering defined over keys—alphabetical order in this case; in contrast, an enumeration over `entrySet()` may list the elements in a different order (the one in which they are stored in the map). Since method `writeGlobalSettings` produces user output, alphabetical order is expected and should not be changed.

```
private void writeGlobalSettings(BufferedWriter fileWriter) throws IOException {
   fileWriter.append("Global server settings:\n");
   Map<String, String> props = globalServerSettings.properties();
   for (String prop : new TreeSet<>(props.keySet())) {
    dumpPropIfNotSensitive(fileWriter, prop, props.get(prop));
   }
```

[4]Curiously, SonarQube includes a rule (S1291), which checks that `NOSONAR` annotations are *not* used. This rule is, however, disabled by default.

```
7  }
```

Listing 3.11. A *spurious* violation of rule C7 on line 4 reported by SpongeBugs.

The remaining 11 cases of false positives correspond to spurious violations of rule C1 (String literals should not be duplicated) where string literals occur inside Java *annotations*. Listing 3.10 shows an example of spurious fix generated by SpongeBugs. The fix does not alter program behavior in any way; it is just not idiomatic since it mixes program code (a static field) and annotations (which should only be used by the compiler).

```
1  public class JCDIServiceIMPL {
2  + private static final String UNCHECKED = "unchecked";
3
4  - @SuppressWarnings("unchecked")
5  + @SuppressWarnings(UNCHECKED)
6     @Override
7     public <T> void injectEJBInstance(JCDIInjectionContext<T> injectionCtx) {
8        JCDIInjectionContextImpl<T> injectionCtxImpl = (JCDIInjectionContextImpl<T>)
              ↪ injectionCtx;
9        // ...
10    }
11 }
```

Listing 3.12. Fix suggestion generated by SpongeBugs for a spurious violation of rule C1 on line 4. SonarQube does not report this as a violation.

False Negatives

Table 3.9 reports SpongeBugs's false-negative rate in each project and for each rule. Overall, 15% of rule violations reported by SonarQube were not detected—and hence not fixed—by SpongeBugs. As we mentioned above, this significant false negative rate is not much detrimental to SpongeBugs's practical applicability, since the tool still provides thousands of useful, accurate suggestions for rule violations. As we discuss in Section 3.2, we designed SpongeBugs to make sure that, when it detects a rule violation, it has all the necessary information to produce a suitable fix suggestion. Therefore, part of the false negative are a consequence of design decisions to trade off some detection capability for additional precision; others are due to other limitations of SpongeBugs's implementation.

The percentage of false negatives changes considerably with different rules. In order to better understand SpongeBugs limitations in practice, the rest of this section presents several examples of false negatives—with at least one example per rule.

Local Analysis. SpongeBugs's analysis is strictly local to each method: if a method `m` calls another method `n`, `m`'s analysis has no information about `n`'s effects and results other than its local calling context. This limitation may cause false negatives in all rules.

Listing 3.13 shows a violation of rule C4 (Parsing should be used to convert strings to primitive types) that is detected by SonarQube but is not fixed by SpongeBugs. The latter's analysis of line 5 is oblivious to the fact that method `getStatementTimeout` returns a `String`.

Table 3.9. For each project and each rule checked by SonarQube, the table reports the number of false negatives (a rule violation reported by SonarQube for which SpongeBugs provides no suggestions). The bottom rows summarize the TOTAL number of false negatives, and what percentage of the overall violations reported by SonarQube these false negatives correspond to; similarly the rightmost column reports the TOTAL per project.

PROJECT	B1	B2	C1	C2	C3	C4	C5	C6	C7	C8	C9	TOTAL
Eclipse IDE	36	3	15	0	1	15	13	6	–	44	5	138
SonarQube	–	–	10	–	–	–	0	–	–	–	–	10
SpotBugs	4	1	42	1	0	10	0	–	–	4	–	62
atomix	0	–	0	–	–	–	0	–	–	1	1	2
Ant Media Server	–	–	0	1	1	1	0	3	–	2	0	8
database-rider	–	–	0	0	–	–	0	–	0	0	–	0
ddf	1	–	6	–	0	–	2	–	0	8	0	17
DependencyCheck	–	–	10	1	–	–	0	–	–	2	–	13
keanu	0	–	–	–	–	–	0	–	0	0	–	0
mssql-jdbc	3	–	32	13	–	7	0	2	–	0	–	57
Payara	3	–	108	45	47	104	203	112	6	137	8	773
primefaces	–	–	50	2	0	0	7	–	0	1	0	60
TOTAL #	47	4	273	63	49	137	223	123	6	199	14	1,140
OVERALL %	52%	29%	9%	24%	69%	88%	8%	55%	9%	29%	8%	15%

Without this information, SpongeBugs cannot detect that rule C4 is being violated by passing a string to `valueOf` instead of `parseInt`.

```
1  public String getStatementTimeout() {
2     return spec.getDetail(DataSourceSpec.STATEMENTTIMEOUT);
3  }
4  // ...
5  int statementTimeout = Integer.valueOf(getStatementTimeout());
```

Listing 3.13. Violation of rule C4 on line 5, which is not detected by SpongeBugs.

It may seem that providing SpongeBugs with the information that is needed to detect violations such as the one in Listing 3.13 is straightforward: after all, method `getStatementTimeout` is defined in the same class as where the violation occurs. In our experiments, however, most of the false negatives due to local analysis involve a method that is called in one class but is defined in a different class. Listing 3.14 shows another violation of rule C4 that SpongeBugs does not detect. In this case, the offending method `getSecurityEnabled` is called in class `IIOPSSLSocketFactory` but is declared more specifically an interface, other than the one where the violation is present.

```java
 1  public interface IiopListener {
 2     String getSecurityEnabled();
 3  }
 4
 5  public class IIOPSSLSocketFactory {
 6     public void aMethod() {
 7        for (IiopListener listener : iiopListeners) {
 8           boolean securityEnabled = Boolean.valueOf(listener.getSecurityEnabled());
 9        }
10     }
11  }
```

Listing 3.14. Violation of rule C4 on line 8, which is not detected by SpongeBugs.

Extending SpongeBugs's analysis beyond purely local would require to process multiple files at once, and to collect more detailed typing information. While such an extension is beyond SpongeBugs's current design—which privileges simplicity and precision over broader applicability—we may consider it in future work. In order to do it efficiently, we may preprocess the whole codebase at once [100]; then, each individual analysis could access this system-wide information as needed in an efficient way. To be truly system-wide, this approach would also need to track dependencies outside a project's source code—such as in calls to pre-compiled or even native libraries.

Rule Restrictions. SpongeBugs analyzes some rules with additional restrictions on their applicability. Some of these restrictions are deliberate design choices that help make detection more precise or more efficient; others are limitations of the current implementation.

Take for example to rule C3 (Strings should not be concatenated using + in a loop), which SpongeBugs checks only when the concatenated string is eventually returned by a method. Thus, SpongeBugs misses the violation of rule C3 on line 7 in Listing 3.15 because string `containerStyleClass`, which is built by concatenation in a loop, is not returned by method `encodeElements`.

```java
 1  protected void encodeElements(Menu menu, List<MenuElement> elements) {
 2     boolean toggleable = menu.isToggleable();
 3     for (MenuElement element : elements) {
 4        String containerStyleClass = menuItem.getContainerStyleClass();
 5
 6        if (toggeable) {
 7           containerStyleClass = containerStyleClass + " " + Menu.SUBMENU_CHILD_CLASS;
 8        }
 9     }
10  }
```

Listing 3.15. Violation of rule C3 on line 7, which is not detected by SpongeBugs.

While it is possible to alleviate this restriction, we found that to fix more cases, not only the runtime performance would increase, but, more importantly, also the number of false positives. Thus, we introduced this restriction on rule C3 because it curbs the number of rule violations that are detected while still covering the most salient cases. Other restrictions mainly simplify the analysis, helping ensure that a rule violation's can be detected correctly.

For example, SpongeBugs only checks rule C8 (`Collection.isEmpty()` should be used to test for emptiness) inside regular methods. Thus, it misses the violation in Listing 3.16 because it appears inside a lambda expression, which makes the analysis of local variables considerably more difficult.

```
1  public class LeaderElectorProxy {
2     private final Map<String, Set<LeadershipEventListener<byte[]>>> topicListeners =
         ↪ Maps.newConcurrentMap();
3
4     public synchronized CompletableFuture<Void> removeListener(String topic,
         ↪ Listener<btye[] listener>) {
5        if (!topicListeners.isEmpty()) {
6           topicListeners.computeIfPresent(topic, (t, s) -> {
7              s.remove(listener);
8              return s.size() == 0 ? null : s;
9           });
10       }
11       // ...
12    }
13 }
```

Listing 3.16. Violation of rule C8 on line 8, which is not detected by SpongeBugs.

Another restriction in the application of rule C8 follows from SpongeBugs's limited information about how types are related by inheritance. Listing 3.17 shows a violation of this rule on line 5: `CopyOnWriteArrayList` implements the `List` interface, and hence the conditional on line 5 should be expressed as `!Collections.isEmpty()`. However, SpongeBugs only knows about the most common implementations of collection classes, and hence it misses this violation. This restriction also affects the other rules that involve `Collection` classes, that is rules C7 and C9.

```
1  import java.util.concurrent.CopyOnWriteArrayList;
2
3  protected CopyOnWriteArrayList<IPlayItem> items;
4  // ...
5  if (items.size > 0)
```

Listing 3.17. Violation of rule C8 on line 5, which is not detected by SpongeBugs.

Toolchain Limitations. Finally, a few false negatives follow from limitations of the tools we used to build SpongeBugs. In particular, Rascal's Java 8 grammar is not complete, and hence a few classes such as class `AMXConfigImpl` in project Payara cannot be parsed. While implementing SpongeBugs, we overcome some of these limitations by extending the Rascal grammar[5] to cover some of the missing cases.

Running Test Suites

A major concern when providing automatic fixes is whether the fixes preserve program behavior. A common way to evaluate program correctness is through executing its test

[5] Our extensions is now available in Rascal's repository.

suite [70, 142]. We report on the execution of test suites in the selected open source projects, to investigate if SpongeBugs introduces behavior faults.

We were able to run tests on 8 out of the 12 projects we evaluated SpongeBugs on. We did not run tests for projects database-rider and mssql-jdbc because they require a running database to execute tests. Despite our best efforts, we could not execute tests for projects Eclipse IDE and Ant Media Server, thus we also did not run their tests. Interestingly, they are the only 2 projects that strictly recommend building without running tests.[6] Projects ddf and Payara, which, in our experiments, builds consistently took longer than 20 minutes, recommend to skip tests only if speed up is desired.

As expected, SpongeBugs fixes mostly does not alter program behavior. Only a single test case failed when running the tests for the 8 remaining projects, as show in Table 3.10. The failing test is related to project SonarQube's false positive described in Listing 3.11.

Table 3.10. Summary of running test suites in all projects. The last column indicates the number of tests that fail after applying SpongeBugs suggestions.

PROJECT	TESTABLE?	FAILING TESTS
Eclipse IDE	No[a]	–
SonarQube	Yes	1
SpotBugs	Yes	0
Ant Media Server	No[a]	–
atomix	Yes	0
database-rider	No[b]	–
ddf	Yes	0
DependencyCheck	Yes	0
keanu	Yes	0
mssql-jdbc	No[b]	–
Payara	Yes	0
primefaces	Yes	0

[a] Recommends building without executing tests.
[b] Requires a database.

3.4.2 RQ2: Effectiveness and Acceptability

As discussed in Section Section 3.3.3, 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 Table 3.5, 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

[6]We collected documentation on the build process for 11 projects. These resources are available in our repository.

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 timely 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 **RQ2** (“Does SpongeBugs generate fixes that are acceptable?”), we submitted 38 pull requests containing 946 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. Table 3.5 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, 825 (87%) of all 946 submitted fixes were accepted; 797 (97%) 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: 87%), or one what was accepted without modifications (precision: 797/946 = 84%). 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 3.18 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 3.18. The lines added by this fix were flagged as not covered by any existing tests.

Sometimes a fix's context affects whether it is 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 3.19 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 3.19. 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 fixes, as the modification was in a line of code other than the one we fixed.

Commenting on the suggested fix in Listing 3.20, 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 3.20. 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 [13]. 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 3.21 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 3.21. 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 3.22 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 3.22. 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.

3.4.3 RQ3: Performance

To answer **RQ3** ("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{23} to measure how long our transformations take to run on the projects considered in Table 3.7. Table 3.11 show the performance outcomes. For each of the measurements in this section, we follow recommendations on measuring performance [71]: 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

Table 3.11. Descriptive statistics summarizing 5 repeated runs of SpongeBugs. Time is measured in minutes.

		RUNNING TIME	
PROJECT	FILES ANALYZED	MEAN	ST. DEV.
Eclipse IDE	5,282	102.3 m	2.31 m
SonarQube	3,876	25.3 m	0.84 m
SpotBugs	2,564	30.6 m	1.20 m
Ant Media Server	228	3.9 m	0.14 m
atomix	1,228	10.1 m	0.74 m
database-rider	109	0.8 m	0.04 m
ddf	2,316	28.6 m	1.55 m
DependencyCheck	245	5.5 m	0.21 m
keanu	445	2.9 m	0.09 m
mssql-jdbc	158	23.2 m	0.62 m
Payara	8,156	166.1 m	8.88 m
primefaces	1,080	15.5 m	1.45 m

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 ran our transformations on a subset of these larger files. As seen in Table 3.12, five larger files are responsible for more than 12 minutes (52%) of running time. Additionally, file `dtv` takes on average longer to run than `SQLServerConnection`; even though `dtv` has 1,600 less lines of code. File `dtv` has numerous class declarations and methods with more than 300 lines, containing multiple `switch`, `if`, and `try`/`catch` statements.

Table 3.12. Descriptive statistics summarizing 5 repeated runs of SpongeBugs on the 5 largest files in projects mssql-jdbc. Time is measured in seconds; size is given in non-blank non-comment lines of code LOC.

		RUNNING TIME	
FILE	LOC	MEAN	ST. DEV.
SQLServerConnection	4,428	202 s	14.1 s
SQLServerResultSet	3,858	158 s	22.9 s
dtv	2,823	208 s	17.8 s
SQLServerBulkCopy	2,529	86 s	5.4 s
SQLServerPreparedStatement	2,285	78 s	6.9 s

Generating some fix suggestions takes longer than others. We investigated this aspect more closely in SpotBugs, as it includes more than a thousand files, and contains 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. 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 Section 3.2.2). For example, step 1's filtering for rule B1 (Strings and boxed types should be compared using `equals()`), shown in Listing 3.23, 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 3.24, which is much more restrictive; as a result SpongeBugs runs in under 20 seconds for rule C9.

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

Listing 3.23. Violation textual pattern in the implementation of rule B1

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

Listing 3.24. 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.

The Impact of the Textual Pattern Matching Step

Our approach of looking for violations' textual patterns (step 1 in Section 3.2.2) might inadvertently skip cases in which SpongeBugs could devise a fix. Since we are matching strings, a minimal difference in layout–such as an extra blank space, or a line break—can be missed by step 1. We modified SpongeBugs' implementation by disabling step 1 and repeated our evaluation. Table 3.13 shows how many additional fixes were produced in comparison to running SpongeBugs with step 1. SpongeBugs produced only 18 additional fixes (0.24%). Listing 3.25 shows one example where step 1 fails to recognize a pattern for rule C5 (Strings literals should be placed on the left side when checking for equality). The reason is the blank space between the parenthesis of the method call and the argument.

```
if ( lookupName.equals( "java:comp/BeanManager" ) ) {
```

Listing 3.25. Warning pattern of rule C5 that step 1 misses.

The fixes all happened in project Payara, which is responsible for almost 60% of all 7,364 warnings detected by SonarQube. We also repeated the runtime performance measurements when step 1 is disabled. Table 3.14 shows that on average the runtime increased by 124%, increasing as high as almost 200% for projects SonarQube and keanu. Interestingly, project mssql-jdbc had the lowest increase (43%). This can be explained by its prevalence of large classes (as analyzed in Table 3.12), which means that most likely, the classes were already being analyzed by AST visiting (step 2). We can conclude that our choice of having a textual pattern matching step is well justified, as we judge the 0.24% increase on fixes is not worth having at the cost of runtime increase ranging from 43% to 196%.

Table 3.13. For each project and each rule checked by SonarQube, the table reports the number of additional fixes when running SpongeBugs *without* violation textual pattern (step 1). The bottom rows summarize the TOTAL number of *additional fixes*, and what percentage of the overall violations reported by SonarQube these additional fixes correspond to; similarly the rightmost column reports the TOTAL per project.

PROJECT	B1	B2	C1	C2	C3	C4	C5	C6	C7	C8	C9	TOTAL
Eclipse IDE	0	0	0	0	0	0	0	0	–	0	0	0
SonarQube	–	–	0	–	–	–	0	–	–	–	–	0
SpotBugs	0	0	0	0	0	0	0	–	–	0	–	0
atomix	0	–	0	–	–	–	0	–	–	0	0	0
Ant Media Server	–	–	0	0	0	0	0	0	–	0	0	0
database-rider	–	–	0	0	–	–	0	–	0	0	–	0
ddf	0	–	0	–	0	–	0	–	0	0	0	0
DependencyCheck	–	–	0	0	–	–	0	–	–	0	–	0
keanu	0	–	–	–	–	–	0	–	0	0	–	0
mssql-jdbc	0	–	0	0	–	0	0	0	–	0	–	0
Payara	0	–	**10**	**1**	0	0	**7**	0	0	0	0	**18**
primefaces	–	–	0	0	0	0	0	–	0	0	0	0
TOTAL #	0	0	10	1	0	0	7	0	0	0	0	18
OVERALL %	0%	0%	0.34%	0.38%	0%	0%	0.26%	0%	0%	0%	0%	0.24%

Table 3.14. Summary of 5 repeated runs of SpongeBugs with and without violation textual pattern filtering (step 1). Time is measured in minutes.

PROJECT	MEAN WITH STEP 1	MEAN W/O STEP 1	INCREASE %
Eclipse IDE	102.3 m	235.6 m	130 %
SonarQube	25.3 m	74.6 m	195 %
SpotBugs	30.6 m	86.8 m	184 %
Ant Media Server	3.9 m	8.4 m	115 %
atomix	10.1 m	28.9 m	187 %
database-rider	0.8 m	2.0 m	137 %
ddf	28.6 m	67.2 m	135 %
DependencyCheck	5.5 m	11.3 m	107 %
keanu	2.9 m	8.5 m	196 %
mssql-jdbc	23.2 m	33.2 m	43 %
Payara	166.1 m	342.1 m	106 %
primefaces	15.5 m	31.4 m	103 %
OVERALL	414.6 m	930.0 m	124 %

3.4.4 RQ4: Student Projects

To evaluate applicability and usefulness of SpongeBugs on code written by non-professionals, we ran additional experiments where we applied it to the 5 student projects presented in Section 18.

First of all, we ran SonarQube on these projects checking the usual 11 rules supported by SpongeBugs. Table 3.15 shows the results in terms of number of violations reported by SonarQube for every thousand lines of code. The same table also shows the same measure for the open-source projects used in the rest of the evaluation. Overall, rule violations occur with higher frequencies in the student projects than in the open-source projects—as it can be expected from code written by non-professional. Nonetheless, the student code generally is of high quality since its violations are not *much* higher. Remember that all projects but project 5 explicitly required students to check their projects with SonarQube and to modify the code to reduce the number of reported violations. The only project in which students were not required to use SonarQube is also the one with the largest number of violations per line of code, but the difference with projects of the same course is not big. Since all students knew SonarQube as a tool from previous courses, it is possible that they used it regardless of whether it was required by the project's specification, and that they generally paid attention to writing code that conforms with accepted coding guidelines.

Table 3.15 summarizes the results of applying SpongeBugs to the rule violations in student projects. For all violations but one (98% of all violations), SpongeBugs produced a correct fix suggestion that avoided the violation. At a high level, these results are comparable to those obtained on the open-source projects.

The results on student projects are consistent with those on open-source projects also in terms of which *rules* are most frequently violated and fixed. The most frequent violations are of rules C1, C5, and C8—in this order in both the student projects and in the open-source projects. On the other hand, student projects violated none of rules B1, B2, C4, and C9; in the open-source projects there were several violations of these rules, but they accounted for only about the 6% of all violations, and did not occur in all projects. Students ran SonarQube with a custom profile, which excluded some of the rules SpongeBugs checks; as a result even projects that were required to use SonarQube still incur some rule violations.

3.4.5 RQ5: Code with Behavioral Bugs

To answer RQ5, we ran SpongeBugs on all 438 bugs in Defects4J. Each experiment targets one bug and consists of two steps:

1. Run SpongeBugs on the *buggy* code, and apply all suggested fixes.
2. Run the *tests* associated with the code (which include at least one failing test) on the version with all SpongeBugs suggestions, and record which tests are passing or failing.

By running the test suite that comes with every bug, we can assess whether SpongeBugs's suggestions interfere with the intended program behavior; and, if they do, whether they improve or worsen correctness as captured by the tests.

Table 3.15. Top half of table: for each student project, its size LOC in non-blank non-comment lines of code and the number of violations of the 11 rules checked by SpongeBugs that are detected by SonarQube per thousands of lines of code (VIOLATIONS/KLOC). Bottom half of table: the same data about the open-source projects used in the rest of the evaluation.

PROJECT	LOC	VIOLATIONS / KLOC
Project 1	6.8 K	0.00
Project 2	5.2 K	4.23
Project 3	2.5 K	0.80
Project 4	2.4 K	4.60
Project 5	3.6 K	5.80
Eclipse IDE	743 K	1.03
SonarQube	500 K	0.22
SpotBugs	280 K	1.80
atomix	550 K	0.13
Ant Media Server	43 K	1.12
database-rider	21 K	0.67
ddf	2.5 M	0.01
DependencyCheck	182 K	0.43
keanu	145 K	0.15
mssql-jdbc	79 K	5.23
Payara	1.95 M	2.26
primefaces	310 K	2.23

Overall, SpongeBugs suggested 675 fix suggestions across all bugs in Defects4J; as usual, all suggestions compile without errors. In the overwhelming majority of cases, SpongeBugs's suggestions did *not* alter program behavior: for 22,253 out of 22,270 tests in Defects4J, tests that were previously passing were still passing, and tests that were previously failing were still failing. As expected, SpongeBugs did not fix any of the semantic bugs in Defects4J, which is what we expected since its rules do not target behavioral correctness.

The exceptions involved 17 tests in Defects4J that were originally passing (in the buggy version of each program) but turned into failing tests after we applied SpongeBugs's suggestions. In these cases, SpongeBugs altered program behavior in a way that is inconsistent with the intended one captured by the originally passing tests.

All these 17 cases involved spurious violations of rule B1 Strings and boxed types should be compared using `equals()`. We observed one similar case of spurious violation in SpongeBugs's detection of rule B1 in the experiments of Section 3.4.1, but the phenomenon is more

Table 3.16. For each student project and each rule, the table reports two numbers x/y: x is the number of warnings violating 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. For brevity, the table only reports the rules for which SonarQube found at least one violation in some project.

PROJECT	C1	C3	C5	C7	C8	TOTAL	FIXED %
Project 1	–	–	–	–	–	–	–
Project 2	18/18	–	–	–	4/4	22/22	100%
Project 3	–	–	2/2	–	–	2/2	100%
Project 4	2/2	–	8/8	1/1	–	11/11	100%
Project 5	8/8	1/0	3/3	2/2	7/7	21/20	95%
TOTAL	28/28	1/0	13/13	3/3	11/11	56/55	–
FIXED %	100%	0%	100%	100%	100%	98%	–

prominent in Defects4J. Let's outline these cases of spurious detection to better understand where SpongeBugs fails. In all cases, SonarQube reports the same spurious warnings.

Out of the 17 failing tests, 14 are from project Closure{24}—a JavaScript optimizing compiler written in Java.

SpongeBugs found 5 violations of rule B1 in the buggy project version included in Defects4J; all of these violations, reported by both SpongeBugs and SonarQube, are spurious.

The root cause of these is the nature of project Closure, which relies on sophisticated optimizations involving string manipulation. Reference equality == is used instead of object equality `equals()` as much as possible—when it is semantically correct—because it is faster. SpongeBugs's fix suggestions replace expressions like `s == t` with `s.equals(t)`; however, the latter implies the former *only if* `s` is not `null`. Thus, some of SpongeBugs's fix suggestions introduce a crash in test that exercise such code with null strings. The following example–which is the single responsible for the 14 failing tests—in project Closure is hard to miss, since the programmer explicitly documented their intention to use reference equality:

```
//yes, s1 != s2, not !s1.equals(s2)
 if (lastSourceFile != sourceFile)
```

The other 3 tests that became failing after applying SpongeBugs's suggestions are from project Lang—Apache's popular Commons Lang base library for Java.

SpongeBugs found 3 violations of rule B1 in the buggy project version included in Defects4J; all of these violations, reported by both SpongeBugs and SonarQube, are spurious.

The root cause of these is again the way in which `null` strings are handled. Project Lang's API for string uses defensive programming, and hence it generally supports `null` values instead of valid `String` objects. Take method `indexOfDifference(String s1, String s2)` of class `StringUtils`,[25] which returns the lowest index at which `s1` differs from `s2`. The method's JavaDoc documentation explicitly says that `s1`, `s2`, or both may be null; correspondingly, reference equality is generally used *before* object equality, so as to be able to reliably compare strings that are `null`. When SpongeBugs introduces changes like:

$$\texttt{if (str1 == str)} \longrightarrow \texttt{if (str1.equals(str2))}$$

the program will throw a `NullPointerException` whenever str1 is `null`.

Interestingly, the developers of Apache Commons modified method `indexOfDifference` in version 3.0 of the library so that it inputs two `CharacterSequence` objects instead of strings. Comparing `CharacterSequence` objects by reference is not considered an anti-pattern, and hence neither SonarQube nor SpongeBugs would flag this more recent version of the library. While we could not verify the actual intentions of the developers, it is possible that they did consider string comparisons using == something to be avoided whenever possible—thus partially vindicating SonarQube's and SpongeBugs's strict application of rule B1.

Another spurious violation of rule B1, in Apache Commons class `BooleanUtils`, follows a similar pattern. Method `toBoolean(String str)` accepts `null` as argument `str`, but SpongeBugs's fix suggestion would crash with a `NullPointerException` in this case:

$$\texttt{if (str1 == "true")} \longrightarrow \texttt{if (str1.equals("true"))}$$

Interestingly, if we combine SpongeBugs's suggestion for rule B1 with its suggestion for rule C5 (Strings literals should be placed on the left-hand side when checking for equality) the code reverts to handling the case `str == null` correctly:

$$\texttt{if (str1.equals("true"))} \longrightarrow \texttt{if ("true".equals(str1))}$$

This suggests that static analysis rules are sometimes not independent—and hence stylistic guidelines should be followed consistently.

The main conclusions we can draw from the experiments with Defects4J are as follows:

- As expected by its design, SpongeBugs cannot fix semantic (behavioral) bugs because it targets syntactic (stylistic) rules.
- By and large, SpongeBugs's fix suggestions do not alter program behavior in any unintended way.
- For programs following unusual conventions or particular implementation styles, the rules checked by SonarQube and SpongeBugs may sometimes misfire. Often, it is still possible to refactor the program so that it follows the intended behavior while also adhering to conventional stylistic rules.

3.4.6 Additional Findings

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 3.26: change string to character).

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

Listing 3.26. 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.

3.5 Limitations and Threats to Validity

Some of SpongeBugs's transformations may violate a project's stylistic guidelines [112]. As an example, project primefaces uses a rule[7] about the order of variable declarations within a class that requires that private constants (`private static final`) be defined after public constants. SpongeBugs's fixes for rule C1 (String literals should not be duplicated) may violate

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

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.[8] 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 [142]. 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 [114]. 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.

3.6 Conclusions: SpongeBugs

In this chapter we introduced a new approach and a tool (SpongeBugs) that finds and repairs violations of rules checked by static code analysis tools such as SonarQube, FindBugs, and SpotBugs. We designed SpongeBugs to deal with rule violations that are frequently fixed in both private and open-source projects. We assessed SpongeBugs by running it on 12 popular open source projects, and submitted a large portion (total of 946) of the fixes it generated as pull requests in the projects. Overall, project maintainers accepted 825 (87%) of those fixes—most of them (97%) without any modifications. A manual analysis also confirmed that SpongeBugs is very accurate, as only a tiny fraction of all its fix suggestions can be

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

classified as false positives. We also assessed SpongeBugs's performance, showing that it scales to large projects (under 10 minutes on projects as large as half a million LOC); and its applicability to student code and to the Defects4J curated collection of bugs. Overall, the results suggest that SpongeBugs can be an effective approach to help programmers fix warnings issued by static code analysis tools—thus contributing to increasing the usability of these tools and, in turn, the overall quality of software systems.

Artifacts: the complete artifacts to support the replication of our experiments are available: `https://github.com/dvmarcilio/spongebugs`.

Part III

Analyzing Exception Behavior

4 How Java Programmers Test Exception Behavior

Exceptions often signal faulty or undesired behavior; hence, high-quality test suites should also target exception behavior. This chapter reports on a large-scale study of *exception tests*—which exercise exception behavior—in 1 157 open-source Java projects hosted on GitHub. We analyzed JUnit exception tests to understand what kinds of exceptions are more frequently tested, what coding patterns are used, and how features of a project, such as its size and number of contributors, correlate to the characteristics of its exception tests. We found that exception tests are only 13% of all tests, but tend to be larger than other tests on average; unchecked exceptions are tested twice as frequently as checked ones; 42% of all exception tests use `try`/`catch` blocks and usually are larger than those using other idioms; and bigger projects with more contributors tend to have more exception tests written using different styles.

Structure of the Chapter

- Section 4.1 provides motivation for this chapter.
- Section 4.2 contextualizes the reader with background concepts, including the exception tests coding patterns.
- Section 4.3 describes our research questions, project selection, and analysis process.
- Section 4.4 presents our results and findings.
- Section 4.5 presents the threats that affect the validity of our work.
- Section 4.6 discusses the possible applications of our findings.
- Section 4.7 draws our conclusions.

4.1 Introduction

The importance of testing in software development has become conventional wisdom; yet, writing high-quality tests remains a challenging endeavor [5, 161]. Among all different kinds of tests that are written, in this chapter we focus on those that *exercise exception behavior*—or *exception tests* for short. Exception behavior is a frequent source of failures [74], and is often implicated in anti-patterns and misuses [4]; on the other hand, proper exception-handling code is a necessary component of robust, maintainable software [21, 133]. Therefore, testing exception behavior is critical in building comprehensive test suites. However, dealing with exceptions—including in tests[1]—can be tricky, because an exception behavior's control flow is intrinsically unstructured (an exception can propagate through the call stack) and it is easy to miss some "corner cases" of exception-inducing inputs [30]. Testing practices have been studied extensively, and exception behavior is an increasingly popular empirical research target, but the combination of the two topics—exception testing—has so far received little attention.

We contribute to narrowing this knowledge gap with a *large-scale empirical study of exception testing in Java*. We analyzed all exception tests we could detect written using any version of the JUnit framework in 1 157 open-source Java projects—including numerous widely-used frameworks maintained by Apache, Google, and Spring—comprising 1 123 846 tests. The main **findings** of this analysis, include:

- Exception tests are often included as part of the test-writing effort: 66% of projects with tests also include some exception tests, and 13% of all tests target exceptions. On average, an exception test is 110% the size (mean lines of code) of any test.
- Exception tests most frequently target Java's standard exception classes (over 2/3 of all exception tests), and unchecked exceptions (about twice as frequently as checked exceptions).
- A standard `try`/`catch` block is the most common way of writing an exception test, followed by JUnit's `@Test(expected=...)` annotation.
- Exception tests written using `try`/`catch` blocks tend to be the longest; those written using `@Test(expected=...)` tend to be the shortest.
- Larger projects with more contributors are more likely to include exception tests written in a variety of styles.

[1]A StackOverflow question{26} asking how to write exception tests in JUnit has over 1.3 million views and several answers from experts such as one of Mockito's core contributors and StackOverflow users with high reputation.

4.2 Background

4.2.1 Exceptions: What They Are For

Exceptions are used to signal that something went wrong during program execution. A program may include *exception handling* code, which executes when an exception is raised to try to recover from the error or at least mitigate it. Thus, at a high-level, exceptions can help improve program *robustness*.

In an object-oriented language like Java, exceptions are instances of some *exception classes*; different exception classes characterize different ways of using exceptions. In our analysis, we consider three orthogonal (and standard [72, 133, 140]) classifications according to *origin*, *kind*, and behavioral *usage*.

Origin an exception class's *origin* in a given project depends on where it is defined: in Java's standard libraries, local to the project's code base, or in an external library.

Kind according to its type,{27} an exception may be unchecked (a subtype of `RuntimeException` or `Error`) or checked (any other subtype of `Throwable`).{28}

Usage exceptions are used to signal three main different categories of program behavior [72, §4.4] [140, §12] [133, §8.4], which we refer to as *usage* failure, fault, and return. A failure is a low-level error that usually depends on an exception state of the execution environment; for instance, the program runs out of memory (`OutOfMemoryError`). A fault signals the violation of a program's expected behavior; for instance, an array is accessed with an invalid index (`ArrayIndexOutOfBoundsException`). Category return captures improper usages of exceptions—not to signal erroneous conditions but to propagate information outside of the language's structured control flow. For example, to "break out of a complex, nested control flow"{29} similarly to a goto; or to signal the end of a file (`java.io.EOFException`).

4.2.2 Exception Testing Patterns

An *exception test* is a test that may trigger exception behavior in the code it exercises. Based on the documentation of JUnit{30} and other testing libraries,{31} as well as on other empirical studies [43, 175, 188, 202], we identified five main coding *patterns* that programmers use to write exception tests. This section outlines them and discusses their comparative advantages and disadvantages. Figure 4.1 shows the patterns in code, Table 4.1 summarizes which libraries support which patterns, and Section 4.4.3 discusses their distribution in the analyzed projects.

Pattern try/catch

Pattern try/catch uses Java's built-in `try`/`catch` statements, and hence it does not require any library. Testing whether some `testing code` throws an exception amounts to setting up a `catch` block for an exception of the expected type. Pattern try/catch's main strength is its *flexibility*: since the test has to explicitly set up the exception-handling code and can include several `catch` blocks or multi-catch exception types, it can check any features of any number

Table 4.1. The libraries or framework where each coding PATTERN (for checking whether an exception was thrown) is supported (green check icon), deprecated (orange exclamation icon), or not supported (red X icon).

	plain	JUnit				assertion
PATTERN	Java	4.0	4.7	4.13	5.0	libraries
try/catch	✓	✓	✓	✓	✓	✓
test	✗	✓	✓	✓	✗	✗
rule	✗	✗	✓	!	✗	✗
assert	✗	✗	✗	✓	✓	✓
generic	✗	✗	✗	✗	✗	✓

```java
@Test
void tryCatch()
  throws Exception {
 Exception caught =
     ↪ null;
 try { /* testing
     ↪ code */ }
 catch (Exception e)
  { caught = e; }
 if (caught == null)
  fail(); // fail
 else pass(); // pass
}
```

(a) Pattern try/catch.

```java
@Test(expected =
   Exception.class)
void test()
  throws Exception {
 /* testing code */
}
```

(b) Pattern expect test.

```java
@Rule
ExpectedException ex;

@Test
void rule()
  throws Exception {
 ex =
   ExpectedException.none();
 ex.expect(Exception.class);
 ex.expectMessage("error");
 /* testing code */
}
```

(c) Pattern expect rule.

```java
@Test void assert()
  throws Exception {
 Exception ex =
  Assertions.assertThrows(
   Exception.class,
   ()->/* testing code */
  );
  assertEquals("error",
     ex.getMessage());
}
```

(d) Pattern assert throws.

```java
@Test void generic()
  throws Exception {
  assertThatThrownBy(
    ()->/* testing code */
  ).isInstanceOf(
      Exception.class)
   .hasMessage("error");
}
```

(e) Pattern generic assertion.

Figure 4.1. The main coding patterns that programmers can use to test for exception behavior in Java.

of thrown exceptions, at any point during the execution of the test, and it can even check that a certain exception is *not* thrown (by failing inside a `catch` block). On the flip side, the required boilerplate code may result in tests that are *verbose*.

Pattern test

Version 4 of JUnit (released in 2006)[2] was the first providing a custom feature to write exception tests: by adding a parameter `expected` to the `@Test` annotation that marks JUnit tests, programmers can specify which tests are expected to throw which exceptions. Pattern test can make exception tests very *concise* and *readable*. However, it has limited flexibility: it is impossible to express which *part* of the `testing code` is expected to throw an exception, to test for *multiple* exception types, or to specify any *attributes* of the thrown exceptions other than their type(for instance, we cannot inspect *messages*).

Pattern rule

Version 4.7 of JUnit (released in 2009) introduced a new way of writing exception tests, using a field of type `ExpectedException` marked with annotation `@Rule`. Any test can set up such field to declare that the `testing code` is expected to throw an exception of a certain type. Pattern rule is somewhat more flexible than pattern test, since we can specify *attributes* of the expected exception other than its type (for example, with method `expectMessage`, its *message*). It can also designate that a *specific* statement of the `testing code` should throw an exception: this is the statement immediately following the calls to methods of class `ExpectedException`. However, patterns test and rule share the limitations that all code in `testing code` following the statement that throws the first exception will be ignored, and that they cannot specify *multiple* exception types in the same testing method. Tests written according to pattern rule remain *concise* but are stylistically quite different from JUnit's run-of-the-mill idioms that usually assert the expected outcome *after* the testing code rather than before it. This may be the reason why this pattern was removed from JUnit as soon as `assertThrows` became available.

Pattern assert throws

Static assertion method `assertThrows` was first introduced in JUnit 5.0 (released in 2017) and then added to JUnit 4.13 in 2020. Since `assertThrows` inputs the `testing code` as a *lambda*, this pattern is expressible only since Java 8. Pattern assert finally combines *conciseness* and *flexibility*, since `assertThrows` also returns the thrown exception object, which can be further inspected in the test code. It also blends with the other *assertion* methods available in JUnit, and with the test idioms they support. Pattern assert can be considered the *recommended* style to write exception tests since JUnit 5.0 (which no longer supports[3] patterns test and rule).

[2]We report release dates of stable releases, even though sometimes a beta release was made available to the public earlier on.

[3]Backward compatibility is still possible through JUnit's module `Vintage`.

Pattern generic assertion

Assertion libraries—such as Hamcrest, AssertJ, and Truth—provide flexible APIs to express all sorts of expected behavior—including exception behavior. Only AssertJ (used in Figure 4.1e) among these libraries includes methods such as `assertThatThrownBy` that implicitly catch any exceptions thrown by `testing code`; the other libraries offer methods to specify properties of exception objects but still rely on Java's `catch` blocks or JUnit's `assertThrows` method to perform the actual catching. As its name suggests, pattern generic is the most *flexible* approach to writing exception tests. It is easy to *combine* assertion methods in chains of method calls using the so-called "fluent" style, making it easier and more *readable* to write complex tests [202]. This structure also helps *readability*, supports powerful auto-completion *suggestions* when used within an IDE, and automatically generates informative *error messages* whenever an assertion fails. Besides the dependency on an additional library, the main *disadvantages* of using assertion libraries may come from their great flexibility: when the same behavior constraint can be expressed in several different ways, it is harder to enforce a *consistent* style within a project, and more *refactoring* and debugging effort may be needed.

4.3 Study Design

4.3.1 Research Questions

Our overall goal is understanding how Java developers *test for exception behavior*. To this end, we consider three main research questions:

RQ1: How often is exception behavior tested?

The first research question investigates how usual exception testing is in Java projects—both in absolute terms and relative to testing in general.

RQ2: What kind of exception behavior is tested?

The second research question looks for trends in the kinds of exception classes that feature more or less frequently in exception tests, and how these affect other characteristics of the tests such as their size.

RQ3: What coding patterns are used for exception testing?

The third research question analyzes how exception tests are written, and how they use the features of the available testing frameworks.

4.3.2 Project Selection

We started from [148]'s list of 2 672 Java projects—those with the most stars hosted on GitHub between November and December 2019. GitHub stars indicate a project's popularity and are commonly used to select source code of consistent quality [37, 123, 148, 154, 204]. Given our focus on tests, we further discarded projects that 1. have no detectable JUnit tests[4] (1 128 projects), or 2. have only the two tests that Android Studio IDE generates automatically by default{32} (374 projects). The second criterion is relevant because the Android Studio IDE makes it easy to generate two example tests for Android projects [18], which both include exception patterns. Therefore we exclude projects that do not add any tests beyond those automatically generated. This leaves 1 157 Java projects with some non-trivial tests, which are the focus of our analysis.[5] The final project selection includes large frameworks and applications, as well as projects in different domains. The filtering criteria indicate these are projects of good quality whose developers have devoted at least some effort to writing JUnit tests (a median of 91 commits of test code per project).

4.3.3 Analysis Process

We built JUnitScrambler: a tool to extract data about exception tests in Java projects. The tool works in three steps:

- **build**: the project from its sources
- **discover**: the test code in the project
- **analyze**: the discovered tests for exception testing patterns

Step *build* looks for recipes for Java's most popular build systems—Maven, Gradle, and Ant—and uses them to compile the project and its tests. It also tries to detect the required Java version and all external dependencies. Automatically looking for this information in build recipes may fail, or a project may not use a build script that we recognize.

If a project is built successfully, step *discover* uses JUnit 5's test discovery API{34}—which can process JUnit tests in any versions, including advanced features of JUnit 5 and non trivial test hierarchies—to find tests. JUnit's test discovery may work even when the build was incomplete; in addition, our tool looks for references to executed testing classes among the output of the build process. As a last resort, our tool scans every Java source file in the project, and marks as "possible tests" those that import testing libraries [18]. The combination of these three ways of looking for tests allows step "discover" to detect tests from all JUnit versions, including tests that may not be trivially found (e.g., located in directories other than a build system's default or generated by the build process).

Step *analyze* parses all discovered tests with JavaParser,{35} and processes the resulting abstract syntax tree and typing information to measure the characteristics of exception tests that we mention in Section 4.3.3. JavaParser's rich information—augmented with the dependencies collected by the build step— supports a fine-grained analysis of testing patterns

[4]The restriction to JUnit is justified by its popularity: we found a mere 6 projects out of 2 672 with tests written exclusively for the TestNG framework.

[5]We also ascertained that the patterns identified in Section 4.2.2 cover most of the exception tests: for example, we found only 5 projects with exception tests using library `catch-exception`{33}, which is not covered by the patterns.

and exception types. Among other things, we distinguish between usages of `assertThat` from various testing libraries (JUnit, Hamcrest, AssertJ, and Truth), can follow nested calls in test methods, and can often determine whether exception classes from external libraries are checked or unchecked.

Measured data

For every *project*, JUnitScrambler reports its build system, JUnit version, whether any tests were found, and the list of classes with testing code. For every test (that is, testing method), it measures its size in non-blank lines of code (LOC), and if it detected any exception testing coding patterns (Section 4.2.2). For every exception test (that is, when a pattern was found), it reports the detailed structure of the pattern, whether the test asserts on an exception message or cause, and the fully-qualified exception static types of the exceptions mentioned in the test (which determine their *origin* and *kind*, see Section 4.2.1), and other contextual information such as any messages in the assertions or code comments.

JUnitScrambler records the raw measured data in CSV format. We then imported the data into R[36] and used it to perform the statistical analysis reported in Section 4.4.

Qualitative analysis

A "Closer Look" section complements the quantitative analysis with qualitative findings about each research question, which we obtained by systematically inspecting the top-10 projects with the "most conspicuous" characteristics relevant to the question. For example, RQ1's closer look inspects projects with "the largest number of exception tests" and with "few exception tests".

4.4 Results

Before we delve into the details of exception testing, let's overview some overall characteristics of the projects we considered. Table 4.2 summarizes the main data.

Table 4.2. The number # and percentage % of ALL analyzed projects that use each BUILD SYSTEM. Percentages do not add up to 100% because a few projects use multiple build systems.

		BUILD SYSTEM			
	ALL	MAVEN	GRADLE	ANT	NA
#	1 157	521	464	39	172
%	100.0	45.0	40.1	3.4	14.9

Overall, we analyzed 1 157 projects with tests. Most projects use one of three build systems: Maven is the most widely used (45.0% of projects), followed by Gradle (40.1% of projects), whereas only 3.4% of projects use Ant, and 14.9% of projects use no build system that we could detect (NA in Table 4.2).

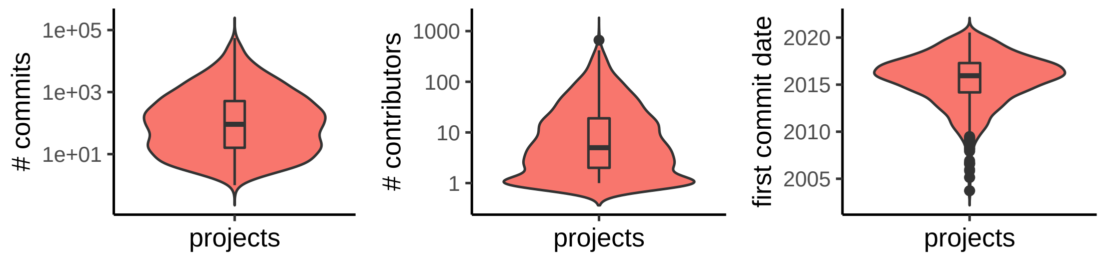


Figure 4.2. Violin plots of the analyzed projects' total number of commits, number of contributors, and initial commit date of their test code. Vertical scale is logarithmic in first two plots.

Figure 4.2 displays other overall characteristics of the test code among the 1 157 projects that we analyzed. The total number of *commits* varies widely among projects: its median is 91, its mean is 1 094, and its maximum 55 090 commits. The number of *contributors* also varies widely: its median is 5, its mean is 22, and its maximum is 663 contributors. The *age* of the test code, measured as the date of test code's first commit, is less spread out: its median is 2015-12-12, close to its mean 2015-07-08; nonetheless there are several outlier older projects: the oldest commit date is more than 17 years ago (2003-09-26).

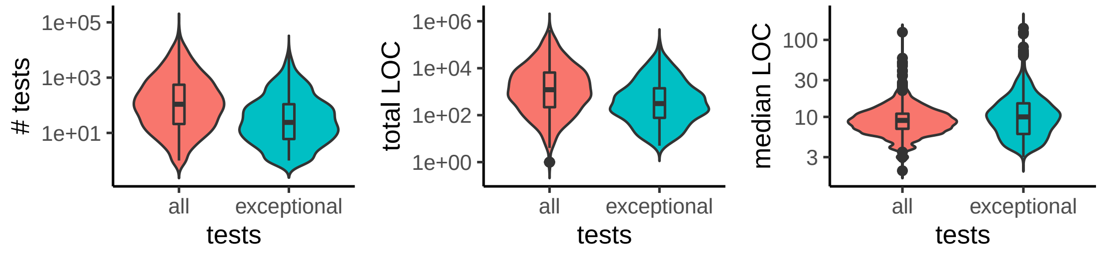
4.4.1 RQ1: How often is exception behavior tested?

RQ1 asks how much exception testing is usually carried out in Java projects. As shown in Table 4.3, 66.2% of the projects with *some* tests also include *exception* tests. The split between exception and regular tests is, however, not even: only 13.2% of all tests target exceptions—making up 14.6% of all lines of testing code. On the other hand, there is a strong positive correlation (Kendall's $\tau = 0.7$) between number of tests and number of exception tests that each project includes, which indicates that exception tests are an integral part of the test-writing effort in the analyzed projects.

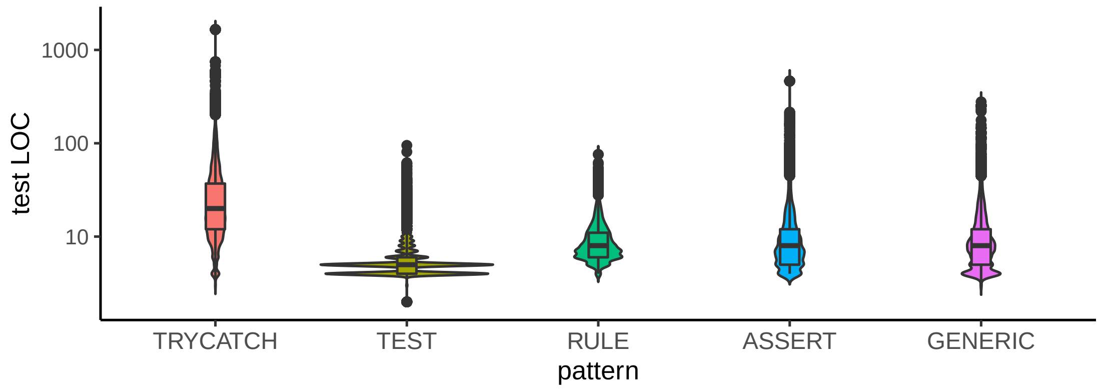
Table 4.3. Number of PROJECTS with some tests, number of testing METHODS and CLASSES, total $\sum$ LOC and per-method mean $\overline{\text{LOC}}$ size of test methods in lines of code. The first row comprises ALL TESTS, the second only EXCEPTION TESTS, and the third the latter as a percentage of the former.

	#PROJECTS	#METHODS	#CLASSES	$\sum$ LOC	$\overline{\text{LOC}}$
ALL TESTS	1 157	1 123 846	171 011	14 023 852	13
EXCEPTION TESTS	766	148 063	41 537	2 046 930	14
% EXCEPTION TESTS/ALL	66.2	13.2	24.3	14.6	110.3

The violin plots in Figure 4.3a provide more information about the effort that is usually devoted to writing exception tests. By comparing the two shapes in the leftmost plot, we notice that the distribution of number of exception tests is wider around the median. Thus, there is less inter-project variability in the number of exception tests compared to all tests.



(a) Analyzed projects' total number of tests, total size of all tests, and median size of a test in lines of code. Each plot shows data about *all* tests next to data about *exception* tests.



(b) Violin plot of the analyzed exception tests' size in LOC grouped by the patterns they use.

Figure 4.3. Violin plots of: (4.3a) the projects' size measures; (4.3b) patterns used in their tests. Vertical scales are logarithmic.

A similar trend exists for the total size (in lines of code) of all tests compared to exception tests—even though the mean size of any exception test is 110.3% that of any test. Indeed, there is a small but definite positive correlation (Kendall's $\tau = 0.1$) between a project's number of tests and their median size, but a negligible correlation ($\tau = -0.02$) between a project's number of exception tests and their median size. Thus, exception tests tend to be more homogeneous in size across projects, indicating that writing exception tests is an activity that receives significant effort but is somewhat more "standardized" than writing tests in general.

Two thirds of the projects with tests also include exception tests; the latter vary less in number and size.

RQ1: A Closer Look at Some Projects

We observed that projects with the largest number of tests typically also have the largest number of exception tests. In particular, the project with the most tests (Eclipse Collections[37] with over 47 283 tests) is also the project with the most exception tests (7 839 tests). To produce such a huge number of tests the project uses *code generation*[38] which can automatically produce variants of tests for classes that have a similar behavior (e.g., they implement the same interface).

Effective large-scale testing (including exception testing) requires clear guidelines and effective practices. Project Apache Geode, for example, comprises tests in 5 different categories[39] (including unit, integration, and acceptance) and explicitly recommends how to catch exceptions in unit tests;[40] this might explain why it ranks 4th and 5th among our projects with the largest number of tests and exception tests. More generally, projects with the largest number of (exception) tests usually recommend providing unit tests when opening an issue or contributing code, and actively try to include tests with high code coverage (for example, project Hazelcast's[41] tests cover over 85% of all project code according to SonarCloud;[42] the project ranks 3rd and 8th among our projects with the largest number of tests and exception tests).

At the opposite end of the spectrum, projects with few (exception) tests tend to be younger, less established, and provide simpler, more limited functionality. Project Rest Countries,[43] for instance, offers a REST API exporting data about worldwide countries (e.g., their currency) to add internationalization support to web applications. Tutorials and extensive examples are another group of projects with a limited number of tests and exception tests. It is reasonable to expect that those projects that will undergo further development will also considerably extend their test suites as they mature; citing project Processing's documentation: "someday" they will have "hundreds of unit tests [. . .] but not today".[44]

4.4.2 RQ2: What kind of exception behavior is tested?

RQ2 asks what kind of exception behavior is most frequently tested in Java projects. The characteristics of the *exception classes* in exception tests are a proxy for such behavior.

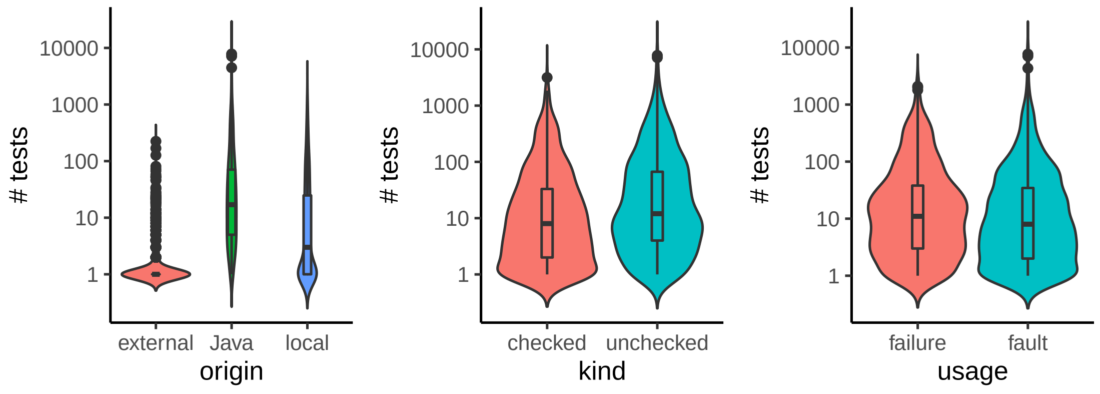
Categorization of exceptions

We classified the exception classes that we found in our projects according to their *origin*, *kind*, and *usage* (see Section 4.2). The classifications into origin and kind are objective and thus automatic. In contrast, an exception class's intended usage is described in the class's documentation and other artifacts where it features; therefore, it is somewhat informal and potentially subjective.

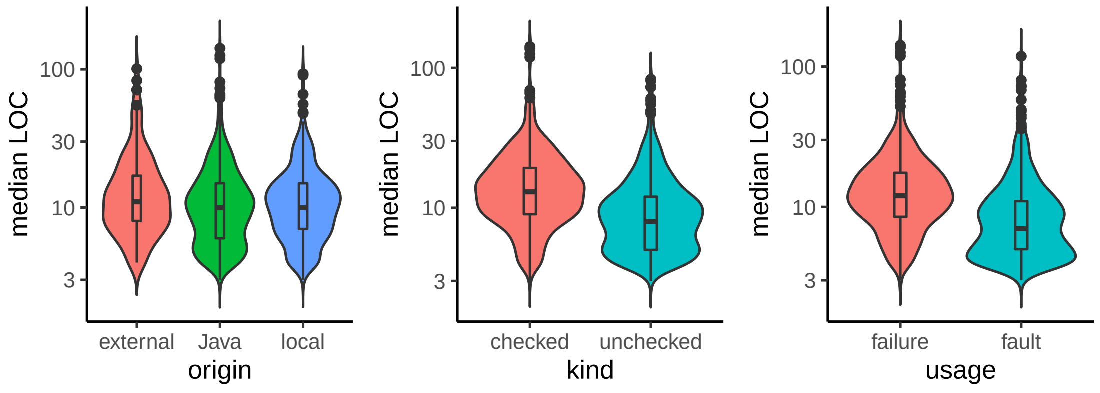
To manage this threat, we proceeded as customary in studies of Java exceptions classes [79, 95, 123, 151] and manually classified the usage of *only standard* Java exceptions—precisely, the same list of Hassan et al. [79]. Furthermore, we only considered categories failure and fault, since usage category return is most of the times sporadic and context-dependent—rather than being an exception class's intrinsic characteristic.

Table 4.4. Each column lists the percentage of EXCEPTIONS, TESTS, and PROJECTS that feature exception classes with certain characteristics: defined in Java's standard libraries, in a different external library, or locally to the project; checked or unchecked; used to signal failure or fault.

	ORIGIN			KIND		USAGE	
	Java	external	local	checked	unchecked	failure	fault
% EXCEPTIONS	6	12	82	40	60	74	25
% TESTS	76	2	26	36	70	47	57
% PROJECTS	95	32	59	81	90	89	83



(a) Number of tests targeting various exception classes



(b) Median size in LOC of tests targeting various exception classes

Figure 4.4. Violin plots of the analyzed projects' total number of tests and median size of a test. Each plot groups data according to various characteristics of the exceptions featured in tests: their *origin* (Java's standard libraries, external, or local to the project), their *kind* (checked or unchecked), and their *usage* (signaling failure or fault). Vertical scales are logarithmic.

The top data row in Table 4.4 summarizes the result of our classification. Out of 5 152 exception classes found in our tests: 1. 82% are project-local; 6% are Java standard exceptions, and the remaining 12% are from external dependencies; 2. 40% are checked types, and the remaining 60% are unchecked. Finally, according to our classification of their intended usage, 74% of all *Java* exceptions are for failures and 25% are for faults.[6]

Origin

Even though Java standard exceptions are only 6% of all tested exception classes, they are by far the most widely used: 76% of all exception tests target an exception of *origin* Java, and 95% of all projects include at least one such test.[7] This indicates that the familiar Java exception classes are tested extensively.

It may also suggest that the bulk of the exception behavior of most projects is not very project-specific—because the project defines no exception classes or tests them indiscriminately using abstract Java exception types. Still, 59% of all projects also test for locally defined exceptions; and about 32% of all projects also test for exceptions from external libraries. However, external exceptions feature in only 2% of the tests—and, indeed, the distribution of number of tests for each origin per project in Figure 4.4a says that most projects have no more than 10 tests targeting external exceptions. This suggests that the exception behavior of external libraries is seldom tested specifically—possibly because libraries mainly expose standard exceptions, or developers prefer testing abstract Java exception types when dealing with third-party code.

Kind

According to Java's official documentation,{45} checked exceptions should be used when the "client can reasonably be expected to recover from [the] exception". This guideline is somewhat informal, and as a result the role of checked vs. unchecked exceptions has long been a controversial point [37]. We found that projects test for unchecked exceptions (90% of all projects, and 70% of all tests) more frequently than for checked exceptions, but the latter still feature prominently—and nearly 72% of all projects include tests involving *both* checked and unchecked exceptions (not shown in Table 4.4). The minority of projects that only test for checked (10%) or unchecked (19%) do an overall limited amount of exception testing targeting at most a dozen exception classes. Since the compiler checks that programs include handling code for checked exceptions, *less* testing might be needed for checked exceptions thanks to these static checks. Nonetheless, we found no clear support for this expectation: distributions of the number of tests per project are qualitatively similar for checked and unchecked exceptions (Figure 4.4a); and exception tests targeting checked exceptions are usually considerably *larger* than those targeting unchecked exceptions (Figure 4.4b). In all,

[6]The missing 1% is a rounding error due to class `java.io.EOFException` that we consider of usage return and we discuss in Section 4.2 and below.

[7]These percentages don't add up to 100% because a test or project may target multiple exceptions of different origins.

the checked vs. unchecked debate is far from being settled, and in practice it seems programmers use any kind of exception classes without rigid rules.

Usage

Even though we classified 3 out of 4 Java exception classes as “usage failure”, projects test for both kinds of exception behavior—failure and fault—about as frequently, as confirmed by the qualitatively similar distributions in Figure 4.4a. However, the median method testing for exceptions signaling failure is nearly twice as big as one testing for fault (16 vs. 9 LOC). A fault indicates a bug in the program—which should never occur—whereas a failure is often due to a transient condition that may be recovered from. Therefore, a test that finds a fault might not have much to do besides signaling it to the programmer for debugging, and hence it is shorter than a test that finds a failure and may try to see if the same calls in different program states lead to different behavior.

Java exception `java.io.EOFException` is mainly used to “signal end of stream”; however, “many other input operations return a special value on end of stream rather than throwing an exception”,{46} which makes `EOFException` a class used primarily to pass an additional return value rather than to signal truly exception behavior. We inspected the tests targeting this exception in projects Apache Hadoop{47} and ExoPlayer{48} (two large projects among those that test for this exception) and confirmed that `EOFException` can be thrown as part of a program’s normal operation: the tests of both{49},{50} expect `EOFException` to be thrown.

Java standard exception classes are the most frequently tested; unchecked exceptions are tested more frequently than checked ones; exceptions signaling failure and fault are tested about as frequently.

RQ2: A Closer Look at Some Projects

Kind

Larger projects with many tests invariably target exceptions of both kinds. Project Spring Framework{51}—a widely-used Java framework—is an interesting example because it is designed so that it only throws unchecked exceptions.{52} Nonetheless, 18% of its 2 460 exception tests target 34 checked exception classes—including several local to the project. This confirms that it is practically impossible to stick to only one kind of exceptions, since the roles of checked and unchecked exceptions are irredeemably intermingled in Java.

Errors (subtypes of `java.lang.Error`) are a distinct category of unchecked exceptions reserved for “serious problems that a reasonable application should not try to catch”{53} such as `OutOfMemoryError`. Projects that primarily test for errors frequently deal with low-level features of system programming, such as virtual machines (Oracle GraalVM{54} and Eclipse OpenJ9{55}), core language features (Apache Commons Lang,{56} Google Guava,{57} and Apache Flink{58}) and language manipulation and translation (Google J2ObjC{59}). Dealing with low-level features, these projects’ tests handle errors to check robustness in different conditions of the runtime environment they execute in. A clear example is a test in Apache Flink[60] that tests the behavior of integer overflows but also includes an empty catch block for `OutOfMemoryError` with the comment "this may indeed happen in small test setups. We tolerate this".

Specific vs. generic exception

Exception-handling code should be exception-class specific,[61] whereas `catch` block with exception types high up in the inheritance hierarchy (such as `Exception` or even `Throwable`) are considered an anti-pattern. Since abstract types `Exception`, `Throwable`, and `RuntimeException` are among those featuring most frequently in exception tests (respectively, 2nd, 6th, and 10th), this anti-pattern may also occur in testing code. By manual analysis of a few larger projects, we found at least a couple of instances. Ten out of 12 of project Saturn's[62] test classes deal exclusively with type `Exception`; on closer inspection, these are *integration* tests, which need not differentiate between specific exception types (a task for unit tests) but just detect thrown exceptions [83]. Another instance is project Apache Flink, which includes a very long method[63] with 150 try/catch blocks all for type `Exception`. In this idiom, a specific exception type `Specific` is checked with an assertion that `e instanceof Specific` in the `catch` block.

Static analysis and tests

A key usage of exceptions is signaling runtime faults; however, some faults can also be detected statically by source-code analysis. Take a relatively basic but widespread bug: accessing a `null` reference. Java uses exception `NullPointerException` to signal "attempts to use `null` in a case where an object is required";[64] several static analyzers, such as Infer [26] and lgtm,[65] automatically detect such faults. We looked at 10 of the projects with the most tests that also use lgtm to detect and fix errors in their code base. All of them still include exception tests targeting `NullPointerException`, but the tests sometimes cover corner cases that are hard to catch using static analysis, or where using `null` is acceptable or even expected [37]. Project Apache ActiveMQ's[66] iterators, for example, throw a `NullPointerException` in some conditions when the iterator is no longer valid; in this scenario,[67] the exception doesn't signal a fault but rather returns information to the caller. Some parts[68] of Apache Geode also use `NullPointerException` for inter-method communication. A couple of exception tests[69] in Hibernate ORM[70] deal with using `null` to initialize object structures with circular references—which is notoriously tricky to analyze statically [141, 178]. In Apache Kafka,[71] a test[72] for `NullPointerException` captures an error that occurs when incorrectly nesting serializers—another scenario that is likely to be off-limits for common static analysis algorithms.

4.4.3 RQ3: What coding patterns are used for exception testing?

RQ3 analyzes the coding *patterns* (Section 4.2.2) that are used in exception tests, and relates them to other project features.

Table 4.5. Each column lists the percentage of exception TESTS and PROJECTS that use any of the 5 coding patterns try/catch, test, rule, assert, and generic (see Section 4.2.2).

	try/catch	test	rule	assert	generic
% TESTS	42	32	5	9	19
% PROJECTS	83	63	24	13	39

Table 4.5 shows that the most widely used pattern is try/catch, which features in 83% of all projects and 42% of all tests, followed by pattern test. In contrast, patterns rule and assert are the least frequently used—by 24% and 13% of all projects and in 5% and 9% of exception tests. Pattern rule's atypical syntax (see Section 4.2.2) may explain why it's not widely used.[8]

[8]Even one of the developers who built this mechanism into JUnit admits that he rarely uses it.[73]

Table 4.6. For each combination of patterns (those marked by ● in each column), the top row reports the percentage of all projects whose exception tests use exclusively that combination. Combinations not shown never occurred among the projects.

%	18.6	17.4	10.8	10.7	9.8	7.0	5.9	3.4	2.0	2.0	1.6	1.5	1.3	1.2	1.2	1.2	1.1	0.8	0.7	0.4	0.4	0.4	0.3	0.1	0.1	0.1	0.1
TRY/CATCH	●	●	●	○	●	●	●	●	●	●	●	●	●	○	○	○	○	●	○	●	○	○	○	●	○	●	○
TEST	○	●	●	●	●	○	●	○	○	●	●	●	○	○	●	●	○	○	○	●	●	○	●	○	○	○	●
RULE	○	○	○	○	●	○	●	○	○	●	○	○	●	●	●	○	○	●	○	●	●	○	●	●	●	●	●
ASSERT	○	○	○	○	○	○	○	●	●	●	●	●	○	○	○	○	○	○	●	●	○	●	●	●	○	●	●
GENERIC	○	○	●	○	●	●	○	●	○	●	○	●	○	○	○	●	●	●	○	○	●	●	○	○	●	●	●

Multiple patterns

The percentages in every row of Table 4.5 add up to more than 100%, because any one project or test may use more than one pattern. Mixing multiple patterns in the same *test* is not common: 94% of all tests stick to a single pattern; the remaining 6% typically combine patterns try/catch or assert with assertions using generic that look for more specific exception types. A handful of tests combine three patterns, but these are outliers that make up only 0.1% of all tests. In contrast, it is common that a *project* includes tests using different patterns (see Table 4.6), even though only 2% of projects include some tests for *every* pattern, and a substantial number of projects use a single pattern: 19% of projects only use pattern try/catch and 11% of projects only use pattern test.

To understand which characteristics of a project are associated with using more or fewer patterns, we fitted a Poisson[9] regression model using a project's number of different patterns as *outcome* variable, and the number of contributors, the number of tests,[10] and the testing code's age (measured by its earliest commit date) as *predictors*. The estimated slope coefficients for number of contributors and tests are positive with 95% probability; hence projects with more contributors and tests also tend to use more exception testing patterns.

Patterns and size

Figure 4.3b pictures the distribution of size (in LOC) of exception tests grouped according the patterns they use. It confirms the intuition that some patterns lead to more concise code than others. Pattern test is by far the most concise: tests using it are on average 6 lines, and their distribution is wider around and below the mean. Tests with pattern try/catch, in contrast, are on average 33 lines, and their distribution spreads over a wide range of lengths with many outliers. Pattern try/catch is the most flexible, doesn't depend on any external library, and is used in combination with other patterns; a large spread in test size is thus not surprising. Patterns assert and generic are also quite flexible, which explains the outlier tests that reach large sizes; on the other hand, their "natural" usage leads to concise tests (on average, around 12–13 lines long) which make up the bulk of the distribution.

To understand which characteristics of a test are associated with its size, we fitted a negative binomial[11] regression model using a test's LOC size as *outcome* variable, and, as *predictors*, the *patterns* it uses, the *origin* and *kind* of the exception classes it features (see Section 4.2.1), and whether it includes assertions on the *message* and *cause* of any exceptions; to control for overall project size, we also included the *total size* of the project's tests as a predictor. The results (see Table 4.7) indicate that all predictors are strongly associated with a test's size. The strongest effect is that of *patterns*: tests using pattern try/catch are the largest; those using patterns rule, assert, and generic are 36%, 39%, and 40% the size; and those with pattern test are the smallest at 18% the size. The association between size and the exceptions' *origin* is clear but less prominent: tests featuring Java standard exceptions

[9]Suitable for "counting" outcome variables [134].

[10]Using all tests or only exception tests lead to similar conclusions.

[11]A negative binomial generalizes a Poisson for outcome variables that are overdispersed [134], like test size in our case ($\mu = 24 \ll 1751 = \sigma^2$).

Table 4.7. Regression estimates of each characteristic's contribution to a test's size. More precisely, each number is the *exponential* of the estimated *slope coefficient* of the corresponding predictor's variable in a negative binomial regression with outcome test size. The exponential is taken to reverse the logarithmic link function, so that the shown estimates are on the outcome scale. All predictor variables except LOC (total size of test code in the project) are dummy selector variables ($\ell-1$ variables for a factor with ℓ possible values; the missing value thus corresponds implicitly to an estimated coefficient of $e^0 = 1$). All estimates are significant with 0.99 probability.

PATTERN				ORIGIN		KIND	INSPECTING		
TEST	RULE	ASSERT	GENERIC	JAVA	LOCAL	UNCHECKED	MSG	CAUSE	LOC
0.18	0.36	0.39	0.40	0.92	1.04	0.99	0.73	0.83	1.06

are 92% the size of those featuring external exceptions, whereas those featuring project-local exceptions are 104% the latter's size. Somewhat counterintuitively, tests inspecting exception messages or causes (columns MSG and CAUSE in Table 4.7) tend to be smaller than those not inspecting them. There is a positive association between a test suite's overall size (controlling for a confounding effect) and each exception test's average size but the effect is small in comparison with the others. Finally, tests targeting unchecked exceptions tend to be smaller than tests targeting checked exceptions, but the effect is small ($1\% = 100 - 99$ reduction in size). In all, the patterns capture different ways in which developers write tests trading off conciseness, expressiveness, and flexibility.

Patterns and checked/unchecked

Projects that only follow pattern try/catch often disproportionately use checked exception classes: on average, a project in this group includes 1.5 as many tests for checked exceptions than for unchecked, and 29% of these projects *only* test for checked exceptions; in contrast, projects that do *not only* use pattern try/catch include, on average, 0.4 fewer tests for checked exception than for unchecked, and just 4% of them only test for checked exceptions. Since checked exceptions must be either caught or explicitly propagated, a try/catch block is often necessary in exception tests targeting checked exception, which may make using other, more concise patterns redundant.

Projects that *exclusively* use pattern test in exception tests show the reverse tendency, namely they primarily test for unchecked exceptions: 78% of these projects *only* test for unchecked exceptions; in contrast, just 14% of projects that do *not only* use pattern test only test for unchecked exceptions. Pattern test is a natural choice to write concise exception tests; indeed, exception tests in projects using only this pattern are, on average, half the size of those in other projects.

Exception tests using `try`/`catch` blocks are the most common and longest; those using `@Test(expected)` are the second most common and shortest.

RQ3: A Closer Look at Some Projects

The largest projects that *exclusively* use pattern try/catch tend to be long-standing projects whose main development took place in the past and currently undergo only standard maintenance. Project Joda-Time,{74} for example, was a date-and-time library often used with older Java versions, but it is no longer maintained since its functionalities were made available in Java 8's package `java.time`. The project uses JUnit 3.8.2, which requires Java's try/catch to define exception tests.

Among the largest projects that exclusively use pattern test, Algorithms{75} (a collection of standard algorithms implementations) is a clear example of tests that privilege conciseness: 89% of its exception tests consist of a single call in the body, and no exception test's body is longer than 3 lines.

Project SonarQube{76}—a popular static analyzer for Java—is one of the largest projects among those that extensively use pattern rule, which features in nearly 70% of its exception tests. We found that this pattern coexists with others—most frequently, with generic assertions using AssertJ—to the extent that the same developer may write, on the same day,{77},{78} tests using both patterns: rule for simpler tests that mainly check that a certain exception is thrown; and AssertJ fluent assertions for "deeper" tests that inspect complex exception objects.

Project Apache Beam{79}—a framework for data-processing tasks—is among the largest projects that use *all* 5 exception test patterns. It is a clear example of how large projects with many contributors (Beam counts 390 contributors to its test code) naturally end up with a variety of different styles of exception testing code. Beam's class `DataFlowRunnerTest`'s Git history{80} is a microcosm of this dynamic. The class includes tests using patterns try/catch, rule, assert, and generic; different contributors (among the 19 that worked on this class) introduced tests using only one of these different patterns. In other words, each developer's preferred practice coexists with the others'.

The development history of Beam's `DataFlowRunnerTest` also shines light on the interplay between availability of JUnit features and how tests are written. When, in late 2014, part of this project was first written, developers added both tests using try/catch and using rule. However, those using rule didn't take full advantage of the pattern's expressive power until two years later, when developers added assertions on the exceptions' messages. The project formally switched to JUnit 4.13—supporting pattern assert—at the end of 2018; however, tests using the new pattern were added only months later, after a period during which maintainers were aware of the new pattern but also stuck to pattern rule for the time being.{81}

Pattern assert has been available for just a few years with recent versions of JUnit (see Section 4.2.2). The commit history of the largest projects that primarily use this pattern clearly show when the migration of older tests to use this new pattern took place. Project RoaringBitmap{82} (providing compressed bitsets) is the largest project using *only* pattern assert; in a large pull request that took place in April 2020,{83} the project migrated from JUnit 4 to 5, and updated all exception tests to use pattern assert. Maintainers of project Neo4j{84} (a popular graph database) planned the migration to JUnit 5 for over two years;{85} the migration is still ongoing,{86} but already 80% of the project's 1 671 exception tests use pattern assert.

Assertion frameworks such as AssertJ have supported fluent assertions, including for exception behavior, for years before JUnit 5 made them more widely available. Several larger projects that predominantly use pattern generic for exception tests started to use this pattern early on and often kept using it over JUnit 5's pattern assert even after migrating to the latest JUnit major version. Project Spring Boot Admin,{87} for example, only uses pattern generic, and chose to rewrite pattern test with AssertJ assertions when updating the project to JUnit 5;{88} project Spring Initializr{89} similarly rewrote pattern rule with AssertJ features instead of JUnit 5's assert.{90}

4.5 Limitations and Threats to Validity

Threats to *construct validity*—are we measuring the right things?—are limited given that we primarily measure well-defined features (size, types, and so on). The classification in exceptions according to their *usage* (see Section 4.2.1) is more delicate; to mitigate this threat, we limited it to well-known and well-documented Java standard exceptions [79], and one author reviewed the classification made by another one until agreement was reached.

We took great care to minimize threats to *internal validity*—are we measuring things right? Our tool JUnitScrambler (see Section 4.3.3) implements complementary strategies to extract useful information even from projects that are hard to build automatically without a custom environment: it parses build files to find dependencies and library versions; scans source code to detect test classes when JUnit's discovery process fails; and feeds any additional information to JavaParser to boost type resolution. Still, a few limitations remain: JUnitScrambler does not recognize some build systems (e.g., Bazel or Make); only processes JUnit tests; may detect an incorrect version of JUnit in projects with overly complex build processes; may miss some unusually complex combinations of fluent assertions or exceptions whose type JavaParser cannot reconstruct. We manually went through hundreds of projects and found these cases are rare—but there are a few more exception tests in the wild that don't feature in our analysis.

Threats to *external validity*—do the findings generalize?—mainly depend on the analyzed projects. The 1 157 projects we analyzed (selected as described in Section 4.3.2) are all open-source; it's possible that the exception tests of closed-source industrial projects have different characteristics. Nonetheless, our projects include plenty of commits of testing code, and span a wide range of size, maturity, and application domains—from widely used Java frameworks maintained by large development teams to single-author simpler mobile apps.

4.6 Applications of Findings and Future Work

Tested exceptions

A large portion (76%) of all tests we analyzed target Java standard exceptions, which are also prominent in web searches [79], StackOverflow posts [123], and mobile app bug reports [37, 64]. Unchecked exceptions `IllegalArgumentException`, `NullPointerException`, and `IllegalStateException` were among the most tested; the same classes are most frequently implicated in API misuses [196] and Android app bugs [37], and are among those with often insufficient documentation [97, 173, 207]. Thus, studying even more closely the tests featuring these exceptions in combination with the code that triggers them is an interesting direction for future work.

Messages and documentation

Undocumented exceptions (which may be thrown but are not mentioned by the documentation or signature [97]) are a common reason for uncaught exception bugs [37, 97, 173]. Exception tests could be a source of implicit documentation in such cases.

We found that 15% of exception tests detail the expected message (stored by exception objects); and 17% use assertions equipped with a string that describes what the assertion checks. Both kinds of messages are a form of documentation{91} that could be studied using natural language processing techniques [74, 207] to help debugging and to discriminate between correct and incorrect behavior [32, 207]. For example, one test{92} in Apache Hadoop clearly outlines which exceptions indicate which behavior (e.g., "setting empty name should fail"), and hence it would be a valid supplement to the tested method's documentation.{93} Such tests could also identify patterns of good testing practices and serve as a guide to writing better, more thorough exception tests.

Wrapping

Exception wrapping—a form of propagation—is when one exception is caught and wrapped by another exception [37]. Wrapping should not be used to hide errors (which should not be handled, see Section 4.4.2) within regular unchecked exceptions—which has been found may lead to crashes in Android apps [37].

We found 3% of all exception tests (in 19% of all projects) test on the *cause* of an exception, which indicates wrapping occurred. Such tests, although not common, may contain precious implicit information useful to debug an application's most complex exception behavior.

Exception testing patterns

The frequent usage of pattern try/catch has positive and negative implications. On the one hand, it is the only pattern that can be used with any JUnit and Java versions—including for complex scenarios.

On the other hand, using try/catch with JUnit 4 is considered a code smell [160, 175]. In fact, incorrect tests using try/catch introduced silent bugs in the test suites of projects using JUnit 4 [188]. It remains that, up to JUnit 5, the expressiveness of alternative patterns test and rule remained limited (e.g., test cannot assert on messages/causes, and rule's unusual syntax is somewhat controversial [188]). In all, it is not surprising that developers of popular open-source projects have been found [175] to often prefer good old try/catch over other patterns.

Introducing empty `catch` blocks is considered an anti-pattern, since it is associated with more defects [48, 59, 86, 204]. Nevertheless, our data suggests that it may be legitimate in testing code (as opposed to application code): empty `catch` blocks featured in about 50% of all exception tests using pattern try/catch; more than 50% of such blocks were accompanied by an informative comment (typically: `//expected`) which explains that idiom's purpose.

Despite its limited expressiveness, pattern test remains popular with developers—probably thanks to its conciseness—to the extent that a "vintage `@Test`"{94} JUnit 5 extension was introduced to write pattern test even with the latest JUnit versions (which no longer support it in the base library). Instead, other JUnit features (e.g., pattern assert) have failed to reach widespread adoption (e.g., 2/3 of projects that use JUnit versions supporting assert do not actually use it). Using our dataset and tool, we could perform a longitudinal study of how projects migrated tests from JUnit 4 to newer versions—extending the qualitative analysis of Section 4.4.3—to shed light on the interplay between available features and their adoption.

Finally, studying how AssertJ is used in combination with JUnit could deepen our understanding of testing practices. We found that 39% of all projects include exception tests using pattern generic—usually through libraries such as AssertJ. Indeed, AssertJ is growing in popularity [148, 202] since it provides expressive and concise idioms for testing all sorts of behaviour—including exceptions [175].

4.7 Conclusions

This chapter described a large-scale study of tests exercising exception behavior in Java projects. We found that such tests are an integral part of the project's test suites we analyzed, and that there is a considerable variety in the ways in which such tests are written and in the different exception behaviors that are exercised—especially in larger projects with many contributors.

Artifacts: the complete artifacts to support the replication of our experiments are available: `https://doi.org/10.6084/m9.figshare.13547561`.

5

Lightweight Precise Automatic Extraction of Exception Preconditions in Java Methods

When a method throws an exception—its *exception precondition*—is a crucial element of the method's documentation that clients should know to properly use it. Unfortunately, exceptional behavior is often poorly documented, and sensitive to changes in a project's implementation details that can be onerous to keep synchronized with the documentation.

We devised WIT, an automated technique that extracts the exception preconditions of Java methods and constructors. WIT uses static analysis to analyze the paths in a method's implementation that lead to throwing an exception. WIT's analysis is precise, in that it only reports exception preconditions that are correct and correspond to feasible exceptional behavior. It is also lightweight: it only needs the source code of the class (or classes) to be analyzed—without building or running the whole project. To this end, its design uses heuristics that give up some completeness (WIT cannot infer all exception preconditions) in exchange for precision and ease of applicability.

We ran WIT on the JDK and 46 Java projects, where it discovered 30 487 exception preconditions in 24 461 methods, taking less than two seconds per analyzed public method on average. A manual analysis of a significant sample of these exception preconditions confirmed that WIT is 100% precise, and demonstrated that it can often accurately and automatically document the exceptional behavior of Java methods.

Structure of the Chapter

- Section 5.1 provides motivation for this chapter.
- Section 5.2 showcases motivating examples for our approach.
- Section 5.3 describes how our technique works.
- Section 5.4 describes the design of our experimental evaluation.
- Section 5.5 describes the results of our experimental evaluation.

- Section 5.6 presents the threats that affect the validity of our work.
- Section 5.7 discusses applications of our technique.
- Section 5.8 draws our conclusions.

5.1 Introduction

To correctly use a method, we must know its *precondition*, which specifies the *valid* inputs: those that the method's implementation can handle correctly. In programming languages like Java, a method's implementation may throw an *exception* to signal that a call violates its precondition. If it does so, knowing the method's exceptional behavior is equivalent to knowing (the complement of) its precondition. Ideally, a method's exceptional behavior should be described in the method's documentation (for example, in its Javadoc comments) and thoroughly tested. In practice, it is known that a method's documentation can be incomplete or inconsistent with its implementation [150, 210], and that only a fraction of a project's test suite exercises exceptional behavior [126]. This ultimately limits the usability, in a broad sense, of insufficiently documented methods: without precisely knowing its precondition, programmers may have a hard time calling a method; test-case generation may generate invalid tests that violate the method's precondition; program analysis may have to explicitly follow the implementation of every called method, which does not scale since it is not modular.

To alleviate these problems, we present WIT (*What Is Thrown?*): a technique to automatically infer the *exception preconditions*—the input conditions under which an exception is thrown—of Java methods. As we discuss in Section 2.4.2, extracting preconditions and other kinds of specification from implementations is a broadly studied problem in software engineering (and, more generally, computer science). Our WIT approach is novel because it offers a distinct combination of features. First, WIT is *precise*: since it is based on static analysis, it reports preconditions only when it can determine with certainty that they are correct. It is also *lightweight*, as it is applicable to the source code of individual classes of a large project without requiring to build the project (or even to have access to all project dependencies), and can combine its analysis of multiple projects in a modular fashion.

A key assumption underlying WIT's design is that a significant fraction of a method's exceptional executions are usually simpler, shorter, and easier to identify than the other, normal, executions. Therefore, WIT's analysis (which we describe in detail in Section 5.3) relies on several heuristics that drastically limit the depth and complexity of the program paths it explores—for example, it bounds the length of paths and number of calls that it can follow. Whenever a heuristics fails, WIT gives up analyzing a certain path for exceptional behavior. In general, this limits the number of exception preconditions that WIT can reliably discover. However, if our underlying assumption holds, WIT can still be useful and effective, as well as lightweight and scalable.

We implemented WIT in a tool with the same name, which performs a lightweight static analysis of Java classes using JavaParser for parsing and the Z3 SMT solver for checking which program paths are feasible. Section 5.4 describes an experimental evaluation where we applied WIT to several modules of Java 11's JDK, and 46 Java projects—including several widely used libraries—to discover the exception preconditions of their public methods. WIT inferred 30 487 exception preconditions of 24 461 methods—running for 1.9 seconds on average on each of the 460 032 analyzed public methods.

A manual analysis of a significant random sample of the inferred preconditions confirmed that WIT is precise: all manually checked preconditions were correct. It also revealed that it could retrieve 9–83%[1] of all supported exception preconditions in project `Apache Commons IO`—achieving even higher recall on projects that use few currently unsupported Java features. Our empirical evaluation also indicates that WIT can be *useful* to programmers: 38% of the exception preconditions in the JDK's sample and 72% in the other projects' were not already properly documented; and 7 pull requests—extending the public documentation of open-source projects with a selection of WIT-inferred preconditions—were accepted by the projects' maintainers.

5.2 Showcase Examples of Using wit

We briefly present examples of applying WIT to detect the exception preconditions of library functions in two Apache projects: Dubbo{95} and Commons Lang.{96} The examples showcase WIT's capabilities and practical usefulness: WIT could automatically extract exception preconditions in many methods of these two projects, including some that were not documented (Section 5.2.1) or incorrectly documented (Section 5.2.2). Section 5.5.6 reports further empirical evidence that WIT's exception preconditions can be useful as a source of documentation.

To better gauge WIT's capabilities, let us stress that the two Apache projects discussed in this section are widely used Java libraries; for instance, Dubbo's GitHub repository{97} has over 24 thousand forks and 36 thousand stars. As a result, they are particularly well documented and tested [150, 207]. The fact that WIT could find some of their few missing or inconsistent pieces of their documentation indicates that it has the potential to be practically useful and widely applicable.

5.2.1 Missing Documentation

Listing 5.1 shows an excerpt of two overloaded implementations of method `bytes2base64`, which takes a byte array and represents it as a string in base 64. As we can see from the initial lines in `bytes2base64`'s second implementation, the two methods have fairly detailed preconditions; furthermore, since the first method calls the second with additional fixed argument values, the first's precondition is a special case of the second's.

Unfortunately, the documentation of these methods does not mention these preconditions: for example, the second method's Javadoc comment vaguely describes `off` and `len` as simply "offset" and "length", without clarifying that they should be non-negative values. This lack of documentation about valid inputs decreases the usability of the methods for users of the library.

[1]The range depends on which features and which output of WIT we consider; see Section 5.5.2 for all details.

```
 1  public static String bytes2base64(byte[] b, char[] code)
 2  { return bytes2base64(b, 0, b.length, code); }
 3
 4  public static String bytes2base64(final byte[] bs, final int off, final int len, final
        ↪ char[] code) {
 5    if (off < 0) throw new IndexOutOfBoundsException();
 6    if (len < 0) throw new IndexOutOfBoundsException();
 7    if (off + len > bs.length) throw new IndexOutOfBoundsException();
 8    if (code.length < 64) throw new IllegalArgumentException();
 9    //...
10  }
```

Listing 5.1. Excerpts of the implementation of two methods in Apache Dubbo's class `Bytes`.

```
/** Returns the minimum value in an array.
* @param array an array, must not be null or empty
* @return the minimum value in the array
* @throws IllegalArgumentException if array is null
* @throws IllegalArgumentException if array is empty */
public static int min(final int... array) {
{ validateArray(array); /* ... */ }
```

Listing 5.2. Excerpt of the Javadoc comment and implementation of a method in Apache Commons Lang's class `NumberUtils`.

Running WIT on class `Bytes` automatically finds the preconditions of these (as well as many other) methods, thus providing a useful form of rigorous documentation. For instance, one of the exception preconditions found by WIT for Listing 5.1's second method:

```
 throws: IndexOutOfBoundException
   when: off >= 0 && len >= 0 && bs.length < len + off
example: [off=0, len=1, bs.length=0]
```

corresponds to the path that reaches line 7 in Listing 5.1. WIT also understands that the first method never throws this exception, but it can still throw others such as:

```
 throws: IllegalArgumentException
   when: b.length >= 0 && code.length < 64
example: [b.length=0, code.length=0]
```

In fact, WIT only reports exception preconditions that correspond to *feasible* paths. Each precondition comes with an example of argument values that make the precondition true. These are not directly usable as test inputs, since they describe the input's properties without constructing them; but they are useful complements to the precondition expressions, and help users get a concrete idea of the exceptional behavior.

5.2.2 Inconsistent Documentation

Listing 5.2 shows the complete Javadoc documentation and a brief excerpt of method `min` in the latest version of Apache Commons Lang's class `NumberUtils`, which computes the

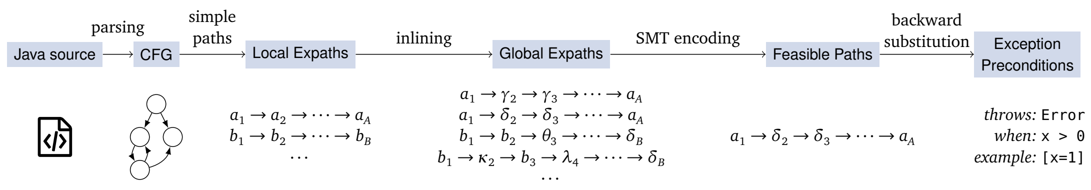

Java source
parsing
CFG
simple paths
Local Expaths
inlining
Global Expaths
SMT encoding
Feasible Paths
backward substitution
Exception Preconditions
$a_1 \to a_2 \to \cdots \to a_A$
$b_1 \to b_2 \to \cdots \to b_B$
$\cdots$
$a_1 \to \gamma_2 \to \gamma_3 \to \cdots \to a_A$
$a_1 \to \delta_2 \to \delta_3 \to \cdots \to a_A$
$b_1 \to b_2 \to \theta_3 \to \cdots \to \delta_B$
$b_1 \to \kappa_2 \to b_3 \to \lambda_4 \to \cdots \to \delta_B$
$\cdots$
$a_1 \to \delta_2 \to \delta_3 \to \cdots \to a_A$
throws: Error
when: x > 0
example: [x=1]

Figure 5.1. An overview of how WIT works. WIT parses the source code of the Java classes to be analyzed, and builds a control-flow graph (CFG) of every method. It enumerates the simple paths in every method's CFG that may end with an exception (expaths). It then transforms these expaths local to a specific method into global expaths by inlining method calls or previously extracted exception preconditions (if they are available); this may transform a single local expath into multiple global expaths. To determine which expaths are feasible, WIT encodes their constraints as an SMT problem and uses the Z3 SMT solver to check if they are satisfiable. It finally transforms all feasible paths into *exception preconditions*.

minimum of an `array` of integers. Unlike the previous example, `min`'s documentation is detailed and clearly expresses the conditions under which an exception is thrown. Unfortunately, the documentation is partially incorrect: when `array` is null, `min` throws a `NullPointerException`, not an `IllegalArgumentException`, as precisely reported by WIT:

throws: `NullPointerException` *when:* `array == null`

This inconsistency is due to a change in the implementation of `validateArray`, which is called by `min` to validate its input and uses methods of class `Validate` to perform the validation. In version 3.12.0 of the library, `validateArray` switched[98] from calling `Validate.isTrue(a!=null)` (which throws an `IllegalArgumentException` when the check fails) to calling `Validate.notNull(a)` (which throws a `NullPointerException` instead) to check that `a` is not null.

To help locate the source of any exceptional behavior, WIT also outputs the line where the exception is thrown, and possibly the triggering method call. In this example, it would clearly indicate that the exceptional behavior comes from a call to `Validate.notNull`.

This information can help detect and debug such inconsistencies, which would be quite valuable to project developers and users. As we discuss in Section 5.5.6, maintainers of Apache libraries were appreciative of our pull requests which extended the projects' documentation with some of WIT's exception preconditions.

5.3 How wit Works

Figure 5.1 overviews how WIT's analysis works. This section details each step and discusses some features of its current implementation.

WIT inputs the source code of some Java classes; it analyzes the methods and constructors of those classes to determine their *exception preconditions*, that is the conditions on the methods' input that lead to the methods throwing an exception. It then outputs the exception preconditions it could find, together with their matching exception class, as well as examples of inputs that satisfy the exception preconditions. WIT's analysis only needs the source code of the immediate classes to be analyzed: it does not need a complete project's source code, nor to compile or build the project.

```java
1  public static boolean[] insert(final int k, final boolean[] a, final boolean... v) {
2    if (a == null) { return null; }
3    if (isEmpty(v)) { return clone(a); }
4    if (k < 0 || k > a.length)
5      { throw new IndexOutOfBoundsException(); }
6    // ...
7  }
8
9  public static boolean isEmpty(boolean[] x)
10 { return getLength(x) == 0; }
11
12 public static int getLength(boolean[] y)
13 { if (y == null) { return 0; } return y.length; }
```

Listing 5.3. Excerpt of method `ArrayUtils.insert` in Apache Commons, and some of the methods it calls.

WIT can analyze both regular methods and constructors of a class. Thus, for brevity, we use the term "methods" to collectively refer to both methods and constructors.

5.3.1 Parsing and CFG

WIT parses the source code given as input using JavaParser,[99] and constructs a control-flow graph (CFG) of the methods in the input classes using library JGraphT.[100] More precisely, we build a CFG for each method `m` individually; and annotate branches in the CFG with each branch's Boolean condition.

Listing 5.3 shows excerpts of 3 methods of class `ArrayUtils`[101] in Apache Commons Lang. Method `insert` puts some values `v` into an array `a` of Booleans at a given index `k`. The initial part of its implementation calls another method, `isEmpty`, of the same class to determine if `v` is empty; in turn, `isEmpty` calls method `getLength`. WIT builds CFGs for `insert`, `isEmpty`, and `getLength`, since they are all part of the input source code.

5.3.2 Local Exception Paths

When analyzing a method `m`, WIT collects its *local exception paths* ("expaths" for short). These are all simple directed paths[2] on `m`'s CFG that end with a node corresponding to a statement that may throw an exception—either explicitly with a `throw` or indirectly with a a *call* (which may return exceptionally).

In Listing 5.3's example, one of `insert`'s local expaths p goes through the else branch on lines 2–3 and through the *then* branch on line 4, ending with the `throw` on line 5:

$$p\colon \texttt{if}_2 \xrightarrow{\texttt{a!=null}} \texttt{if}_3 \xrightarrow{\texttt{!isEmpty(v)}} \texttt{if}_4 \xrightarrow{\texttt{k<0 || k>a.length}} \texttt{throw}_5$$

[2]A simple path is one where any one node appears at most once. We compute them using JGraphT's `AllDirectedPaths` method.[102]

5.3.3 Global Exception Paths

After collecting expaths local to each method, WIT converts them into *global* expaths by *inlining* calls to other methods.

Given a local expath ℓ, for each node $n_{\mathtt{x}}$ in ℓ that calls some other method `x`, WIT checks whether `x`'s CFG is available (that is, whether `x`'s implementation was part of the input). If it is, WIT enumerates all simple paths that go through the CFG of `x`, and splices each of them into ℓ at $n_{\mathtt{x}}$. In other words, it transforms the local path ℓ so that it follows inter-method calls. Since a method usually has multiple paths, one local expath may determine several global expaths after inlining. WIT inlines calls recursively (with some limits that we discuss in Section 5.3.7).

When a called method `x`'s CFG is not available in the current run, WIT first looks whether it analyzed `x`'s source code in some of its previous runs. If this is the case, WIT replaces the call to `x` with `x`'s exception preconditions it extracted in the previous runs—following the modular analysis procedure we explain in Section 5.3.4. Otherwise, if no information about `x` is available or the user deliberately disabled modular analysis, WIT doesn't inline calls to it and marks them as "opaque".

WIT inlines the call to `isEmpty` in local expath p (Listing 5.3's example) since `isEmpty` is part of the same analyzed class `ArrayUtils`. Inlining the call replaces p's edge $\mathtt{if}_3 \xrightarrow{\mathtt{!isEmpty(v)}} \mathtt{if}_4$ with `getLength`'s only path: $\mathtt{if}_3 \xrightarrow{\mathtt{!(getLength(v)==0)}} \mathtt{if}_4$. Since the implementation of `getLength` is available too, WIT recursively inlines its two paths, which finally gives two global expaths p_1, p_2 that inline `insert`'s local expath p's calls:

$$p_1 : \mathtt{if}_2 \to \mathtt{if}_3 \to \mathtt{if}_{13} \xrightarrow{\mathtt{v==null, 0\ !=\ 0}} \mathtt{if}_4 \to \mathtt{throw}_5$$

$$p_2 : \mathtt{if}_2 \to \mathtt{if}_3 \to \mathtt{if}_{13} \xrightarrow{\mathtt{v!=null, v.length\ !=\ 0}} \mathtt{if}_4 \to \mathtt{throw}_5$$

5.3.4 Modular Analysis

By default, WIT saves all exception preconditions it extracts—together with their associated global exception paths—in a database, so that they can be reused to perform a modular analysis. This is useful whenever a method `m` in some project A calls another method `n` in some other project B. If we provide A and B in a single run, WIT's analysis has access to all the source code; thus, in principle, it can inline the code of B's `n` when analyzing A's `m`. However, this may not scale, as the number of paths to be considered grows like the product of `m`'s and `n`'s paths. To perform modular analysis, we instead first run WIT on B alone; then, we run it on A alone. When WIT analyzes `m` in A, it finds that it calls an external method `n` in B; thus, it reuses `n`'s saved exception precondition information to analyze the exceptional behavior of `m` when analyzing A without having to analyze `n` again (or without treating it like an opaque method, which may miss information).[3]

[3]"Modular analysis" simply refers to WIT's capability of reusing the exception preconditions of previously analyzed projects. The user controls how this capability is applied: WIT will always access the complete source code of the project or projects given to it as input; if the user wants to analyze a project B separately from another project A, they will have to run WIT twice (once on B, and then once on A) with modular analysis enabled.

More precisely, if modular analysis is enabled, whenever a node $n_{\mathtt{x}}$ in a local expath ℓ calls a method `x` that was analyzed in a previous run, WIT replaces the call to `x` by inlining any global exception path associated with `x`'s exception preconditions (and replacing, as usual, `x`'s formal parameters with the actual call arguments). Just like regular inlining (Section 5.3.3), this may introduce multiple global expaths for a single call to `x`. It is necessary, in general, to consider all available global expaths for a called method, so that all possible side effects of the call are accounted for. WIT can use both expres and maybes for modular analysis.[4] Since maybes are not guaranteed to be correct, any global expath that includes a maybe is automatically also classified as maybe.

As an example of where modular analysis can improve WIT's capabilities, consider Listing 5.4. Method `generate`{103} of class `RandomStringGenerator` in project Commons Text calls method `Validate.isTrue`{104} in another project Commons Lang. If we run WIT on project Commons Text alone, the call to `isTrue` is marked as opaque, and hence no exception precondition would be reported for this path. We could run WIT on both projects Commons Text and Commons Lang together; this would take a considerable amount of time, and it would not scale to combining even more projects. Instead, we can use WIT's modular analysis and first analyze Commons Lang in isolation; this would report the exception precondition `!expr` for method `validate.isTrue`. Then, when WIT runs on Commons Text, it would replace the call to `isTrue` in `generate` with `if (!(length > 0))throw new IAE()`,[5] which leads to inferring exception precondition `length <= 0` for this path in method `generate`.

As we will demonstrate in Section 5.5, modular analysis can boost WIT's output and help achieve a better scalability.

Implementation-wise, WIT persists JSON objects into a MongoDB{105} instance. For JSON serialization and deserialization, we combine JavaParser's serialization package{106} with the Moshi JSON library.{107}

5.3.5 Path Feasibility

WIT builds global expaths only based on syntactic information in the CFGs; therefore, some paths may be infeasible (not executable). To determine whether a global expath is feasible, WIT encodes it in logic form as an SMT (Satisfiability Modulo Theory) formula [15], and uses the Z3 SMT solver [47] to determine whether the expath's induced constraints are feasible.

To this end, it first transforms the path into SSA (static single assignment) form, where complex statements are broken down into simpler steps, and fresh variables store the intermediate values of every expression. We designed a logic encoding of Java's fundamental types (`int`, `boolean`, `byte`, arrays, strings) with their most common operations (including arithmetic, equality, `length`, `contains`, `isEmpty`), as well as of a few widely used JDK library methods (such as `Array.getLength`). WIT uses this encoding to build an SMT formula ϕ corresponding to each global expath p: if ϕ is satisfiable, then the global expath p is feasible, and hence it corresponds to a possible exceptional behavior of method `m`.

[4]"Expres" and "maybes" are precisely introduced in Section 5.3.5. In a nutshell, expres come from expaths that are provably feasible, and hence they are correct by constructions; maybes come from expaths with inconclusive feasibility analysis, and hence they are just educated guesses that may be incorrect.

[5]IAE here is short for `IllegalArgumentException`.

```java
// In project Commons Text, class RandomStringGenerator
public String generate(final int length) {
   if (length == 0) {
      return StringUtils.EMPTY;
   }
   Validate.isTrue(length > 0, "Length %d is smaller than zero.", length);
   // ...
}

// In project Commons Lang, class Validate
public static void isTrue(final boolean expr, final String msg, final long value) {
  if (!expr) {
     throw new IllegalArgumentException(String.format(msg, Long.valueOf(value)));
  }
}
```

Listing 5.4. An example of code that can benefit from modular analysis: method `RandomStringGenerator.generate()` in project Commons Text calls method `Validate.isTrue()` in another project Commons Lang.

WIT encodes ϕ as a Python program using the Z3 SMT solver's Z3Py Python API.[6] Listing 5.5 shows a simplified excerpt of the SMT program encoding the feasibility of `insert`'s global expath p_1. First, it declares logic variables of the appropriate types to encode program variables (e.g., `k`), their basic properties (e.g., `a_length`, which corresponds to the Java expression `a.length`), and the values passed via method calls (e.g., `getLength` is an integer variable storing `getLength()`'s output). Then, it builds a list `c` of constraints that capture the path constraints and the semantics of the statements along the path. For example, `a_length` must be nonnegative, since it corresponds to array `a`'s length (line 5); the properties of array `v` are copied to those of `x`, since `insert`'s argument `v` is the actual argument for `isEmpty`'s formal argument `x` (line 7); and path constraint `!isEmpty(v)` corresponds to the complement of Boolean variable `isEmpty` (line 12). In this case, Z3 easily finds that the constraints in `c` are unsatisfiable, since `Not(0 == 0)` is identically false. In contrast, the constraints corresponding to path p_2 are satisfiable, and thus Z3 outputs a satisfying assignment of all variables in that case.

Sometimes WIT does not have sufficient information to determine with certainty whether a path is feasible. When a path includes a call to an opaque method (whose implementation is not available or when the analysis fails) WIT's feasibility check is underconstrained. In these cases, WIT still performs a feasibility check but reports any results as *maybe*, to warn that the output may not be correct.

In Listing 5.3's example, suppose that `getLength`'s implementation wasn't available. In this scenario, based on its signature, WIT would only know that `getLength` returns an integer without any constraints; therefore it would classify path p as feasible but mark it as *maybe* since it is just an educated guess without correctness guarantees.

[6] WIT's Z3 ad hoc encoding also handles aliasing by explicitly keeping track of possible aliases along each checked path. Thanks to the other heuristics that limit path length (Section 5.3.7), this approach is feasible in practice.

```
# logic variables
k = Int('k')
a_null = Bool('a==null')
a_length = Int('a.length')
c = [a_length >= 0, v_length >= 0] # implicit
c += [Not (a_null)] # a != null
x_null, x_length = v_null, v_length # call isEmpty
y_null, y_length = x_null, x_length # call getLength
c += [y_null] # y == null
getLength = 0 # return 0
isEmpty = (getLength == 0) # return getLength(x)==0
c += [Not(isEmpty)] # !isEmpty(v)
c += [Or(k < 0, k > a_length)] # k < 0 || k > a.length
```

Listing 5.5. Excerpt of the SMT encoding corresponding to global expath p_1 of method `insert` in Listing 5.2.

5.3.6 Exception Preconditions

A feasible path p identifies a range of inputs of the analyzed method `m` that trigger an exception. In order to characterize those inputs as an *exception precondition*, WIT encodes p's constraints as a formula that only refers to `m`'s arguments, as well as to any members that are accessible at `m`'s entry (such as the target object `this`, if `m` is an instance method). To this end, it works backward from the last node of exception path p; it collects all path constraints along p, while replacing any reference to local variables with their definition. For example, method `void f(int x){int y=x+1; if(y > 0)throw;}` has a single feasible expath with path condition `y > 0`, which becomes `x + 1 > 0` after backward substitution through the assignment to variable `y`. Since `x + 1 > 0` only mentions argument `x`, it is a suitable exception precondition for method `m`.

Sometimes WIT cannot build an exception precondition expression that only mentions arguments and other visible members. A common case is when a path includes opaque calls: since the semantics or implementation of these calls is not available, any expressions including them may not make sense in a precondition. In all these cases, WIT still reports the exception expression obtained by backward substitution, but marks it as a *maybe* to indicate that it may not be correct. Another, more subtle case occurs when the exception precondition Boolean expression includes calls to methods (as opposed to just variable lookups). If these methods are not *pure* (that is, they do not change the program state), the precondition may be not well-formed. For instance, a precondition `x.inc()== 0`, where calling `inc` increments the value of `x`. Here too, WIT is conservative and marks as *maybe* any exception precondition that involves calls to methods that are not known to be pure.

Before outputting any exception preconditions to the user, WIT *simplifies* them to remove any redundancies and display them in a form that is easier to read. To this end, it uses SymPy [139],[108] a Python library for symbolic mathematics. Java's syntax is sufficiently similar to C's that we can also enable SymPy's pretty printing of expressions using C syntax, and then additionally tweak it to amend the remaining differences with Java. While conceptually simple, the simplification step is crucial to have readable exception preconditions. For example, SymPy simplifies the ugly expression

```
(!(x==null))&&(!(x==null))&&(0+1==1)&&(y<0||y>x.length)
```

into the much more readable

```
(y > x.length || y < 0)&& null != x
```

which doesn't repeat `x != null` and omits the tautology `0 + 1 == 1`.

WIT's final output consists of a series of tuples with: (a) an exception precondition, (b) whether it is a *maybe*, (c) the thrown exception type, (d) and an example of inputs that satisfy the precondition (given by Z3's successful satisfiability check). For debugging, WIT can also optionally report the complete `throw` statement (including any exception message or other arguments used to instantiate the exception object), the line in the analyzed method `m` where the exception is thrown or propagated, and a sequence of method calls starting from the analyzed method and ending in the throwing method. Moreover, WIT reports the generated Z3 and SymPy Python programs' source code.

5.3.7 Heuristics and Limitations

Let us now zoom in on a few details of how WIT's implementation works, which clarify its capabilities and limitations. To put these details into the right perspective, let us recall WIT's design goals: it should be precise and lightweight; it's acceptable if achieving these qualities loses some generality—as long as a sizable fraction of exception preconditions can be precisely determined.

Using maybes. As discussed in Section 5.3.5, WIT provides two disjoint sets of exceptional preconditions as output: expres and maybes. In practice, reporting both gives users more flexibility in how to use WIT's output according to different use cases. If correctness is crucial (for example, if one uses WIT's output as formal specification), then users should only consider expres and ignore maybes. On the other hand, if some degree of uncertainty in the correctness of an exception precondition is acceptable in exchange for a higher recall, then users may also consider maybes. The snag is that they may have to spend extra effort to validate the maybes, but this may be acceptable if there exist practical validation means (for example, an extensive test suite). Any kind of hybrid approach is also possible; for instance, one may first only use expres, but consider using maybes selectively for a few methods where WIT's feasibility analysis struggled due to the features used there.

Implicit exceptions. WIT only tracks exceptions that are explicitly raised by a `throw` statement; it does not consider low-level errors—such as division by zero, out-of-bound array access, and buffer overflow—that are signaled by exceptions raised by the JVM. This restriction is customary in techniques that infer exceptional behavior, since implicitly thrown exceptions are "generally indicative of programming errors rather than design choices [195]" [24], and usually do not belong in API-level documentation [66] and are best analyzed separately. Extending WIT to also track implicit exceptions would not be technically difficult; for example, one could first instrument the code to be analyzed with explicit checks before any statement that may thrown an implicit exception.[7] However, indiscriminately considering all exceptions that are thrown implicitly would produce a vast number of boilerplate exception preconditions that are not specific to a method's explicitly programmed behavior; hence, they would be outside WIT's current focus.

Java features. WIT's CFG construction currently does not fully support some Java features: `instanceof` operators, `for-each` loops, `switch` statements, and `try/catch` blocks. When these features are used, the CFG may omit some paths that exist in the actual program. (Supporting the latter three features is possible in principle, but would substantially complicate the CFG construction.)[8] The SMT encoding used for path feasibility (Section 5.3.5) is limited to a core subset of Java features and standard library methods. As a result, WIT won't report exception preconditions that involve unsupported features (or will report them as *maybe*, that is without correctness guarantee).

Path length and number. In large methods, even some local expaths can be too complex, which bogs down the whole analysis process. Therefore, WIT only enumerates paths of up to $N = 50$ nodes, which have a much higher likelihood of being manageable. Complex methods may have thousands of local paths. Therefore, WIT analyzes up to $N = 500$ paths of a given method or constructor.

Inlining limits. Inlining can easily lead to a combinatorial explosion in the number and length of the expaths; therefore, a number of heuristics limit inlining. First, a path can be inlined only if it is up to $N = 50$ nodes—the same limit as for local expaths. Second, WIT stops inlining a call in a path after it has reached a limit of $I = 100$ inlined paths—that is, it has branched out the call into I different ways. It can still inline other calls in the same path, but this limit avoids recursive inlinings that are likely to blow up. Third, WIT enumerates the inlinings of a call in random order; in cases where the limit I is reached, this increases the chance of collecting a more varied set of inlined paths instead of getting stuck in some particularly complex ones (if the limit I is not reached, the enumeration order is immaterial).

Maybes heuristics. The feasibility of exception preconditions reported as maybes could not be verified; hence, they are educated guesses. Consequently, WIT deploys two simple heuristics that filter out maybes that are overwhelmingly unlikely to be correct. First, WIT does not report any maybe assertion that consists of more than six conjuncts or disjuncts; we found that the constraints of such large maybes are usually unsatisfiable. Second, WIT drops any maybe that includes constraints over private fields of the JDK's `String` and `StringBuilder` classes. This heuristic only applies when WIT uses *modular* analysis: these two JDK classes have a complex implementation involving native code and JVM internals. Thus, WIT's analysis of `String` and `StringBuilder` can only retrieve a few correct maybes; as a result, using them in the modular analysis of other client classes is likely to introduce a large number of spurious maybes—which this heuristic avoids.

Timeouts. Z3's satisfiability checks (to determine if a path is feasible) may occasionally run for a long time. WIT limits each call to Z3 to a $Z = 15$-second timeout; when the timeout

[7] As a simple example, as done for testing [68], before every array access such as `x := a[k]` add a guard `if (!(0 <= k && k < a.length))throw new IndexOutOfBoundsException()`, so that the implicitly thrown exception becomes explicit.

[8] Even mature static analysis frameworks such as Spoon have only partial/experimental support for features such as try/catch.[109]

expires, Z3 is terminated and the path is assumed to be infeasible. There is also an overall timeout of $T = 10$ minutes per analyzed class. If WIT's analysis still runs after the timeout, it probably means that the class's methods are particularly intricate and hard to process;to remain lightweight, WIT skips to the next class.

Configurable options. The parameters regulating these heuristics can be easily changed if one needs to analyze code with peculiar characteristics, when a large running time is not a problem. WIT also offers two slightly different Z3 logic encodings of some Java features. By default, it employs a conservative encoding that ensures that all expressions used in an exception precondition are well defined (for example, `a.length` implicitly requires that `a != null`). In some complex cases, this encoding may be overly conservative, leading to marking as unsatisfiable exception preconditions that are actually correct. To accommodate these unusual cases, WIT also offers a less conservative logic encoding of the same features, which trades off correctness for recall; users can switch to this alternative encoding when analyzing software where a high recall is more important than an absolute correctness guarantee.

Modular analysis. WIT's modular analysis (Section 5.3.4) is also configurable to fit each application scenario. By default, WIT performs modular analysis: if it encounters a call to a method that it analyzed in a previous run, it uses the called method's exception preconditions to determine the exception preconditions of the caller. In contrast, if the user explicitly disables modular analysis, WIT analyzes each project in isolation. Section 5.5.4 describes experimental data that we collected to better understand the practical impact of using WIT's modular analysis. When modular analysis is enabled, WIT can reuse only expres or both expres and maybes. This is another parameter that one can choose according to how important a high recall is: reusing also maybes can only increase the number of maybes inferred by WIT, which come with no guarantee of being correct. In general, modular analysis is an additional option made available by WIT, which need not be used in all situations: whether enabling it is beneficial depends on the projects under analysis and on the user's requirements.

5.4 Experimental Evaluation

This section describes the empirical evaluation of WIT, which targets the following research questions.

RQ1 (precision): How many of the exception preconditions detected by WIT are correct?

RQ2 (recall): How many exception preconditions can WIT detect?

RQ3 (features): What are the most common features of the exception preconditions detected by WIT?

RQ4 (modularity): How do the exception preconditions detected by WIT change if modular analysis is disabled?

RQ5 (efficiency): Is WIT scalable and lightweight?

RQ6 (usefulness): Are WIT's exception preconditions useful to complement programmer-written documentation?

5.4.1 Experimental Subjects

In our evaluation, we ran WIT on two groups of projects: several standard libraries in Java's JDK and 46 open-source Java projects surveyed by recent papers investigating the (mis)use of Java library APIs [96, 196, 207] and the automatic generation of tests for some of these libraries [150]. Table 5.1 lists all our experimental subjects.

JDK modules. The JDK (Java Development Kit) includes arguably Java's most widely used and mature libraries, featuring virtually in every Java project [95, 150] and abundantly documented. We selected JDK 11{110} to run our experiments, since it's the most recent LTS (Long Term Support) release that JavaParser can handle at the time of writing. Given the JDK's gargantuan size and complexity, we selected five of its modules (subdirectories of `java.base/share/classes`) and ran WIT on all of them as if it were a regular Java project: modules com/sun, java, javax, sun, and jdk.

Other projects. The other group of 46 experimental subjects includes several projects that are also large, widely-used, mature Java projects in various domains (base libraries, GUI programming, security, databases)—especially the 26 projects from the Apache Software Foundation, which recent empirical research has shown to be extensively documented and thoroughly tested [150, 207]. On the other hand, a few projects taken from [96] are smaller, less used, or both. For instance, projects gae-java-mini-profiler, visualee, and AutomatedCar are no longer maintained. This minority of projects makes the selection more diverse, so that we will be able to evaluate WIT's capabilities in different scenarios.

We used the latest commit/stable release in every project, at the time of writing, with two exceptions: Apache lucene-solr was recently split into two separate projects, and thus we used the last version before the split; we analyzed version 2.6 of Apache Commons IO to match [150]'s thorough manual analysis—which we used as ground truth to answer RQ2.

5.4.2 Experimental Setup

We ran WIT on the source code of all projects, after excluding directories that usually contain tests (e.g., `src/test/`) or other auxiliary code. All experiments ran on a Windows 11 Intel i9 laptop with 32GB of RAM. By default, WIT only infers the exception preconditions of *public* methods; if a public method calls a non-public one, WIT will also analyze the latter , but will report only public exception preconditions. WIT analyzes each class in isolation; then, it combines the results for all classes in the same project and outputs them to the user.

Unless we explicitly state otherwise, WIT ran with default options in the experiments. In particular, it performed *modular analysis* (described in Section 5.3.4); therefore, we first ran WIT on the JDK modules, then on the Apache Commons libraries (lang, io, text, math, configuration, in this order) followed by all other projects in alphabetical order. Since practically all projects use some JDK libraries, and several projects also use Apache Commons libraries, this execution order maximizes the chances that WIT can reuse the results of one of its previous runs to perform an effective modular analysis. In contrast, client-of dependencies between projects other than the JDK and Apache Commons libraries are more sparse; therefore, the alphabetical order is somewhat arbitrary, but even following a different order is unlikely to significantly affect WIT's capabilities.

To answer **RQ1 (precision)**, we performed a manual analysis of a sample of all exception preconditions reported by WIT to determine if they correctly reflect the exceptional behavior of the implementation. This thesis' author tried to map each inferred exception precondition to the source code of the analyzed method. In nearly all cases, the check was quick, and its outcome clear. The few exception preconditions whose correctness was not obvious were analyzed by this work's co-author as well, and the final decision was reached by consensus. We were conservative in checking correctness: we only classified an exception precondition as correct if the evidence was clear and easy to assess.

To answer **RQ2 (recall)**, we used Nassif et al. [150]'s dataset—henceforth, DSC—as ground truth. DSC includes 844 manually-collected exception preconditions[9] (expressed in structured natural language, e.g. "if `offset` is negative") for all public methods in Apache Commons IO's base package collected from all origins (package code, libraries, tests, documentation, ...). We counted the exception preconditions inferred by WIT that are semantically equivalent to any in DSC. Matching DSC's natural-language preconditions to WIT's was generally straightforward, as we didn't have to deal with subtle semantic ambiguities: since WIT only reports correct exception preconditions as expres, we only had to match (usually simple) natural-language expressions to their Java Boolean expression counterparts.

Using DSC as ground truth assesses WIT's recall in a somewhat restricted context: (i) DSC targets exclusively the Commons IO project, whose extensive usage of I/O operations complicates (any) static analysis; (ii) DSC describes all sorts of exceptional behavior, including the "not typically documented" runtime exceptions [150]. To assess WIT's recall on a more varied collection of projects, we also considered Zhong et al. [207]'s dataset—henceforth, DPA—which includes 503 so-called "parameter rules" of public methods in 9 projects (a subset of our 46 projects described in Section 5.4.1). A parameter rule is a pair $\langle m, p \rangle$, where m is a fully-qualified method name and p is one of m's arguments; it denotes that calling m with some values of p may throw an exception. Important, parameter rules do not express the *values* of p that determine an exception, and hence they are much less expressive than preconditions; however, they are still useful to determine "how much" exceptional behavior WIT captures. We counted the exception preconditions inferred by WIT that match DPA: a precondition c matches a parameter rule $\langle m, p \rangle$ if c is an exception precondition of method m that depends on the value of p. This is a much weaker correspondence than for DSC, but it's all the information we can extract from DPA's parameter rules.

To better characterize the exception preconditions that WIT could *not* infer, we performed an additional manual analysis of: (a) 746 of DSC's exception preconditions among those that WIT did not infer (b) 218 exception preconditions reported by WIT as "maybe" (that is, which may be incorrect). These 964 additional cases help assess what it would take to improve WIT's recall.

To answer **RQ3 (features)**, during the manual analysis of precision we also classified the basic features of each exception precondition r of a method m. We determine whether

[9]We exclude 6 innacurate cases.

r corresponds to an exception that is thrown directly by *m* or propagated by *m* (and thrown by a called method). We count the number of Boolean connectives `||` and && in *e*, which gives an idea of *r*'s complexity. Then, we determine if each subexpression *e* of *r* constraints *m*'s *arguments*, or *m*'s object *state*; and we classify *r*'s check according to whether it is: (a) a *null* check (whether a value is null), (b) a *value* check (whether a value is in a certain set of values), (c) a *query* check (whether a function call returns certain values). For example, here are expressions of each kind for a method `m` with arguments `int x` and `String y`, whose class includes fields `int[] a`, `int count`, and method `boolean active()`:

`void m(int x, int[] y)`	*argument*	*state*
null	`y == null`	`this.a != null`
value	`x == 1`	`this.count > 0`
query	`y.isEmpty()`	`!this.active()`

An exception precondition may combine expressions of different kinds; for instance, `a != null && a.length > 0` combines a null and a value check.

To answer **RQ4 (modularity)**, we ran WIT again on 5 projects with modular analysis *disabled*, and compared WIT's output on these projects with and without modular analysis. We selected the 5 projects from diverse domains, which demonstrate using different JDK libraries and methods. Besides comparing the number of reported exception preconditions with and without modular analysis, we manually inspected 75 maybes: (a) For each project, among methods for which both the modular and non-modular analysis reported *some* maybes, we randomly picked 6 maybes reported by the non-modular analysis and 6 maybes reported by the modular analysis for the same methods;[10] this sample of 60 maybes ($6 \times 5 \times 2$) gives us an idea of how maybes change when modular analysis is enabled. (b) For each project, among methods for which *only* the modular analysis reported some maybes, we randomly picked 3 maybes; this sample of 15 maybes (3×5) demonstrates cases where the modular analysis strictly outperforms the non-modular one.

To answer **RQ6 (usefulness)**, we first inspected the source code documentation (Javadoc and comments) of all methods with exception preconditions analyzed to answer RQ1, looking for mentions of the thrown exception types and of the conditions under which they are thrown. We focused on Javadoc documentation: while we also considered non-structured comments a priori, all cases of documented exceptional behavior that we found used at least some Javadoc syntax. We also selected 90 inferred exception preconditions among those that were not already documented, and submitted them as 8 pull requests in 5 projects: Accumulo,[111] Commons Lang,[112],[113],[114] Commons Math,[115],[116] Commons Text,[117] and Commons IO.[118] We selected these five projects as they are very active and routinely spend effort in maintaining a good-quality documentation. Each pull request combines the exception preconditions of methods in the same class or package, and expresses WIT's exception preconditions using Javadoc `@throws` tags. To compile each pull request, we sometimes complemented the Javadoc with a brief complementary natural-language description, and possibly some tests (expressing WIT's example inputs in the form of unit tests). We also tried to adjust the Javadoc syntax to be consistent with each project's style (for example, express-

[10] To ensure a more varied sample, we targeted $3 + 3$ methods that use the JDK and $3 + 3$ that do not.

ing `a != null` as either `a not null` or `@code a != null`). In all cases, reformulating WIT's output was a trivial matter.

5.5 Experimental Results

As described in Section 5.3.6, WIT produces two kinds of exception preconditions. The main output are those whose feasibility was fully checked (Section 5.3.5); others are marked as *maybe* and can still be correct but have no guarantee. As done in previous sections, we call "expres" the former and "maybes" the latter. Unless explicitly stated otherwise, the term "project" denotes any of the 51 experimental subjects (Section 5.4.1): one of the 5 JDK modules or one of the 46 open-source projects we analyzed.

5.5.1 RQ1: Precision

Overall, WIT reported 30 487 expres and 31 043 maybes in 40 263 methods (24 461 methods with some expres and 17 564 with some maybes)—out of a total of 460 032 analyzed public methods from 59 733 classes in 51 projects.

In order to validate WIT's feasibility check, we manually analyzed a sample of 742 expres to determine if they are indeed correct. This sample size is sufficient to estimate precision with up to 5% error and 99% probability with the most conservative (i.e., 50%) a priori assumption [44]; thus, it gives our estimate good confidence without requiring an exhaustive manual analysis [150, 210]. We applied stratified sampling to pick the 742 expres: we randomly sampled 10 instances in each of the 49 projects where WIT detected some expres.[11] This manual analysis found that *all* expres were indeed correct, that is 100% precision.

As we explained in Section 5.3, WIT's maybes still have a chance of being correct exception preconditions, but they remain educated guesses in general. We randomly picked 218 maybes uniformly in the 50 projects that report some[12] and manually checked them as we did for the expres. We found that 47% (102) of them are indeed correct; thus, WIT's precision remains high ($88\% = (102 + 742)/(218 + 742)$) even if we consider all maybes. As we further discuss in Section 5.5.2, in most cases, WIT could not confirm the maybes as correct because they involve unsupported Java features (see Section 5.3.7).

Manually analyzing a significant sample of exception preconditions (expres) confirmed that WIT is 100% precise.

5.5.2 RQ2: Recall

We compute the recall on both datasets DSc and DPa in four ways: considering only expres or also maybes; and considering only WIT's supported features or all Java features. Table 5.2 summarizes the results that we detail in the following.

[11] We pick all expres for 7 projects with less than 10 expres in total.

[12] To keep the manual analysis manageable, this sample size (218) is sufficient to estimate the precision of maybes with up to 5% error and 95% probability but with a stronger (i.e., 83%) a priori assumption.

Table 5.1. Exception preconditions inferred by WIT. For each analyzed PROJECT: the short git commit HASH; the size of the analyzed source code in thousands of lines (KLOC); WIT's total running TIME in minutes; the number # of inferred exception preconditions (EXPRES), the number M of methods and constructors with some inferred exception preconditions, the precision P based on a manual analysis of a sample, the number ?# of MAYBES exception preconditions, and the percentage ?P of these that are correct based on a manual analysis of a sample.

				EXPRES			MAYBES	
PROJECT	HASH	KLOC	TIME	#	M	P	?#	?P
com/sun	–	30	–	55	48	1.0	78	–
sun	–	128	–	566	474	1.0	1 068	–
java	–	209	–	3 420	2 578	1.0	1 666	–
javax	–	8	–	190	145	1.0	41	–
jdk	–	52	–	847	742	1.0	598	–
overall JDK	da75f3c4ad5	428	765	5 078	3 987	1.0	3 451	0.36
accumulo	7db0561cac	33	311	995	908	1.0	1 335	0.3
Activiti	31024bc756	103	150	685	543	1.0	212	0.2
asm	72e8ec49	28	130	203	126	1.0	428	0.8
asterisk-java	5c56735c	30	27	27	24	1.0	46	0.4
AutomatedCar	c137e56a	4	2	2	2	1.0	4	0.5
Baragon	10660b41	15	6	10	10	1.0	50	0.2
bigtop	ee28ba88	6.5	4	9	9	1.0	6	0.2
byte-buddy	4c57c80aab	57	974	356	348	1.0	374	0.8
camel	0a735ae926c	972	2 626	1 558	1 276	1.0	1 111	0.4
closure-compiler	fe0cebacd	287	538	158	157	1.0	654	0.2
commons-bcel	f1a1459f	35	137	76	74	1.0	896	0.4
commons-configuration	1b406c17	20	12	170	139	1.0	53	0.4
commons-io	2ae025fe	9.5	23	240	187	1.0	186	0
commons-lang	90e0a9bb2	29	55	611	484	1.0	230	0.8
commons-math	674805c64	61	264	1 078	612	1.0	573	0.8
commons-text	21fc34f	10	32	235	156	1.0	138	0.6
Confucius	e375cb9	0.5	1	45	18	1.0	14	0.4
curator	9aafdec9	26	35	192	116	1.0	126	0.6
dubbo	b5e65a6d2	99	274	413	341	1.0	225	0.4
flink	db248b2176	568	1 245	5 661	4 059	1.0	5 201	0.8
gae-java-mini-profiler	9cb1ba6	0.5	1	0	0	–	0	–
h2database	0ee51f54a	150	229	526	507	1.0	834	0.6
httpcomponents-client	29ba623eb	32	37	27	24	1.0	90	0.4
itext7	ae78654a5	145	880	681	522	1.0	702	0.7
jackrabbit	35d5732bc	260	300	1 224	1 111	1.0	1 595	0.8
jackrabbit-oak	f8c7b551a4	26	334	502	493	1.0	667	0.4
jackson-databind	972d5a28a	63	57	180	166	1.0	153	0.6
jfreechart	5aac9ae4	84	133	1 387	1 149	1.0	800	1.0
jmonkeyengine	499e73ab0	19	376	634	569	1.0	1 220	0.2
joda-time	27edfffa	29	58	250	228	1.0	355	0.6
logging-log4j2	59f6848b7	99	159	472	304	1.0	392	0.2
lucene-solr	7ada4032180	685	1 545	3 380	2 755	1.0	4 132	0.6
pdfbox	01bce4dde	106	230	255	239	1.0	362	0.2
poi	270107d9e	260	403	710	624	1.0	1 851	0.2
santuario-xml-security-java	86179876	35	38	167	142	1.0	131	0.6
shiro	0c0d9da2	27	39	154	141	1.0	145	0.2
spoon	34c23fc7	75	86	272	268	1.0	357	0.4
spring-cloud-gcp	6c95a16f	20	20	13	13	1.0	10	0.8
spring-data-commons	4acd3b70	28	24	31	29	1.0	123	0.4
swingx	9e33bc0	72	108	157	149	1.0	217	0.8
traccar	eac5f4889	54	60	2	2	1.0	76	0
visualee	88732d9	1.8	3	0	0	–	3	0
weiboclient4j	80556b1	7.8	10	6	6	1.0	9	0.2
wicket	7c0009c8df	109	1 069	930	811	1.0	656	0.6
wildfly-elytron	3457737d98	80	128	340	316	1.0	233	0.2
xmlgraphics-fop	7edce5dd5	165	940	385	318	1.0	617	0.6
overall other projects	–	5 720	14 116	25 409	20 474	1.0	27 592	0.5
overall	–	6 148	14 881	30 487	24 461	1.0	31 043	0.5

Table 5.2. WIT's recall using two datasets DSc and DPa (described in Section 5.4.2) as ground truth. For each PROJECT, # is the dataset's total number of exception preconditions (DSc) or parameter rules (DPa); the other columns reports the percentage correctly inferred by WIT: E only considers expres, E+M expres and maybes; ALL considers all exception items; SUPPORTED only those with features WIT supports.

			ALL		SUPPORTED	
DATASET	PROJECT	#	E%	E+M%	E%	E+M%
DSc [150]	commons-io	844	9	12	57	72
DPa [207]	asm	54	6	23	25	75
	commons-io	65	77	78	94	96
	jfreechart	42	80	85	84	89
	overall	1,345	13	23	48	84

Dataset DSc

Out of DSc [150]'s 844 manually identified exception preconditions, WIT detected 77 expres in 6 classes of Commons IO (1 in `FileNameUtils`, 4 in `LineIterator`, 15 in `IOUtils`, 8 in `FileCleaningTracker`, 44 in `FileUtils`, 3 in `HexDump`, and 2 in `ByteOrderMark`), that is a recall of 9% (77/844). However, 708 out of DSc's 844 exception preconditions are of kinds unsupported by WIT (see Section 5.3.7). After excluding unsupported exception precondition kinds,[13] WIT's recall estimate becomes 57% (77/(844 − 708)).

To better understand WIT's recall, we analyzed the 708 Commons IO exception preconditions from DSc that WIT didn't report as expres. We can classify these missed preconditions in two groups.

Unsupported features: As mentioned, the largest group of missed preconditions (547 or 77% of the missed preconditions) involve Java language features that WIT does not support.

Implicit exceptions: Another group of missed preconditions (161 or 23% of the missed preconditions) correspond to implicit exceptions that are thrown by the Java runtime (e.g., when a null pointer is dereferenced), which we deliberately ignore (as discussed in Section 5.3.7). A significant case is class `EndianUtils`{119} for which DSc reports 48 exception preconditions involving `ArrayIndexOutOfBounds` or `NullPointer` exceptions thrown implicitly.

[13] Excluding unsupported annotation kinds is a common practice in the empirical evaluation of tools that infer annotations [210].

```java
static void copyToDir(File src, File destDir) {
   if (src == null) { throw new NullPointerException(); }
   if (src.isDirectory()) { copyDirToDir(src, destDir); }
   else if (src.isFile()) { copyFileToDir(src, destDir); }
   else { throw new IOException("Source does not exist"); }
}

static void copyDirToDir(File srcDir, File destDir) {
   if (srcDir == null) { throw new NullPointerException(); }
   if (srcDir.exists() && !srcDir.isDirectory())
   { throw new IllegalArgumentException(); }
   if (destDir == null) { throw new NullPointerException();}
   if (destDir.exists() && !destDir.isDirectory())
   { throw new IllegalArgumentException(); }
   // ...
}
```

Listing 5.6. Excerpt from class `FileUtils` in project Commons IO.

Dataset DPa

Using 175 parameter rules[14] of DPA [207]'s dataset as reference suggests that WIT's recall varies considerably depending on the characteristics of the analyzed project. Overall, WIT inferred 85 matching expres and 8 matching maybes, corresponding to a recall of 49% (expres only) and 53% (expres+maybes). If we exclude the parameter rules involving features unsupported by WIT, the recall becomes 71% (expres only) and 78% (expres+maybes). WIT struggles the most on projects like asm, which extensively uses features and coding patterns{120} that WIT currently doesn't adequately support: as a result, WIT's recall is fairly low (considering all parameter rules, 6% with expres only and 23% with expres+maybes; considering only supported ones, 25%/75%). In contrast, more "traditional" Java projects like JFreeChart{121} extensively follow programming practices such as validating a method's input, which are a better match to WIT's current capabilities: as a result, WIT's recall is quite high (considering all parameter rules, 80% with expres only and 85% with expres+maybes; considering only supported ones, 84%/89%).

WIT *inferred 9–83% of the exception preconditions in Commons IO. Its recall varies considerably (6–96%) depending on the analyzed project's characteristics.*

5.5.3 RQ3: Features

Section 5.5.2's comparison of WIT's preconditions with those in DSc [150]'s extensive collection confirmed what also reported by other empirical studies [20, 210]: exception preconditions are often concise and structurally simple. This was also reflected in a manual sample

[14]The dataset contains 503 parameter rules for 9 projects; we manually analyzed 175 from projects asm, Commons IO, and jfreechart.

of 412 expres inferred by WIT,[15] which we manually inspected to determine their features. In terms of size, 74% of them are simple expressions without Boolean connectives &&/||; and only 7% include more than one connective.

In terms of control-flow complexity, 68% of WIT's expres involve exceptions that are thrown directly by the analyzed method (as opposed to propagated from a call).

Over 70% of all expres constrain a method's arguments (65% constraint *only* the arguments), whereas about 24% predicate over object state. null checks are more frequent (49% of expres), followed by value checks (40% of expres); and 81% of expres have either or both. In contrast, query checks are considerably less frequent (11% of expres include one). These features are a combination of the intrinsic characteristics of exception preconditions, and WIT's capability of detecting them. If we look at maybes, they tend to include query checks more frequently (50%), which is to be expected since a method call can be soundly used in a precondition only when it is provably pure (Section 5.3.6).

Up to 12% of the expres in the sample are the simplest possible Boolean expression: `true`. Nine of 13 expres of spring-cloud-gcp are of this kind. These usually correspond to methods that unconditionally throw an `UnsupportedOperation` exception to signal that they are effectively not available;[16] see project lucene-solr's class `ResultSetImpl` for an example.{123} In Java, this is a common idiom to provide "placeholders," which will be replaced by actual implementations through overriding in subclasses. While this is a common programming pattern that leverages polymorphism, it nominally breaks behavioral substitutability [111, 151]: a method's precondition should only be weakened [140], but no Boolean expression is weaker than `true`.

Some of the exception preconditions that we manually inspected revealed interesting and non-trivial features. WIT could infer expres embedded in complex expressions, such as in the case{124} of an empty string that triggers an exception in the "else" part e of a ternary expression. `c ? t : e`. It also followed method calls collecting complex conditions and presenting them in a readable, simplified form. For example, for a `ConcurrentModification` exception,{125} or after collecting constant values from other classes.{126} We also found examples of exceptional behavior documented in Javadocs in a way that mirrors WIT's output, such as "`IndexOutOufBoundsException` if `i < 0` or `i > array.length`".{127} In all, WIT's output is often concise and to the point—and thus readable and useful.

The exception preconditions inferred by WIT *are usually succinct and mainly involve checks of method arguments.*

5.5.4 RQ4: Modularity

To answer RQ4 (the impact of modular analysis), Section 5.5.4 first discusses how the output of WIT changes when modular analysis is disabled vs. when it is enabled; then, Section 132

[15] A subset of the 742 expres we checked for correctness in Section 5.5.1.

[16] A common instance of this programming pattern occurs when implementing immutable data structures. For example, state-modifying `List` interface methods such as `add` in class `UnmodifiableList`,{122} which is instantiated by method `unmodifiableList` in `java.util.Collections`.

presents the results of a manual comparison of a sample of exception preconditions obtained with and without modular analysis.

Exception Preconditions in Modular vs. Non-Modular

Table 5.3 presents the results of the comparison between WIT running with and without modular analysis (Section 5.3.4) on five of the projects used in our experimental evaluation.

Running time. In terms of running time, modular analysis usually leads to an increase of running time (32% longer on average); this is to be expected, since modularity generally increases the number of paths that are analyzed by WIT, as it "extends" them with information about methods analyzed in a different run.

Effectiveness. Modular analysis usually brings a modest (but non-trivial in absolute numbers) increase in the number of expres reported by WIT (2% more on average). These cases correspond to exceptional paths that include calls to external methods: in the non-modular analysis, these paths may only lead to maybes; in contrast, in the modular analysis, WIT has enough information to completely and correctly reconstruct the exceptional behavior about these paths, thus reporting expres.

Modular analysis usually brings a much bigger increase in the number of maybes (156% more on average): since maybes have no guarantee of correctness, using a maybe in a library to reason about a call within a caller method is quite likely to determine an additional maybe in the caller—which also may or may not be correct.

When modular analysis is counterproductive. However, modular analysis does not always lead to detecting more expres; for example, WIT reported 1–2% *fewer* expres in projects jfreechart and pdfbox when enabling modular analysis. This happens because modular analysis replaces a call to an opaque method with whatever exception path WIT extracted from the called method. In some cases, the called method's exception precondition may be a very partial approximation of the callee's full exceptional behavior; therefore, using it in place of the call may be counterproductive to obtain a provably feasible exception precondition in the caller. In fact, this is a common problem of modular reasoning [184]: if the callee's specification is weak, there is very little we can conclude about the caller's behavior.

Our manual analysis indicates that the overwhelming majority of cases where using modular reasoning led to *fewer* expres involved methods calling string methods such as `String.length()` and `String.equals()`. For example, when WIT analyzes `String.equals()`'s implementation in the JDK,{128} it encounters several features and special cases that limit its effectiveness, such as different string encodings{129} and compacted strings;{130} furthermore, the Java runtime represents a `String` as a byte array,{131} a type that WIT does not currently support. As a result, WIT only reports some very narrow, overly complex exception paths for `String.equals()`, corresponding to the few paths within its implementation that do not depend on any of those complex language features. What happens when WIT processes a method such as the one in Listing 5.7, which makes numerous calls to `String.equals()`, with modular analysis enabled? Replacing the calls with the previously extracted exception paths leads to an overly narrow, needlessly complex path condition, which bogs down the SMT solver and does not lead to any provably feasible path in the caller. In contrast, if

```java
public void setHighlightingMode(String highlightingMode)
{
   if ((highlightingMode == null) || "N".equals(highlightingMode)
         || "I".equals(highlightingMode) || "O".equals(highlightingMode)
         || "P".equals(highlightingMode) || "T".equals(highlightingMode))
   {
      this.getCOSObject().setName(COSName.H, highlightingMode);
   }
   else
   {
      throw new IllegalArgumentException("Valid values for highlighting mode are "
            + "'N', 'N', 'O', 'P' or 'T'");
   }
}
```

Listing 5.7. Method `setHighlightingMode` in class `PDAAnnotationWidget` of project pdfbox includes numerous calls to JDK's `String.equals()`, which complicate modular analysis.

modular analysis is disabled, WIT simply encodes the calls to `String.equals()` as Boolean variables with basic constraints, which is sufficient in some cases to get to a working proof of feasibility—and hence to an expre correctly characterizing `setHighlightingMode`'s[132] exceptional path.

Correctness of Maybes in Modular vs. Non-Modular

We first sampled 30 methods where both the non-modular and modular analysis reported *some* maybes, and inspected one maybe in each case (for a total of $30+30 = 60$ maybes). In the non-modular analysis, 27 (90%) of the 30 maybes were correct; in the modular analysis, 16 (53%) of the 30 maybes were correct. Then, we sampled 15 other methods where *only* the modular analysis reported *some* maybes, and inspected one maybe in each case (for a total of 15 maybes). Only 4 (27%) of the 15 maybes were correct.

These results suggest that WIT's modular analysis is usually less reliable at inferring (correct) maybes. This is in contrast to the inference of expres, which are correct by construction. In all, unless one wants to maximize the output of reported maybes, it may be preferable to only perform modular analysis for expres, excluding maybes.

This inferior performance of the non-modular analysis is usually due to complex language features used in the JDK or other called libraries that WIT does not adequately support; in these cases, the non-modular analysis's approach of treating these calls as black boxes is more likely to avoid generating incorrect maybes than the modular approach that reuses probably inconsistent or mismatched maybes extracted when analyzing the called libraries.

Let us discuss a few concrete examples of language features that led to incorrect maybes with the modular analysis. One is the complex behavior of floating-point arithmetic (type `Double` in Java); WIT's simple encoding of numbers cannot deal with special values such as `NaN`[133] and `Inf` (obtained, for example, when dividing `1.0` by `0.0`[134]). Another one is the JDK's Collections Framework, which would require a suitable (non-trivial) logic encoding in Z3 to work in WIT.

Table 5.3. Impact of using WIT's modular analysis (Section 5.3.4) for five PROJECTS. For each project, we consider the same measures as Table 5.1: the overall running TIME, the number # of reported expres , the number M of methods for which WIT reported at least one expre , and the number ?# of reported maybes . Each column ΔX reports the ratio between X measured with modular analysis and X measured without modular analysis; for example, WIT reports 4% more expres (1.04) in project camel when modular analysis is enabled.

		EXPRES		MAYBES
PROJECT	ΔTIME	Δ#	ΔM	Δ?#
camel	1.17	1.04	1.03	3.51
commons-io	0.65	1.03	1.06	1.81
commons-lang	2.30	1.05	1.05	6.21
jfreechart	1.70	0.99	1.01	0.91
pdfox	2.67	0.98	0.97	2.71
overall	1.32	1.02	1.02	2.56

A different kind of problem occurred when analyzing data-structure methods such as the JDK's `Stack.pop`,{135} which throws an exception when the stack is empty. WIT reports a correct exception precondition for `pop`; however, the precondition expression mentions a protected field[17] used in `Stack`'s internal representation.{136} As a result, the exception precondition is not usable correctly to analyze clients of the `Stack` class, such as in one of the maybes we inspected for project pdfbox.{137} To handle such cases [201], one could try to convert any references to private members into calls to public getter methods—if they are available.

It remains that WIT's modular analysis increases the number of expres in most projects. We found a few cases where some exception preconditions reported as maybe by the non-modular analysis became an expre in the modular analysis. One such cases was the constructor of class `IntersectionResult`{138} in project Commons Text. As you can see in Listing 5.8, the exception path that ends at line 10 involves a call to the JDK's `Math.min` function. Without modular analysis, WIT can only report the whole conditional expression `inters < 0 || inters > Math.min(sizeA, sizeB)` as a maybe. In contrast, WIT's modular analysis can recover `Math.min`'s behavior from its previous analysis of the JDK; thus, it reports two correct expres for the same exceptional path:

- `sizeB >= 0 && (inters < 0 || inters > sizeB)&& sizeB > sizeA`
- `sizeA >= 0 && sizeA <= sizeB (inters < 0 || inters > sizeA)`

Using WIT's modular analysis tends to moderately increase the number of detected expres. It also usually increases the number of detected maybes, while also lowering their correctness rate.

[17] Remember that WIT targets only top-level public methods, but may follow paths that go into private members.

```java
public IntersectionResult(final int sizeA, final int sizeB, final int inters) {
  if (sizeA < 0) {
     throw new IllegalArgumentException("Set size |A| is not positive: " + sizeA);
  }
  if (sizeB < 0) {
     throw new IllegalArgumentException("Set size |B| is not positive: " + sizeB);
  }
  if (inters < 0 || inters > Math.min(sizeA, sizeB)) {
     throw new
       IllegalArgumentException("Invalid intersection of |A| and |B|: " + inters);
  }
  // ...
```

Listing 5.8. Excerpt of class `IntersectionResult`'s constructor in project Commons Text.

5.5.5 RQ5: Efficiency

Thanks to the heuristics it employs (Section 5.3.7) and to the nature of exception preconditions WIT can infer (which tend to be simpler compared to general program behavior), WIT's analysis is quite lightweight and scalable. As shown in Table 5.1, its running times are generally short: it processed the entire Apache Commons Lang in just 55 minutes—17 seconds on average for each of the project's 200 top-level classes. It also scales well to very large projects: it analyzed the 9 780 classes of Apache Camel (the largest project in our collection) in 44 hours—just 16 seconds per class on average. Key to this performance is WIT's capability of analyzing each class in isolation, without requiring any compilation or build of the whole project.

Take method `ASMifier.appendAccess()`[139] as an example of how WIT's heuristics are useful. It is from project ASM and embedded under the internal subdirectory of the JDK. The method has several nested if-else branches, that lead to millions of paths. WIT's heuristics are crucial to avoid getting bogged down analyzing such complex pieces of code.

WIT's analysis is lightweight: on average, it takes 15 seconds per class; 30 seconds per exception precondition.

5.5.6 RQ6: Usefulness

This section discusses to what extent WIT's exception preconditions and the documented exceptional behavior of methods overlap. We first look into all projects except the JDK modules (Section 5.5.6), and then analyze the JDK separately (Section 142); finally, we discuss how we submitted some of WIT's inferred exception precondition as pull requests (Section 151).

Usefulness: Regular Projects

Let us first focus on the 46 projects in Table 5.1 excluding the JDK modules. We analyzed a subset sample of 517 expres and maybes that WIT correctly inferred for these projects; 72%

```java
public static void copyURLToFile(final URL source,
                                 final File destination,
                                 final int connectionTimeout,
                                 final int readTimeout)
  throws IOException {

  final URLConnection connection = source.openConnection();
  connection.setConnectTimeout(connectionTimeout);
  connection.setReadTimeout(readTimeout);
  copyInputStreamToFile(connection.getInputStream(), destination);
}
```

Listing 5.9. Implementation of `copyURLToFile()` in Commons IO's class `FileUtils`.

(374) of them are not documented; precisely, 242 of them belong to methods without any Javadoc, and 120 to methods with some Javadoc that does not describe that exceptional behavior. In contrast, 27% (138) of WIT's exception preconditions are properly documented; and 6% (29) of them are only partially documented (usually with a `@throws Exception` tag that does not specify the conditions under which an `Exception` is thrown).

Scenarios (such as the one in Section 5.2.2) where a method propagates an exception thrown by one of its callees may be hard to characterize precisely (especially when the callees' exceptional behavior is not documented); WIT's analysis can be particularly valuable in these cases. Indeed, 36% (187) of WIT's 517 exception preconditions analyzed in this section involve *nested* exception preconditions; only 24% (47) of these 196 exception preconditions are documented. This corroborates [24]'s finding that Javadocs rarely mention exceptions thrown by called methods.

Section 5.5.2's manual analysis of recall further surfaced evidence of WIT's practical usefulness. Even though the DSc dataset (which we used as ground truth to assess recall) is a paragon of comprehensiveness, WIT's modular analysis still managed to detect exception preconditions that were missed by DSc's painstaking manual analysis. Listing 5.9 shows Commons IO's method `FileUtils.copyURLToFile()`,{140} which calls methods from JDK class `URLConnection`{141},{142}. Commons IO's documentation of this method mentions five conditions under which the method will throw an `IOException`. The DSc dataset reports another two exception preconditions that trigger implicitly a `NullPointerException`. However, only WIT found that that the calls to `setConnectTimeout` and to `setReadTimeout` will throw an `IllegalArgumentException` if their argument is a negative integer. This is yet another example that manually detecting and documenting exception preconditions is tedious, time-consuming, and error prone; thus, the kind of automation provided by WIT can be very useful.

Usefulness: JDK Modules

We analyze the JDK separately, since it is arguably Java's most thoroughly documented library [95, 210]; therefore, it is natural to expect that a higher fraction of WIT's inferred exception preconditions will also feature in the JDK's official Javadoc documentation.

We analyzed a subset sample of 361 expres and maybes that WIT correctly inferred for the JDK; 38% (136) are not documented. We also found that 48% (172) of the 358 preconditions occur in nested calls (when an exception is propagated from a method call); and 61% (106) of them are documented, which is significantly higher than the ratio for the other projects.

Even though the JDK's documentation is generally outstanding, we found inconsistencies in when and how it documents exceptional behavior. For example, it sometimes only documents a subset of all possible unchecked exceptions a method may throw;{143} or occasionally uses the `throws` keyword to declare (unchecked) runtime exceptions.{144},{145} JDK's package `Time`{146} uses a distinctly different style of documenting `NullPointerException`s, which betrays the package's origins as a derivative of project `joda-time`; to declare that a method thows a null pointer exception when one of its parameters `p` is `null`, it writes: `@param p <description of p>, not null`.{147} Incidentally, project JFreechart uses a similar style of documentation.

Another interesting finding in the JDK is that older modules are more likely to neglect using exception *messages*—which, however, can provide valuable debugging information [126]. For instance, classes introduced in versions 1.0{148} and 1.1{149} seem to always instantiate `NullPointerException` without arguments (i.e., no message). Despite these outliers, the JDK generally tries to use expressive exception messages, and to improve their clarity. For example, `Integer.parseInt` throws a null pointer exception with an uninformative message `"null"` in JDK 11;{150} in JDK 17, however, the maintainers changed it to the more informative `"Cannot parse null string"`.{151}

In a manually analyzed sample, 38–72% of WIT*'s exception preconditions were* ***not*** *documented.*

Improving Project Documentation Using WIT

While there may be situations where documenting every source code method is not needed or recommended, properly documenting *public* methods of APIs (remember that all of WIT's exception preconditions refer to public methods) is an accepted best practice [150, 210]. Indeed, there is evidence that several of the projects used in our evaluation (Section 5.4.1) routinely improve their Javadoc documentation of exceptions,{152},{153} and often recommend{154} or even require{155},{156} accurate Javadocs in any code contributions. To determine whether WIT's inferred preconditions can be a valuable source of API documentation, we collected 90 exception preconditions extracted by WIT in 5 Apache projects and submitted them as 8 pull requests (as described in Section 5.4.2). At the time of writing, maintainers accepted (without modifications) 6 pull requests containing 81 preconditions—63 (78%) of them occurring in nested calls. Two pull requests to project Commons Math have not been reviewed yet. Interestingly, one to project Commons Lang was on hold for several months because the project maintainers realized that the 10 methods whose exceptional behavior we document are inconsistent in using `IllegalArgumentException` vs. `NullPointerException`, and they preferred to fix this inconsistency before updating the documentation.

When submitting our improvements to project `Commons Lang`, we opened a JIRA issue{157} sharing our findings. Several months after our initial pull request, a GitHub user submitted four Javadoc modifications in a new pull request{158} that mentioned our JIRA issue. Shortly afterwards, a `Commons Lang` maintainer asked us to review the modifications in the new pull request, and suggested that we submit all our findings (i.e., all the exception preconditions that could be included in the documentation) in order to close the JIRA issue. In the end, we worked together with the author of the latest pull request to submit 89 WIT exception preconditions (27 new pieces of Javadoc documentation and 62 fixing existing documentation), as well as tests for 9 classes. All of the exceptions from the additions and fixes occur in nested calls, which may explain why they went undetected for a long time. The pull request was accepted in the same day and merged ten days later.

Overall, our 9 pull requests (8 initial ones, plus the latest one suggested by the maintainers) include 189 exception preconditions (90 in the initial batch, and 89 in the latest one). These pull requests contain 157 (88%) preconditions occurring in nested calls; 61 (34%) that refer to missing documentation, and 118 (66%) that target a wrongly documented exception. A total of 170 preconditions (81 in the initial batch, and 89 in the latest one—or 95% of all those submitted) were merged into the projects' official documentation. It is significant that the projects that accepted these pull requests are known for their extensive and thorough documentation practices [150, 207]. The fact that WIT could automatically detect several exception preconditions that were missing from their documentation, and promptly added following our pull requests,[18] indicates that WIT's output can be quite useful. We expect that WIT's precise output can have an even bigger impact on scarcely documented projects.

WIT's precise exception preconditions can be useful to improve also large and mature projects: maintainers from 4 Apache projects accepted 95% of a sample of WIT preconditions submitted as pull requests.

5.6 Threats to Validity

The main threat to the *internal validity* of our assessment of WIT's *precision* (Section 5.5.1) comes from the fact that it is based on manual inspection of Java code and documentation. Like all manual analyses, we cannot guarantee that no mistakes were made. Nevertheless, various evidence corroborates the claim that WIT's precision is high. First, WIT's precision follows from its design; therefore, the manual analysis was primarily a validation of WIT's *implementation*, checking that no unexpected source of incorrectness occurred in practice. Second, we inspected not only the source code but also any official documentation, tests, as well as the datasets of related studies of Java exceptions [126, 150]. Third, the authors extensively discussed together the few non-obvious cases, and were as conservative as possible in the assessment. We followed similar precautions to mitigate threats to our assessment of WIT's *recall* (Section 5.5.2), where we relied on [150]'s and [207]'s manual analyses as ground truth.

[18] One maintainer from Accumulo remarked that ours "are nice fixes to the javadoc, thanks for finding them."

```java
 * @throws IllegalArgumentException if {@code src} is empty,
 * {@code src.length > 8} or {@code src.length - srcPos < 4}
 * @throws NullPointerException if {@code src} is {@code null}
 */
static char binaryToHexDigitMsb0_4bits(boolean[] src, int srcPos) {
  if (src.length > 8) {
      throw new IllegalArgumentException("src.length > 8");
  }
  if (src.length - srcPos < 4) {
      throw new IllegalArgumentException("src.length - srcPos < 4");
  }
  if (src[srcPos + 3]) {
      // ...
```

Listing 5.10. Excerpt of a method and its Javadoc from class `Conversion`[160] in project Commons Lang.

As customary [210], we assume that the implementations of all analyzed methods are correct: WIT's goal is to capture an implementation's exceptional behavior as faithfully as possible; detecting bugs in such implementations is out of its (current) scope.

Our selection of 46 Java projects includes several very popular Java open source libraries, which were used in recent related work, and in addition several modules in Java's official JDK; this helps reduce threats to *external validity*. It remains that the exceptional behavior of libraries may be different than that of other kinds of projects. Since library APIs tend to perform more input validity checks [169], it is possible that WIT would report fewer exception preconditions simply because fewer are present in other kinds of software. Indeed, a handful of the projects with the smallest number of reported expres turned out not to be libraries (see Table 5.1).

As one of the ground truths to estimate recall, we used a recent survey [150] that extensively manually analyzed a single project (Commons IO). As we discuss in Section 5.5.2, the nature of this project makes it especially challenging for WIT, which implies that its recall may be higher on other projects (as the experiments using the other dataset DPA [207] suggest).

WIT's implementation has a number of limitations; some reflect deliberate trade-offs, while others could simply be removed by extending its implementation. In its current state, WIT has demonstrated to produce useful output and to be precise and scalable.

5.7 Discussion of Applications

This section outlines possible applications of WIT's technique that take advantage of its characteristics. WIT's precision is especially handy when generating documentation (discussed in Section 5.7.1) or tests (Section 5.7.2). WIT's other key feature (that it's *lightweight*) helps apply it to different scenarios. For research in mining software repositories, not requiring complete project builds enables scaling analyses to a very large number (e.g., several thousands) of projects—whereas building all of them would be infeasible [80]. Using WIT as a component of a recommender system that runs in real-time is another scenario where speed/scalability would be of the essence.

5.7.1 Documentation

As we demonstrated in Section 151, the output of WIT's analysis can be useful to extend, complement, and revise the documentation of public methods' exceptional behavior. Accurately documenting exceptions is crucial for developers [210], but writing documentation is onerous [150, 151]; as a result, APIs often lack documentation [169], especially for exceptions [24]. WIT's high *precision* ensures that its output can generally be trusted without requiring manual validation, and hence it can directly help the job of developers writing documentation (or tests).

In most cases, WIT's exception preconditions are in a form that can be easily transformed into method documentation—for example by expressing them in natural language using pattern matching [20, 75, 210]. In fact, since it uses precise static analysis, we found several cases where WIT's exception preconditions provide more rigorous information than what is available in programmer-written documentation. For example, Listing 5.10 shows the programmer-written exceptional behavior documentation and the initial part of the implementation of a method from class `Conversion` in project Apache Commons Lang. WIT outputs two exception preconditions for the method:

$$\texttt{src.length > 8} \tag{5.1}$$

$$\texttt{src.length <= 8 \&\& srcPos - src.length > -4} \tag{5.2}$$

both corresponding to an `IllegalArgument` exception. At first sight, it may seem that WIT's output is incomplete (it doesn't mention the preconditions "`src` is empty" and "`src` is `null`" in the Javadoc) and needlessly verbose (isn't `src.length <= 8` redundant?). A closer look, however, reveals that several aspects of the natural-language documentation are questionable or inconsistent. First, it mixes explicitly and implicitly thrown exceptions: a `NullPointer` exception is thrown by the Java runtime when evaluating the expression on line 6, not by the method's implementation. WIT ignores such language-level exceptions by design; as we mentioned in Section 5.3.7, not including implicit exceptions in API documentation may be preferable [24, 66]. A second issue with Listing 5.10's documentation is that it is incorrect: if `src` is empty, the method does not throw an `IllegalArgument` exception; instead, the Java runtime throws an `IndexOutOfBounds` exception at line 12 (another system-level implicit exception). Finally, Listing 5.10's documentation is inconsistent regarding the *order* in which the various exception preconditions are checked: whether `src` is `null` is checked first (implicitly), then `src.length > 8` (explicitly), `src.length - srcPos < 4` (explicitly), and whether `src` is empty (implicitly)—in this order. Thus, `src.length <= 8` in WIT's second inferred preconditions is not redundant but rather useful to ensure that the precondition precisely captures the conditions under which a certain path is taken. Admittedly, WIT may sometimes present preconditions in a form that is harder to understand for a human; for example, it is questionable that the "simplification" of `src.length - srcPos < 4` into `srcPos - src.length > -4` improves readability. However, these are just pretty-printing

```
 1  Fraction getFraction(final int whole, final int num, final int den) {
 2    if (den == 0) throw new ArithmeticException("The denominator must not be zero");
 3    if (den < 0) throw new ArithmeticException("The denominator must not be negative");
 4    if (num < 0) throw new ArithmeticException("The numerator must not be negative");
 5    final long nv;
 6    if (whole < 0) { nv = whole * (long) den - num; }
 7    else { nv = whole * (long) den + num; }
 8    if (nv < Integer.MIN_VALUE || nv > Integer.MAX_VALUE)
 9      throw new ArithmeticException("Numerator too large to represent as an Integer.");
10    // ...
11  }
```

Listing 5.11. Simplified excerpt of method `Fraction.getFraction` from in project Commons Lang.

details that are currently left to SymPy; changing them to generate constraints that follow certain preferred templates could be done following Nguyen et al.'s [151] approach. In fact, one could even let the user decide the output format according to their preference. Overall, this example demonstrates that WIT's output often has all the information needed to generate accurate documentation that avoids ambiguities or other inconsistencies.

5.7.2 Generating Tests

Automatically generating tests that exercise a method's exceptional behavior is another natural applications of WIT. Fully pursuing it is outside this work's scope; nevertheless, we briefly discuss this directions on a few concrete examples that we encountered while carrying out Section 5.4's empirical evaluation.

As mentioned in Section 5.3, each exception precondition reported by WIT also comes with an example of inputs that satisfy it; for instance, for exception precondition (5.2), WIT outputs the example `[src.length=2, srcPos=0]`. Writing a test that initializes an array with two elements, calls the method in Listing 5.10, and checks that an `IllegalArgumentException` is thrown (and that it contains a specific message) is straightforward. In fact, one could even try to automate the generation of tests and oracles from WIT's examples and preconditions. For example, using property-based testing [36]: after expressing (5.2) (or even the specific example) as an input property, let a tool like jqwik{161} randomly generate inputs that satisfy it.

The information captured by WIT can support increasing the level of automation and generally make programmers more productive. It can also improve the quality of the tests that are written, as demonstrated by the following example. Listing 5.11 shows a (simplified) excerpt of method `Fraction.getFraction` in Apache Commons Lang, which takes three integers `whole`, `num`, `den`, and returns an object representing the fraction `whole+num/den`. As we can see in Listing 5.11, `getFraction` has 4 exception preconditions: (a) (line 2) when `den` is 0; (b) (line 3) when `den` is negative; (c) (line 4) when `num` is negative; (d) (line 8) when the resulting numerator `nv` exceeds the largest integer in absolute value. Commons Lang is a thoroughly tested project [150], and in fact all four exceptional behaviors are tested.{162} The 4 behaviors are not evenly tested though: 3 calls cover (a), 6 calls cover (b) (including

```
 * @throws StringIndexOutOfBoundsException if {@code offset} is not in the
 * range {@code 0 <= offset <= chars.length}
 * @throws StringIndexOutOfBoundsException if {@code length < 0}
 * @throws StringIndexOutOfBoundsException if {@code offset + length > chars.length}
```

Listing 5.12. Documentation of `StringSubstitutor.replace()` submitted as pull request in project Commons Text.

three identical calls, which is likely a copy-paste error), 1 call covers (c), and 4 calls cover (d). Comments in the test method which refer to the four categories are sometimes misplaced (for example, two calls under "zero denominator" actually cover (d)). In contrast, WIT's example inputs correspond one-to-one and uniquely to each exception precondition: (a) `den=0`; (b) `den=-1`; (c) `num=-1, den=1`; (d) `whole=2147483648, num=0, den=1`. If we wanted multiple example inputs for the same precondition, we could just ask Z3 to generate more. In all, WIT's output can be quite useful to guide a systematic test-case generation process.

Another situation where WIT's output helps write tests that exercise exceptional behavior is when this requires a combination of inputs for different arguments. One example is Commons Text's method `FormattableUtils.append()`,[163] which takes 6 arguments and comes from Java's `Formatter` interface.[164] `FormattableUtils.append()`'s exception precondition involves the negation of a disjunction of three Boolean predicates: `!(e == null || p < 0 || e.length()<= p)`. WIT suggests an input where `e.length()` is 1, and `p` is 0, which is easy to implement as a test. Another example is method `StringSubstitutor.replace()`[165] in the same project, which takes three arguments (one character array and two integers) and may throw an exception in a nested call. As regularly seen in Apache Commons projects, the method accepts null or empty arrays; however, when the array is non-null, the exception precondition gets quite complex. WIT provides exception triggering inputs for the three arguments, including that the character array must not be null and could be empty. In cases like this, we could reuse parts of WIT's extracted precondition to document the complex exception condition. The complexity of the precondition, together with it being in a nested call, may be the reason why the documentation and tests were missing in the project.

5.8 Conclusions

We presented WIT: a static analysis technique to extract exception preconditions of Java methods. WIT focuses on precision: it only reports correct preconditions.

An evaluation on 46 open-source Java libraries and five JDK 11 modules demonstrated also that it is lightweight (under two seconds per analyzed public method on average), precise (all inferred preconditions are correct), and can recover a significant fraction of the known exception preconditions (9–83% of the supported exception preconditions using [150]'s manual analysis as ground truth).

While the exception preconditions detected by WIT tend to be syntactically simple, they often complement the available documentation of a method’s exceptional behavior, as we demonstrated by merging a selection of 170 inferred exception precondition as pull requests in the projects’ open source repositories.

In order to combine scalability and applicability, WIT can perform a modular analysis: after inferring the exception preconditions of a project `A`, it can use them to analyze the behavior of another project `B` whenever it calls out to any methods in `A`. Our empirical analysis suggested that modular analysis is a bit of a mixed bag: it does increase the number of exception precondition WIT can detect, but it may also decrease the precision for the so-called “maybes”—exception preconditions that are reported separately, as WIT could not conclusively establish that they are correct. Accordingly, WIT can be configured to use modular analysis selectively, according to what is the main goal of its users. Investigating heuristics to help the automatic selection of these configuration options is an interesting direction for future work.

Artifacts: the complete artifacts to support the replication of our experiments are available: `https://doi.org/10.6084/m9.figshare.22217014`.

6

Towards Code Improvements Suggestions from Client Exception Analysis

Modern software development heavily relies on reusing third-party libraries; this makes developers more productive, but may also lead to misuses or other kinds of design issues. In this chapter, we focus on the *exceptional* behavior of library methods, and propose to detect *client* code that may trigger such exceptional behavior. As we demonstrate on several examples of open-source projects, exceptional behavior in clients often naturally suggests *improvements* to the documentation, tests, runtime checks, and annotations of the clients.

In order to *automatically detect* client calls that may trigger exceptional behavior in library methods, we show how to repurpose existing techniques to extract a method's exception precondition—the condition under which the method throws an exception. To demonstrate the feasibility of our approach, we applied it to 1,523 open-source Java projects, where it found 4,115 cases of calls to library methods that may result in an exception. We manually analyzed 100 of these cases, confirming that the approach is capable of uncovering several interesting opportunities for code improvements.

Structure of the Chapter

- Section 6.1 provides motivation for this chapter.
- Section 6.2 describes our approach.
- Section 6.3 describes our preliminary experiments.
- Section 6.4 draws our conclusions and outlines ideas for future work.

6.1 Introduction

Any piece of client code that calls a library method must comply with the method's *precondition*. Thus, analyzing method calls against the callees' preconditions can reveal issues with how a library is used, and possibly suggest useful improvements to the client code.

A widespread scenario occurs when a library method may throw an *exception*; for example, to signal that one of its arguments should not be `null`. If we can show that a client *never* calls the method with a `null` argument, or suitably handles the exception (for example, with a `try`/`catch` block), we rule out a certain category of faults. Conversely, if we find a concrete client execution where the exception is raised and not handled, this suggests a number of improvements to the client code, such as adding tests or documentation for this possible exceptional behavior, or perhaps modifying the client so that it handles the exception directly. As we discuss in Section 6.2 on concrete examples of Java code, such scenarios of client code calling library methods that may throw exceptions are quite common in open-source projects; analyzing them systematically (and automatically) has the potential of revealing interesting instances of misuses and critical cases, as well as of suggesting ways of *improving* the client code to address the issues.

Unfortunately, there are two main obstacles that stand in the way of practically pursuing this idea of *analyzing exception preconditions of library methods in client code to suggest code improvements to the client*. First, (exception) preconditions are often documented only informally (e.g., using natural-language comments) and partially [96, 173]. Second, even if we have a formula precisely expressing a library method's exception precondition, determining whether a call to the method may actually raise an exception requires precise reasoning about the client code (for example, through symbolic execution), which remains challenging to carry out on code bases of realistic size and complexity.

We discuss a practical approach that can deal with these two difficulties. To this end, it leverages recent work on automatically *extracting* exception preconditions in a way that is scalable (applicable to realistic projects) and precise (always returns correct exception preconditions) [127, 152, 201]; most notably, this includes our work on WIT, which we presented in Chapter 5. Running these tools on widely used Java libraries and frameworks (including a substantial portion of the JDK) populates a database of exception precondition Boolean formulas, which precisely indicate under what conditions calling a certain library method results in an exception. To pursue our approach, we then discuss how to repurpose these existing detection techniques so that they can run on *client* code and find *feasible matches* of any exception preconditions in their database; in other words, they detect calls to any of the methods with an exception precondition that may result in an exception being thrown.

Section 6.3 describes preliminary experiments that we conducted to assess the practical feasibility of our approach. Among the aforementioned tools for exception precondition detection we used WIT [127] for our experiments. First, we selected 1,523 open-source Java projects that use some of the libraries that can be analyzed with WIT. Then, we modified WIT to detect feasible matches of exception preconditions—that is, possible exceptional behavior in the clients—and ran it on the selected projects. We found 4,115 such matches, which indicates that our analysis is widely applicable. We also manually analyzed a random sample of 100 matches, in order to better understand what kinds of issues the matches reveal, and how they could be turned into actionable suggestions for improvements to the client code, its tests, or its documentation. We report several concrete examples taken from open-source Java projects, which lend weight to this chapter's core idea: identifying possible improvements to client code by automatically analyzing the exception preconditions of

```java
// @throws IllegalArgumentException if bound is not positive
public int nextInt(int bound) {
  if (bound <= 0) throw
    new IllegalArgumentException("bound must be positive");
  // ...
}
```

Listing 6.1. Documentation of `java.util.Random.nextInt(int)`.

```java
public static int random(final int min, final int max) {
  return Utils.RANDOM.nextInt(max - min) + min;
}
```

Listing 6.2. Client code calling Listing 6.1's method.

library methods.

While there is plenty of related work about analyzing exceptional behavior and detecting API misuses, the combination of the two concepts has hardly systematically been explored.

6.2 From Exception Preconditions to Code Improvements

Our main idea is an approach to automatically analyze client code for potential throws of exceptions in library methods. Precisely, a method `m`'s *exception precondition* is a Boolean condition $P_{\mathtt{m}}$ under which the method terminates with an exception. Conversely, a *potential throw* ("*pothrow*" for short) is a piece of client code with a call to `m` whose actual arguments *may* match `m`'s exception precondition $P_{\mathtt{m}}$, and that *does not* handle the corresponding exception. Potential throws point to client code that may not fully conform to the library's (exceptional) specification—a possible case of design issues or even misuses. Section 6.3 describes some experiments supporting our hypothesis that pothrows in real projects can indeed suggest code improvements and refactoring. The rest of this section outlines an approach to detect pothrows automatically.

6.2.1 An Example of Potential Throw Detection

Listing 6.1 shows Java's `java.util.Random.nextInt(int)`,[166] a library method that throws an `IllegalArgumentException` (IAE) if its argument is strictly less than one. Even a basic exception precondition like `nextInt`'s (explicitly documented in the method's Javadoc natural language documentation) may be non-trivial to handle properly for clients. For example, consider method `random` in project Zelix Injection;[167] as shown in Listing 6.2, `random` is a pothrow of `nextInt`: if `max` $\leq$ `min`, the call to `nextInt` fails with an uncaught IAE whose error message is not very informative in the client's context.

```java
// @throws IllegalArgumentException if @code\{max <= min\}
public static int random(final int min,
  @Refinement ("max > min") final int max) {
    Validate.isTrue (max > min, "max <= min");
    return Utils.RANDOM.nextInt(max - min) + min;
}

@Test
void random_throws_IAE() {
  assertThrows(IllegalArgumentException.class,
    () -> Utils.random(1, 1);
  // ... other cases
}
```

Listing 6.3. Possible improvements to Listing 6.2's code (in color).

6.2.2 Code Improvements

Even though code including potential throws might be perfectly correct, more commonly it indicates possible design issues, which, in turn, may suggest improvements to the code, its documentation, or its tests that increase its quality for its own clients and for the whole project. In fact, there is evidence that exceptional behavior is often insufficiently documented and tested even *within* a project [126, 207]; the same issues are likely to intensify when considering a project's *clients*.

Listing 6.3 shows four possible improvements for Listing 6.2's pothrow. *Documenting* the derived exception precondition of `random` with a `@throws` tag helps its users know when to expect an exception. *Argument checking* (using `Validate.isTrue` in Listing 6.3) performs a runtime check that `random`'s actual arguments will not trigger an exception; this follows the *fail fast* principle, signaling precisely the condition and location of the exception when one occurs. Extended *type annotations* (using Liquid Java's `@Refinement` annotation in Listing 6.3) go one step further as they support checking for possible exceptional behavior *at compile time* using tools such as the Checker Framework [158] and Liquid-Java [69]. Providing *tests* that exercise exceptional behavior (through JUnit's `@Test` in Listing 6.3) also helps code quality, as it provides means to detect possible regressions, and serves as a concrete counterpart to the method's documentation.

6.2.3 Detecting Potential Throws Automatically

In order to automatically find instances of pothrows, we propose an approach in three steps. First, we collect exception preconditions of library methods; to this end, we can use any recently developed static techniques [127, 152, 201] that are applicable to realistic projects and return *correct* exception preconditions (if the preconditions may be incorrect, the whole analysis would become noisy and imprecise).

Second, we analyze *clients* of the libraries, looking for *calls* to any of the library methods for which exception preconditions are available. Ideally, this step would be performed *without* fully building the client code, so that the analysis is more lightweight and can also target parts of a project. Here too, we privilege precision (every match is a real match) over recall (all possible matches are detected).

Third, we determine whether the arguments of any calls identified in the previous step may actually satisfy any of the available exception preconditions. This amounts to a *feasibility* check that finds a condition over the client's arguments (such as `max` ≤ `min` in Listing 6.2) that triggers the exception. In general, the feasibility check requires precisely reasoning about client code at its call locations. For example, if method `random` in Listing 6.2 called `nextInt` with argument `1 + Math.max(min, max)- Math.min(min, max)`, it should recognize that this expression is always positive, and hence `nextInt` will not throw. The feasibility checks should be precise as well (since we do not want to report many false alarms), but they should also achieve a reasonable recall—otherwise, the analysis would produce hardly any output. To perform the feasibility check, we encode it as a modular variant of the same exception precondition detection performed in the first step: given a piece c of client code calling library method `m`, determine c's exception precondition using `m`'s. Any such exception preconditions of c are reported as *pothrows*.

6.3 Experimental Evaluation

In this section, we first discuss our prototype implementation of our approach to detect potential throws of library exceptions in client code (Section 6.3.1); then, we present the design (Section 6.3.2) and quantitative results (Section 6.3.3) of an empirical evaluation on several open-source Java projects; finally, we discuss several interesting cases that emerged in these experiments, which we manually inspected to validate the approach and to illustrate its practical usefulness (Section 6.3.4).

6.3.1 Potential Throw Detector Implementation

Among the available techniques for exception precondition detection, we used WIT [127] as the basis for our implementation. When run on a library, WIT returns two kinds of exception preconditions—called *expres* and *maybes* in [127]. For our work, we only consider the former, which pass a path feasibility check, and hence are *correct* by construction.

First, we added support to store in a database the exception precondition WIT collects over multiple runs, so that they can be queried by library and method signature. Second, we wrote a simple program that uses JavaParser{168} to scan through a project and resolve the fully qualified names of any called library methods, and then searches the database of exception precondition for any match of these called methods. Third, we modified WIT so that it analyzes any enclosing method in the client that includes a call to one of the matching library methods; WIT determines whether the callee's exception can be propagated to the caller (the client) and under which conditions; in other words, it reports potential throws (pothrows) in the client. Again, we enable WIT's feasibility checks, so that it only reports pothrows that are indeed feasible.

6.3.2 Empirical Study: Design

We ran an empirical study to confirm that our approach is applicable to realistic projects, that it can identify a significant number of pothrows, and that several of these pothrows are indicative of design issues—and potential code improvements.

First, we selected 21 widely used open-source Java libraries including 6 analyzed in WIT's original work [127] (joda-time, and Apache Commons Lang, IO, Text, Configuration, and Math), as well as 15 new ones (Java 11's[1] JDK, Apache Commons Codec and Collections, Eclipse Collections, ehcache3, gson, Guava, hibernate-orm, jaxb-ri, jsoup, retrofit, and Spring boot, data-jpa, framework, and security).

We also selected several *client* projects from two different sources. Using the GHS search tool [42], we gathered 1,312 Java (non-fork) projects on GitHub with at least 10 stars and a thousand lines of code. We did not perform any a priori check that these projects use any of the 21 libraries we considered; however, it's overwhelmingly likely that these projects at least use some JDK library classes (e.g., `String`). To further increase the diversity of client projects, we also gathered another 220 client projects from the DUETS dataset [57], which consists of library/client pairs among Java open source projects developed with Maven; we specifically collected all client projects that use the latest version of joda-time, jsoup, and all Apache Commons projects we analyzed. In the following, GHS denotes the first batch of 1,312 projects, and DUETS the second batch of 220 projects.

Finally, we selected 100 pothrows among those reported in all projects and analyzed them manually. This sample of pothrows corresponds to 2.5% of all the 4,115 pothrows reported by our tool (see Section 6.3.3). This is a reasonable sample size for an exploratory study, given that manual checks like this can be very time-consuming [127, 150]: they took the first author more than eight hours. We sampled opportunistically, trying to cover several different libraries, called methods, and library projects. The manual analysis was, first of all, a sanity check to confirm that the pothrows are correct (i.e., they identify method calls that *may* throw an exception). Most of the times, confirming the correctness of a pothrow was straightforward (e.g., a possible null argument), and required only a cursory analysis of the call context. For more complex cases (e.g., a call to `StringBuilder.append`{169} in project feathersui-starling-sdk{170} with arguments `empty array`, `2`, and `-1` throws an `IndexOutOFBoundsException`), we inspected the code more extensively using `jshell`.{171} After the sanity checks, we also thought about what kinds of code improvements the manually analyzed pothrows suggest; Section 6.3.4 presents a few selected interesting examples.

6.3.3 Empirical Study: Quantitative Results

Running WIT on the 21 selected libraries populated our database with 14,180 exception preconditions of 10,204 public library methods. The analysis of the 1,312 GHS client projects

[1]We focus on Java 11 because it's the latest Java LTS version that JavaParser fully supports.

found 106,345 calls to 1,961 of the analyzed library methods. The analysis of the 220 DUETS client projects found 28,324 calls to 806 of the analyzed library methods. Overall, we found 134,579 calls matching 1,961 of the analyzed library methods (i.e., the called methods in the DUETS batch are a subset of the called methods in the GHS batch). Running our modified version of WIT on the code snippets surrounding each of these 134,579 client calls identified 4,115 pothrows (2,885 in the GHS projects and 1,260 in the DUETS projects)—around 3% of the client calls. We confirmed that all the 100 pothrows we manually analyzed were correctly identified by the tool.

We can think of a possible explanation for why only a fraction of all matching calls are pothrows. Precondition inference techniques like WIT trade off recall for precision [127]; in our experiments, the modified WIT only reports a call as pothrow if it can conclusively establish that the call is feasible, which may miss some real instances. (In fact, its original evaluation [127] indicates that WIT's recall can dip below 10% on some projects.) Regardless, it is also reasonable to expect that a large fraction of library method calls are set up by the client to comply with the library's preconditions or are within a `try` block—and thus, they never result in an exception.

Table 6.1 gives an overview of ten of the most frequently called library methods among those we considered in our experiments; all of them are to JDK methods. In fact, it is clear that JDK methods dominate both the matching calls and the pothrows: overall, only 4.7% (6,371) of all calls, and 9.5% (387) of all pothrows refer to methods in libraries *other* than the JDK. Even though the DUETS projects should focus on non-JDK libraries, they still use plenty of JDK libraries: among DUETS projects, 96% (27,092) of calls, and 92% (1,164) of pothrows, refer to some of 614 JDK library methods; among GHS projects, 95% (101,206) of calls, and 90% (2,564) of pothrows, refer to some of 1,758 JDK library methods. Overall, the 6,371 calls and 387 pothrows involve only 1,007 non-JDK library methods; just three of these libraries (Apache Commons Lang, Guava, and Spring framework) account for 998 calls and 93 pothrows of 29 argument-checking library methods.

In hindsight, JDK's dominance is not surprising. First, virtually every project—even if it uses other common libraries—is a client of the JDK. Second, just because a project declares a certain library as a dependency does not mean that it uses it extensively; in fact, it may not use it at all: Harrand et al.'s empirical study [78] found that 41% of declared project dependencies do not correspond to any API usages at the bytecode-level. The study also found that, for more than half of the 94 analyzed libraries, 75% of the clients use only 12% of the libraries' methods; thus, expecting a much larger number of pothrows in our experiments is unrealistic.

6.3.4 Empirical Study: Qualitative Discussion

Examples of Potential Throws

`java.util.ArrayList`'s constructor throws an `IllegalArgumentException` if the given initial capacity is a negative number. As shown in Table 6.1, calls to this method are common in our client projects; 140 of them are pothrows, which happen when the actual argument is an expression that may be negative. (None of these pothrows is a sure bug, i.e., none

Table 6.1. Ten of the most widely called library methods in our experiments. For each LIBRARY METHOD, the table reports the number of CLIENT projects with at least one call to the method, the total number of CALLS to the method, and how many of the calls are POTHROWS (potentially throwing).

LIBRARY METHOD	CLIENTS	CALLS	POTHROWS
`ArrayList.ArrayList(int)`	351	3 446	140
`ArrayList.get(int)`	333	8 538	11
`File.File(String)`	610	7 597	660
`Integer.parseInt(String)`	589	7 359	62
`Objects.requireNonNull(T)`	156	2 292	832
`Objects.requireNonNull(T, String)`	89	1 002	643
`Optional.of(T)`	166	1 234	36
`Random.nextInt(int)`	296	2 769	38
`String.substring(int)`	640	6 285	12
`String.String(char[], int, int)`	81	304	60

```java
// @throws IllegalArgumentException if there are
// more columns requested than the dimension
public static List<Vector> getBasis(int dim, int nCols) {
  if (dim < nCols)
    throw new IllegalArgumentException(msg);
  List<Vector> basis = new ArrayList<Vector>(nCols);
  // ...
}
```

Listing 6.4. Pothrow call to `ArrayList`'s constructor.

of them passes a negative *literal* to `ArrayList`.) Listing 6.4 shows an interesting case from project SuanShu, involving two arguments of public static method `getBasis`.{172} The client method first checks the precondition `dim` ≥ `nCols`, and then calls `ArrayList`'s constructor with argument `nCols`; thus, if `dim < 0`, the constructor's exception will propagate to the client. Perhaps `dim`, which should denote a dimension, is supposed to always be a nonnegative number; if this is the case, project SuanShu could benefit from making this assumption explicit using a combination of the code improvements outlined in Section 6.2: adding documentation, argument checking, extended type annotations, and tests to boot.

Here is another piece of evidence in support of our hypothesis that automatically analyzing potential throws can reveal subtle semantic differences between different clients and libraries. The constructor of `LinkedBlockingDeque`,{173} another `java.util` data structure, throws an exception if its initial capacity argument is negative *or zero*. Indeed, we found two pothrow calls that may violate this constraint in our analyzed projects.{174},{175} Interestingly, whereas `LinkedBlockingDeque` implements interface `Deque`, other implementations of the same interface may have different exception preconditions; for example, `ArrayDeque` robustly accepts any value as initial capacity, and simply resets it to one if given a zero or

```java
// @param title the chart title ({@code null} permitted).
// @param plot the plot ({@code null} not permitted).
public JFreeChart(String title, Plot plot) {
  // ...
}
```

Listing 6.5. The signature and header comment of `JFreeChart`'s constructor in client project JFreeChart.

```java
private void add(String key, String value) {
  properties.setProperty(key, value);
}
```

Listing 6.6. Private method calling JDK's `Properties.setProperty`.

negative number.{176} Thus, its clients cannot incur any pothrow when constructing instances of `ArrayDeque`.

It is well known that null pointer derefencing—signaled by `NullPointerException` (NPE) in Java—is a widespread problem in programming languages where they can happen [82]—and one that prompted numerous attempts at mitigating it [55].{177} Analyzing some of the numerous instances of pothrows that may result in a NPE, we realized that the problem is compounded when `null` is a perfectly valid value for some arguments but not for others. Take the constructor of class `JFreeChart`{178} from the homonymous project, whose signature is shown in Listing 6.5: argument `title` may be `null` (denoting an empty title), whereas argument `plot` results, through an indirect check, in a IAE if it is `null`. This instance of pothrow is already explicitly documented in `JFreeChart`'s constructor; but it still highlights the usefulness of an automated analysis that can follow third-party library dependencies and disentangle different valid usages of a method.

Besides, not all methods are as accurately documented as Listing 6.5's constructor. Consider, for example, method `Properties.setProperty(String, String)`{179} in JDK's package `java.util`, which throws a NPE if any of its two arguments is `null`. Despite being widely used (we found 1,545 calls to it in our projects), method `setProperty`'s documentation does not mention its exception precondition. In fact, we found 12 pothrows that involve calls to `setProperty`. Listing 6.6 shows one of these cases from project javapos_shtrih: a wrapper of `setProperty`,{180} which thus has the same exception precondition as pothrow. If we want to think about possible code improvements in this case, it is important to notice that `add` is

```java
private static CacheStatus getCacheStatus(
  @NonNull LibraryCachingConfiguration cachingConfiguration,
  @NonNull final FilePath versionCacheDir) {
  // ...
}
```

Listing 6.7. An example of a private method annotated with `@NonNull`.

private. While it might be called with a null argument, all its calls *within* the project supply non-null arguments; the project developers may have certain guidelines on how (and if) such cases need documentation or other kinds of annotations. In general, however, extended type checking annotations are often applied to private methods as well—for the few projects that make the effort of producing and maintaining such annotations; for example, Listing 6.7 shows a snippet[{181}] from a project part of the popular `Jenkins` automation server which systematically annotates its methods (also private ones) with `@NonNull` annotations.

We could say that the punch line of all these examples is that "knowing is half the battle": once they identify concrete cases of potential throws, developers can exercise their judgment, preferences, and project knowledge to come up with the most suitable code improvements (including the conscious decision to leave everything as it is). The "knowing" part, however, is not straightforward without adequate tool support, as it is well know that exceptions are often undocumented [96, 97], and even the few projects that pride themselves of exhaustively documenting and testing their APIs may miss some corner cases [127].

Another point worth mentioning is that there is usually a certain latitude in what code improvements can be applied, but even seemingly minor changes may be beneficial—especially with exceptional behavior, where often "failing fast" is the best one can do. For example, the authors of JSpecify remark that "what it prevented from happening could have been much worse.".[{182}] Another example are exception handling guidelines that recommend to catch an exception, logging, and finally rethrowing it [137]; the exception may be unavoidable in the end, but the countermeasures help a lot with debugging and analysis. Yet another example is the approach "convert library exceptions" [137], which recommends to wrap any exception received from a third-party library into an exception defined by the client application before propagating it; here too, the exceptional behavior is not suppressed but it surfaces in a way that is less surprising and, arguably, easier to use.

6.4 Conclusions

Two main directions to mature this idea of analyzing exception preconditions of library methods in client code to suggest code improvements are: 1) improving the quality and quantity of exception preconditions; and 2) automating the generation of code improvement suggestions. Direction 1) is motivated by findings that a small portion of a library is generally responsible for a large portion of client usage [78, 109]; and can naturally lead to progress in direction 2) by specializing the suggestions to cover the most common cases. Finally, applying our ideas to different benchmark collections [3, 96] is also a natural way to further validate them.

Dataset: the complete dataset of our experiments, including links to code snippets, is available: `https://doi.org/10.6084/m9.figshare.23634747`.

Part IV

Epilogue

7 Conclusions

In this thesis, we presented our research on practical program analysis for improving Java software. To maximize practicality, we tried to minimize assumptions and requirements when analyzing source code, and focus on amenable, yet frequent occurring, problems. In this vein, our research was organized into two topics namely i) repairing static analysis violations, and ii) analyzing exception behavior. With the first, we can help developers using tools that are already integrated into their workflows, but, unfortunately, lack automated fixes and may report too many warnings. With the latter, we aid developers on dealing with a common cause of failures and anti-patterns: exceptions. Exception-handling code is often complex and among the most poorly understood and scarcely documented parts of a system. The literature shows that exceptions are commonly implicated in bugs of Java libraries, Android apps, and cloud systems.

For repairing static analysis violations, we developed SpongeBugs, a static technique that can produce fixes for Java violations of simple rules detected by widely used static analyzers SonarQube and SpotBugs. The technique scales to realistic, popular, and large open-source projects and generates patches that developers find acceptable. Maintainers of popular Java open-source projects accepted 87% of 946 fixes generated by SpongeBugs.

We followed multiple directions to analyze Java exception behavior. First, we gained valuable insights into the context of frequently thrown exceptions by looking into how Java developers test exceptions with the JUnit framework. We then developed WIT, a static analysis technique that automatically extracts exception preconditions by analyzing the Java source code of public methods and constructors. Our extracted preconditions can often complement the available documentation of a project, even of carefully documented ones; we merged a selection of 170 exception preconditions as accepted pull requests in popular and widely used Java projects from the Apache Software Foundation. Later, we extended WIT to perform a modular analysis: after inferring the exception preconditions of a project `A`, it can use them to analyze the behavior of another project `B` whenever it calls out to any methods in `A`. This modular analysis enabled us to perform a large-scale investigation to automatically detect client calls that may trigger exception behavior in library methods. Automatically detecting such cases—which are under-explored in Automated Program Repair [96] and usually not detected by static analysis tools—allows us to suggest improvements to a project, based on how it uses libraries, in its documentation, tests, runtime and compile-time checks.

Although our tools and techniques can work with minimal requirements and yielded useful results—fitting in our vision of practicality— they still remain research prototypes. Aiming at fully practical approaches is a huge endeavour; it requires practical engineering decisions and solutions (e.g., ultra scalability, workflow integration) [130], attending to usability aspects (e.g., good warning message and description) [145], and factoring in human aspects (e.g., trust in the tool, reasons for accepting patches) [197].

7.1 Catering to Industry's Needs

SonarQube

In this section, we discuss some developments in the SonarQube tool that demonstrate that some of the research challenges we tackled are practically relevant to the industry. Our work on SonarQube [125, 129], started as part of my Master's thesis (University of Brasilia, Brazil) and continued in my PhD research, led to the submission and acceptance of GitHub pull requests in open-source projects maintained by commercial companies. For instance, we had pull requests accepted to projects maintained by Microsoft, and another to the SonarQube tool. During my time as a PhD candidate, I had some brief contact with SonarQube employees through X (formerly known as Twitter) and e-mail.

In September 2021, SonarQube introduced quick-fixes{183} in SonarLint, its IDE plugin. We exchanged a few messages with a developer who announced that the feature was under-development and pointed them towards our paper. The developer appreciated the work and thanked us. Sometime in 2022, we received an e-mail from SonarQube staff, mentioning our work on SpongeBugs.

During e-mail exchanges in April 2023, we briefly discussed our on-going work on analyzing calls to third-party code. We also referred to our original work on WIT, published in the ICSME conference 2022. In version 9.5, SonarQube announced rules that leverage symbolic execution,{184} which could detect illegal arguments, for instance. They clarified that these new rules did not work in third-party calls, but agreed that the direction was an interesting one. In August 2023, SonarQube announced the feature "deeper SAST",[1] which "uncovers security vulnerabilities [...] when your code uses and interacts with third-party dependency code".{185} The blog post of the feature release mentions that "traditional SAST tools scan **only your project code**. These tools are unaware of the **dependency code** and their security-relevant interactions [...] Traditional SAST tools understand only a fraction of the code actually executed and, as a result, miss deeply hidden vulnerabilities". SonarQube's approach and WIT's modular analysis share similarities: both i) analyze the source code of popular open-source dependencies; ii) populate a database with the analyzed library code that is used to check for interactions in client code. In all, although SonarQube's feature focus on security, it lends strength to our idea and work on focusing on analyzing the interaction between client and library code. To our knowledge, both features, "deeper SAST"

[1]SAST is an abbreviation of "Static application security testing".

and rules that require symbolic execution, are offered in the paid version of SonarQube.[2]

Research Internship

In 2022, during my time as a PhD candidate, I had the pleasure of doing a four-month internship in Aarhus, Denmark, as a member of the Programming System Group (PSG){187} of Uber Technologies, Inc.{188} The PSG team regularly publishes papers in the topics of programming languages and systems research. Notably, it is responsible for NullAway [12], a practical tool that provides compile-time checks to prevent Java's `NullPointerExceptions`. Below, I share some observations that connect to this thesis's vision of practical program analysis to my work at Uber.

During my internship, I worked on Piranha [166], an open-source tool for cleaning up code related to feature flags. Feature flags enable a software application to contain several functionalities as part of a monolithic source code repository; the set of functionalities visible to an user are later dynamically configured, usually at by a network payload during application initialization [166]. The tool has been recently revamped to support more general rewrites, not only related to feature flags.{189} Piranha supports a graph language, which allows users to define intricate transformations by chaining rewrite rules. My main contribution was adding GoLang{190} (Go, for short) support for Piranha and defining rules for cleaning stale feature flag code in Uber's internal codebase.

Source code analysis without building a project. A significant portion of Uber's tech stack is developed in Go.{191} The codebase is structured in a monorepo: a single repository containing multiple distinct projects.{192} Uber's Go monorepo is likely one of the largest Go codebases in the world. First of all, a monorepo facilitates code reuse, and simplifies dependency management. Moreover, a monorepo eases large-scale refactorings, as, for instance, a developer, or a source code tool, has access to all usages of e.g., a function. Unfortunately, monorepos also bring disadvantages. Build processes become complex, take longer times to complete, and require more resources; developers can be encouraged to use remote development environments,{193} which are typically significantly more powerful than a personal developer machine. The bottom-line is that Monorepos are hard to build. A source code tool relying on a successful build, which likely include running tests, won't scale well. Given the difficulty of building the entire repository, we see that tools that work solely on source code such as WIT—without compilation or test execution requirements—can be of great value.

Focused analysis and low false positive rate. At the level of thousands of engineers,{194} there is a huge effort on saving developers time—reinforced by the papers reporting APR experiences in Meta [10, 130], Microsoft [88], and Bloomberg [102, 198]. In other words, false positives are a waste of developers time; they provide a low "signal-to-noise ratio" [130]. This is very much consistent with Christakis and Bird [34] precision threshold of "no lower than 75–80%" and with my experience at Uber. To achieve a (very) low false positive ratio, one may follow Bloomberg's "deliberate strategy of only attempting to fix well-known, recurring bugs that take up much of software engineers' time". In the SpongeBugs work, we

[2]SonarCloud,{186} SonarQube's cloud offering, may provide some of these features with no cost for open-source projects.

focused on fixing recurrent warnings; in WIT work, we focused on analyzing only exception preconditions.

7.2 Closing Words

In this dissertation, we conducted work to improve Java software with an emphasis on *practicality*. We explored repairing static analysis violations and analyzed exception behavior. These topics showed to be useful for developers; we devised several pull requests from the outputs of our tools and techniques. These pull requests were accepted by popular and mature open-source projects, including some tools and libraries that we used for our research (e.g., Eclipse IDE, SonarQube, Apache Commons Lang, IO, and Text).

Part V

Appendices

8

Additional Contributions on Static Analysis Violations

This brief chapter summarizes the highlights of two publications in which I was a co-author (i.e., not the main contributor of the work), that complement my work in the area of repairing static analysis warnings.

8.1 C# Replication of SpongeBugs

We replicated our work of automatically fixing Java static analysis warnings for the C# language. Our results lead to a publication [155][1] in the "Replications and Negative Results" track of the SANER 2022 conference. This short chapter summarizes the motivation and main results of this replication.

C# is a popular[2] object-oriented programming language, which shares several similarities with Java but also some interesting differences. Its user community, in particular, is somewhat narrower than Java, as it revolves around the .NET framework—which was initially focused on the Windows operating system, but has gradually become available in other systems as well. To our knowledge, prior to our work, no study looked into how SATs are used by C# developers nor how to automatically fix some of their common warnings.

We investigated to what extent some known results about using static analyzers for Java change when considering C#. To this end, we combined two replications of previous Java studies. First, we studied which static analysis tools are most widely used among C# developers, and which warnings are more commonly reported by these tools on open-source C# projects. Second, we developed and empirically evaluated EagleRepair: a technique to automatically fix code in response to static analysis warnings; this is a replication of our previous work for Java [128, 129].

Our replication indicates, among other things, that

[1]This work was done as a part of Martin Odermatt's master's thesis, which I co-supervised. Martin was primarily in charge of the C# implementation of the SpongeBugs's technique, whereas I contributed to designing the research questions, analyzing the results, and comparing them to the original SpongeBugs work.

[2]C# ranks 6th in the 2023 IEEE Spectrum ranking;[195] Java ranks 2nd.

1. static code analysis is fairly popular among C# developers too;
2. ReSharper is the most widely used static analyzer for C#;
3. several static analysis rules are commonly violated in both Java and C# projects;
4. automatically generating fixes to static code analysis warnings with good precision is feasible in C#.

We also submitted pull requests containing some automatic generated fixes for C# warnings. Developers of popular projects (including several components of .NET's core framework) accepted 24 of 27 pull requests (including 250 fixes from a total of 281).

8.2 A Bot for Fixing Static Analysis Violations via Pull Requests

Providing fixes for common violations of static analysis warnings is useful, but integrating these tools into the development workflow is not a trivial task. We introduced C-3PR [27], an event-based bot infrastructure that automatically proposes fixes to static analysis violations through pull requests. This short section summarizes the results of the paper (published in the "Research Track" of the SANER 2020 conference).

C-3PR follows an approach of generating small patches only for violations found in recently modified code. In other words, the bot *will not* analyze the whole codebase to avoid overwhelming developers with a high number of fixes. C-3PR does not implement itself any static analysis or program transformation technique; it can seamlessly integrate existing static analysis tools that provide automatic fixes. At the time of writing, C-3PR was integrated with ESLint, TSLint, and the WalkMod Sonar Plugin; the tools target JavaScript, TypeScript, and Java, respectively. Unfortunately, we did not integrate SpongeBugs at the time, as it was under development.

We evaluated C-3PR for 8 months in an industrial setting of 8 to 14 developers and 16 projects. The bot created a total of 610 pull requests after performing 20,1346 analysis for 2,179 commits. Developers accepted and merged 346 (57%) of the pull requests.

The study unveiled some interesting aspects of how developers perceived and interacted with the fixes and pull requests. Developers rejected all pull request that had merge conflicts; the merge resolution effort was deemed too high. Another finding was that developers want a feature to exclude some files, or portions of code, from any analysis. For instance, SonarQube provide means to achieve both, either through its web interface, or through source code comments.

9 Other Contributions on Software Engineering Topics

During my time as a PhD candidate, I also contributed to research works that do not fit in the thesis main topics. These works all fall under the software engineering umbrella and helped me to improve as a researcher and collaborator.

[46] **Understanding the Impact of Introducing Lambda Expressions in Java Programs**
Walter Lucas Monteiro de Mendonça, José Fortes, Francisco Vitor Lopes, **Diego Marcilio**, Rodrigo Bonifácio, Edna Dias Canedo, Fernanda Lima, João Saraiva. In *Journal of Software Engineering Research and Development (JSERD 2020)*, Volume 8, 2020

[81] **A fine-grained data set and analysis of tangling in bug fixing commits**
Steffen Herbold, Alexander Trautsch, Benjamin Ledel, Alireza Aghamohammadi, Taher Ahmed Ghaleb, Kuljit Kaur Chahal, Tim Bossenmaier, Bhaveet Nagaria, Philip Makedonski, Matin Nili Ahmadabadi, Kristof Szabados, Helge Spieker, Matej Madeja, Nathaniel Hoy, Valentina Lenarduzzi, Shangwen Wang, Gema Rodríguez-Pérez, Ricardo Colomo Palacios, Roberto Verdecchia, Paramvir Singh, Yihao Qin, Debasish Chakroborti, Willard Davis, Vijay Walunj, Hongjun Wu, **Diego Marcilio**, Omar Alam, Abdullah Aldaeej, Idan Amit, Burak Turhan, Simon Eismann, Anna-Katharina Wickert, Ivano Malavolta, Matús Sulír, Fatemeh H. Fard, Austin Z. Henley, Stratos Kourtzanidis, Eray Tuzun, Christoph Treude, Simin Maleki Shamasbi, Ivan Pashchenko, Marvin Wyrich, James Davis, Alexander Serebrenik, Ella Albrecht, Ethem Utku Aktas, Daniel Strüber, Johannes Erbel. In *Empirical Software Engineering (EMSE 2022)*, Volume 27, Number 6, 2022

[183] **An Investigation of confusing code patterns in JavaScript**
Adriano Torres, Caio Oliveira, Marcio Vinicius Okimoto, **Diego Marcilio**, Pedro Queiroga, Fernando Castor, Rodrigo Bonifacio, Edna Dias Canedo, Marcio Ribeiro, Eduardo Monteiro. In *Journal of Systems and Software (JSS 2023)*, Volume 203, 2023

10

List of Submitted Pull Requests

As part of the evaluation of the work in this thesis, we created and submitted pull requests to open-source projects. We list all the pull requests below for the SpongeBugs (Chapter 3) and WIT (Chapter 5) works, respectively.

10.1 SpongeBugs

Table 10.1. Pull requests submitted as part of the evaluation of the SpongeBugs (Chapter 3) work.

#	Project	Rules	Merged?	Link
1	Eclipse IDE	C1	Yes	https://git.eclipse.org/r/#/c/140484/
2	Eclipse IDE	B2	Yes	https://git.eclipse.org/r/#/c/140524/
3	Eclipse IDE	C1	Yes	https://git.eclipse.org/r/#/c/140668/
4	Eclipse IDE	C2	Yes	https://git.eclipse.org/r/#/c/141027/
5	Eclipse IDE	C1	Yes	https://git.eclipse.org/r/#/c/140856/
6	Eclipse IDE	C9	Yes	https://git.eclipse.org/r/#/c/140959/
7	Eclipse IDE	C4	Yes	https://git.eclipse.org/r/#/c/142386/
8	Eclipse IDE	C3	Yes	https://git.eclipse.org/r/#/c/143599/
9	Eclipse IDE	C7	Yes	https://git.eclipse.org/r/#/c/143788/
10	SonarQube	C1	Yes	https://github.com/SonarSource/sonarqube/pull/3212
11	SpotBugs	C2	Yes	https://github.com/spotbugs/spotbugs/pull/967
12	atomix	B1	Yes	https://github.com/atomix/atomix/pull/1032
13	atomix	C5	Yes	https://github.com/atomix/atomix/pull/1031
14	Ant-Media Server	C5	Yes	https://github.com/ant-media/Ant-Media-Server/pull/1301
15	Ant-Media Server	C5	No	https://github.com/ant-media/Ant-Media-Server/pull/1302
16	Ant-Media Server	C2	Yes	https://github.com/ant-media/Ant-Media-Server/pull/1303
17	database-rider	C5	Yes	https://github.com/database-rider/database-rider/pull/138
18	database-rider	C2	Yes	https://github.com/database-rider/database-rider/pull/139
19	database-rider	C1	Yes	https://github.com/database-rider/database-rider/pull/140
20	database-rider	C7	Yes	https://github.com/database-rider/database-rider/pull/141
21	ddf	C5	No	https://github.com/codice/ddf/pull/4933
22	ddf	C7	Yes	https://github.com/codice/ddf/pull/4934
23	ddf	C8	Yes	https://github.com/codice/ddf/pull/4935
24	DependencyCheck	C2	Yes	https://github.com/jeremylong/DependencyCheck/pull/1976
25	keanu	C7	No	https://github.com/improbable-research/keanu/pull/566
26	keanu	C8	No	https://github.com/improbable-research/keanu/pull/567
27	keanu	C1	No	https://github.com/improbable-research/keanu/pull/568
29	mssql-jdbc	C5	Yes	https://github.com/microsoft/mssql-jdbc/pull/1077
30	Payara	C2	Yes	https://github.com/payara/Payara/pull/4022
31	Payara	C2	Yes	https://github.com/payara/Payara/pull/4026
32	Payara	C4	Yes	https://github.com/payara/Payara/pull/4030
33	Payara	C9	Yes	https://github.com/payara/Payara/pull/4032
34	Payara	B2	Yes	https://github.com/payara/Payara/pull/4033
35	Payara	C7	Yes	https://github.com/payara/Payara/pull/4038
36	PrimeFaces	C2	Yes	https://github.com/primefaces/primefaces/pull/4879
37	PrimeFaces	C3	Yes	https://github.com/primefaces/primefaces/pull/4880
38	PrimeFaces	C4	Yes	https://github.com/primefaces/primefaces/pull/4885
39	PrimeFaces	C7	Yes	https://github.com/primefaces/primefaces/pull/4887

10.2 wit

All the pull requests for the WIT work were submitted to Java projects part of the Apache Software Foundation.{196}

Table 10.2. Pull requests submitted as part of the evaluation of the WIT (Chapter 5) work.

#	Project	Merged?	Link
1	accumulo	Yes	`https://github.com/apache/accumulo/pull/2594`
2	commons-lang	Yes	`https://github.com/apache/commons-lang/pull/869`
3	commons-lang	Yes	`https://github.com/apache/commons-lang/pull/870`
4	commons-lang	Yes	`https://github.com/apache/commons-lang/pull/871`
5	commons-lang	Yes	`https://github.com/apache/commons-lang/pull/1047`
6	commons-math	No	`https://github.com/apache/commons-math/pull/206`
7	commons-math	No	`https://github.com/apache/commons-math/pull/207`
8	commons-io	Yes	`https://github.com/apache/commons-io/pull/339`
9	commons-text	Yes	`https://github.com/apache/commons-text/pull/331`

Bibliography

[1] Abreu, R., Zoeteweij, P. and van Gemund, A. J. [2007]. On the accuracy of spectrum-based fault localization, *Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION (TAICPART-MUTATION 2007)*, pp. 89–98.

[2] Aftandilian, E., Sauciuc, R., Priya, S. and Krishnan, S. [2012]. Building useful program analysis tools using an extensible java compiler, *12th IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2012, Riva del Garda, Italy, September 23-24, 2012*, IEEE Computer Society, pp. 14–23.
URL: *https://doi.org/10.1109/SCAM.2012.28*

[3] Amann, S., Nadi, S., Nguyen, H. A., Nguyen, T. N. and Mezini, M. [2016]. MUBench: a benchmark for API-Misuse Detectors, *Proceedings of the 13th International Conference on Mining Software Repositories, MSR 2016, Austin, TX, USA, May 14-22, 2016*, ACM, pp. 464–467.

[4] Amann, S., Nguyen, H. A., Nadi, S., Nguyen, T. N. and Mezini, M. [2019]. A systematic evaluation of static api-misuse detectors, *IEEE Trans. Software Eng.* **45**(12): 1170–1188.
URL: *https://doi.org/10.1109/TSE.2018.2827384*

[5] Ammann, P. and Offutt, J. [2007]. *Introduction to Software Testing*, 2nd edn, Cambridge University Press.

[6] Aniche, M., Treude, C. and Zaidman, A. [2022]. How developers engineer test cases: An observational study, *IEEE Trans. Software Eng.* **48**(12): 4925–4946.
URL: *https://doi.org/10.1109/TSE.2021.3129889*

[7] Asaduzzaman, M., Ahasanuzzaman, M., Roy, C. K. and Schneider, K. A. [2016]. How developers use exception handling in java?, *in* M. Kim, R. Robbes and C. Bird (eds), *Proceedings of the 13th International Conference on Mining Software Repositories, MSR 2016, Austin, TX, USA, May 14-22, 2016*, ACM, pp. 516–519.
URL: *https://doi.org/10.1145/2901739.2903500*

[8] Ayewah, N., Pugh, W., Hovemeyer, D., Morgenthaler, J. D. and Penix, J. [2008]. Using static analysis to find bugs, *IEEE Software* **25**(5): 22–29.

[9] Azim, T., Neamtiu, I. and Marvel, L. M. [2014]. Towards self-healing smartphone software via automated patching, *in* I. Crnkovic, M. Chechik and P. Grünbacher (eds), *ACM/IEEE International Conference on Automated Software Engineering, ASE '14, Vasteras, Sweden - September 15 - 19, 2014*, ACM, pp. 623–628.
URL: *https://doi.org/10.1145/2642937.2642955*

[10] Bader, J., Scott, A., Pradel, M. and Chandra, S. [2019]. Getafix: learning to fix bugs automatically, *Proc. ACM Program. Lang.* **3**(OOPSLA): 159:1–159:27.
URL: *https://doi.org/10.1145/3360585*

[11] Baker, W., O'Connor, M., Shahamiri, S. R. and Terragni, V. [2022]. Detect, fix, and verify tensorflow API misuses, *IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2022, Honolulu, HI, USA, March 15-18, 2022*, IEEE, pp. 925–929.
URL: *https://doi.org/10.1109/SANER53432.2022.00110*

[12] Banerjee, S., Clapp, L. and Sridharan, M. [2019]. NullAway: practical type-based null safety for Java, *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2019, Tallinn, Estonia, August 26-30, 2019*, ACM, pp. 740–750.

[13] Barik, T., Song, Y., Johnson, B. and Murphy-Hill, E. R. [2016]. From quick fixes to slow fixes: Reimagining static analysis resolutions to enable design space exploration, *2016 IEEE International Conference on Software Maintenance and Evolution, ICSME 2016, Raleigh, NC, USA, October 2-7, 2016*, IEEE Computer Society, pp. 211–221.
URL: *https://doi.org/10.1109/ICSME.2016.63*

[14] Barr, E. T., Harman, M., McMinn, P., Shahbaz, M. and Yoo, S. [2015]. The oracle problem in software testing: A survey, *IEEE Trans. Software Eng.* **41**(5): 507–525.
URL: *https://doi.org/10.1109/TSE.2014.2372785*

[15] Barrett, C., Sebastiani, R., Seshia, S. and Tinelli, C. [2009]. Satisfiability modulo theories, *in* A. Biere, M. J. H. Heule, H. van Maaren and T. Walsh (eds), *Handbook of Satisfiability*, IOS Press.

[16] Bavishi, R., Yoshida, H. and Prasad, M. R. [2019]. Phoenix: Automated data-driven synthesis of repairs for static analysis violations, *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2019, ACM, New York, NY, USA, pp. 613–624.
URL: *http://doi.acm.org/10.1145/3338906.3338952*

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

[18] Beller, M., Gousios, G., Panichella, A., Proksch, S., Amann, S. and Zaidman, A. [2019]. Developer testing in the IDE: patterns, beliefs, and behavior, *IEEE Trans. Software Eng.* **45**(3): 261–284.
URL: *https://doi.org/10.1109/TSE.2017.2776152*

[19] Bird, C., Ford, D., Zimmermann, T., Forsgren, N., Kalliamvakou, E., Lowdermilk, T. and Gazit, I. [2023]. Taking flight with copilot: Early insights and opportunities of

ai-powered pair-programming tools, *Queue* **20**(6): 35–57.
URL: *https://doi.org/10.1145/3582083*

[20] Blasi, A., Goffi, A., Kuznetsov, K., Gorla, A., Ernst, M. D., Pezzè, M. and Castellanos, S. D. [2018]. Translating code comments to procedure specifications, *in* F. Tip and E. Bodden (eds), *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2018, Amsterdam, The Netherlands, July 16-21, 2018*, ACM, pp. 242–253.
URL: *https://doi.org/10.1145/3213846.3213872*

[21] Bloch, J. [2018]. *Effective Java*, 3 edn, Pearson Education Inc.

[22] Brito, A., Xavier, L., Hora, A. and Valente, M. T. [2018]. Why and how Java developers break APIs, *25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pp. 255–265.

[23] Brown, T. B., Mann, B., Ryder, N., Subbiah, M., Kaplan, J., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., Agarwal, S., Herbert-Voss, A., Krueger, G., Henighan, T., Child, R., Ramesh, A., Ziegler, D. M., Wu, J., Winter, C., Hesse, C., Chen, M., Sigler, E., Litwin, M., Gray, S., Chess, B., Clark, J., Berner, C., McCandlish, S., Radford, A., Sutskever, I. and Amodei, D. [2020]. Language models are few-shot learners, *in* H. Larochelle, M. Ranzato, R. Hadsell, M. Balcan and H. Lin (eds), *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*.
URL: *https://proceedings.neurips.cc/paper/2020/hash/1457c0d6bfcb4967418bfb8ac142f64a-Abstract.html*

[24] Buse, R. P. L. and Weimer, W. [2008]. Automatic documentation inference for exceptions, *in* B. G. Ryder and A. Zeller (eds), *Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2008, Seattle, WA, USA, July 20-24, 2008*, ACM, pp. 273–282.
URL: *https://doi.org/10.1145/1390630.1390664*

[25] Cabral, B. and Marques, P. [2011]. A transactional model for automatic exception handling, *Comput. Lang. Syst. Struct.* **37**(1): 43–61.
URL: *https://doi.org/10.1016/j.cl.2010.09.002*

[26] Calcagno, C., Distefano, D., Dubreil, J., Gabi, D., Hooimeijer, P., Luca, M., O'Hearn, P. W., Papakonstantinou, I., Purbrick, J. and Rodriguez, D. [2015]. Moving fast with software verification, *NASA Formal Methods*, Springer.

[27] Carvalho, A., Luz, W. P., Marcilio, D., Bonifácio, R., Pinto, G. and Canedo, E. D. [2020]. C-3PR: A bot for fixing static analysis violations via pull requests, *in* K. Kontogiannis, F. Khomh, A. Chatzigeorgiou, M. Fokaefs and M. Zhou (eds), *27th IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2020, London, ON, Canada, February 18-21, 2020*, IEEE, pp. 161–171.
URL: *https://doi.org/10.1109/SANER48275.2020.9054842*

[28] Carzaniga, A., Gorla, A., Mattavelli, A., Perino, N. and Pezzè, M. [2013]. Automatic recovery from runtime failures, *in* D. Notkin, B. H. C. Cheng and K. Pohl (eds), *35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013*, IEEE Computer Society, pp. 782–791.
URL: *https://doi.org/10.1109/ICSE.2013.6606624*

[29] Chandra, S., Fink, S. J. and Sridharan, M. [2009]. Snugglebug: a powerful approach to weakest preconditions, *in* M. Hind and A. Diwan (eds), *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2009, Dublin, Ireland, June 15-21, 2009*, ACM, pp. 363–374.
URL: *https://doi.org/10.1145/1542476.1542517*

[30] Chang, B. and Choi, K. [2016]. A review on exception analysis, *Inf. Softw. Technol.* **77**: 1–16.
URL: *https://doi.org/10.1016/j.infsof.2016.05.003*

[31] Chang, H., Mariani, L. and Pezzè, M. [2013]. Exception handlers for healing component-based systems, *ACM Trans. Softw. Eng. Methodol.* **22**(4): 30:1–30:40.
URL: *https://doi.org/10.1145/2522920.2522923*

[32] Chen, H., Dou, W., Jiang, Y. and Qin, F. [2019]. Understanding exception-related bugs in large-scale cloud systems, *34th IEEE/ACM International Conference on Automated Software Engineering, ASE 2019, San Diego, CA, USA, November 11-15, 2019*, IEEE, pp. 339–351.
URL: *https://doi.org/10.1109/ASE.2019.00040*

[33] Chen, M., Tworek, J., Jun, H., Yuan, Q., de Oliveira Pinto, H. P., Kaplan, J., Edwards, H., Burda, Y., Joseph, N., Brockman, G., Ray, A., Puri, R., Krueger, G., Petrov, M., Khlaaf, H., Sastry, G., Mishkin, P., Chan, B., Gray, S., Ryder, N., Pavlov, M., Power, A., Kaiser, L., Bavarian, M., Winter, C., Tillet, P., Such, F. P., Cummings, D., Plappert, M., Chantzis, F., Barnes, E., Herbert-Voss, A., Guss, W. H., Nichol, A., Paino, A., Tezak, N., Tang, J., Babuschkin, I., Balaji, S., Jain, S., Saunders, W., Hesse, C., Carr, A. N., Leike, J., Achiam, J., Misra, V., Morikawa, E., Radford, A., Knight, M., Brundage, M., Murati, M., Mayer, K., Welinder, P., McGrew, B., Amodei, D., McCandlish, S., Sutskever, I. and Zaremba, W. [2021]. Evaluating large language models trained on code, *CoRR* **abs/2107.03374**.
URL: *https://arxiv.org/abs/2107.03374*

[34] Christakis, M. and Bird, C. [2016]. What developers want and need from program analysis: An empirical study, *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ASE 2016, Association for Computing Machinery, New York, NY, USA, p. 332–343.
URL: *https://doi.org/10.1145/2970276.2970347*

[35] Ciniselli, M., Pascarella, L., Aghajani, E., Scalabrino, S., Oliveto, R. and Bavota, G. [2023]. Source code recommender systems: The practitioners' perspective, *45th IEEE/ACM International Conference on Software Engineering, ICSE 2023, Melbourne,*

Australia, May 14-20, 2023, IEEE, pp. 2161–2172.
URL: *https://doi.org/10.1109/ICSE48619.2023.00182*

[36] Claessen, K. and Hughes, J. [2000]. QuickCheck: a lightweight tool for random testing of Haskell programs, *in* M. Odersky and P. Wadler (eds), *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP '00), Montreal, Canada, September 18-21, 2000*, ACM, pp. 268–279.
URL: *https://doi.org/10.1145/351240.351266*

[37] Coelho, R., Almeida, L., Gousios, G., van Deursen, A. and Treude, C. [2017]. Exception handling bug hazards in android - results from a mining study and an exploratory survey, *Empir. Softw. Eng.* **22**(3): 1264–1304.
URL: *https://doi.org/10.1007/s10664-016-9443-7*

[38] Cornu, B., Seinturier, L. and Monperrus, M. [2015]. Exception handling analysis and transformation using fault injection: Study of resilience against unanticipated exceptions, *Inf. Softw. Technol.* **57**: 66–76.
URL: *https://doi.org/10.1016/j.infsof.2014.08.004*

[39] Cousot, P. and Cousot, R. [1977]. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints, *in* R. M. Graham, M. A. Harrison and R. Sethi (eds), *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages, Los Angeles, California, USA, January 1977*, ACM, pp. 238–252.
URL: *https://doi.org/10.1145/512950.512973*

[40] Cousot, P., Cousot, R., Fähndrich, M. and Logozzo, F. [2013]. Automatic inference of necessary preconditions, *in* R. Giacobazzi, J. Berdine and I. Mastroeni (eds), *Verification, Model Checking, and Abstract Interpretation, 14th International Conference, VMCAI 2013, Rome, Italy, January 20-22, 2013. Proceedings*, Vol. 7737 of *Lecture Notes in Computer Science*, Springer, pp. 128–148.
URL: *https://doi.org/10.1007/978-3-642-35873-9_10*

[41] Cousot, P. and Halbwachs, N. [1978]. Automatic discovery of linear restraints among variables of a program, *in* A. V. Aho, S. N. Zilles and T. G. Szymanski (eds), *Conference Record of the Fifth Annual ACM Symposium on Principles of Programming Languages, Tucson, Arizona, USA, January 1978*, ACM Press, pp. 84–96.
URL: *https://doi.org/10.1145/512760.512770*

[42] Dabic, O., Aghajani, E. and Bavota, G. [2021]. Sampling projects in github for MSR studies, *18th IEEE/ACM International Conference on Mining Software Repositories, MSR 2021*, IEEE, pp. 560–564.

[43] Dalton, F., Ribeiro, M., Pinto, G., Fernandes, L., Gheyi, R. and Fonseca, B. [2020]. Is exceptional behavior testing an exception?: An empirical assessment using java automated tests, *in* J. Li, L. Jaccheri, T. Dingsøyr and R. Chitchyan (eds), *EASE '20: Evaluation and Assessment in Software Engineering, Trondheim, Norway, April 15-17, 2020*,

ACM, pp. 170–179.
URL: *https://doi.org/10.1145/3383219.3383237*

[44] Daniel, W. W. [1999]. *Biostatistics: A Foundation for Analysis in the Health Sciences*, 7 edn, Wiley.

[45] Dantas, R., Carvalho, A., Marcílio, D., Fantin, L., Silva, U., Lucas, W. and Bonifácio, R. [2018]. Reconciling the past and the present: An empirical study on the application of source code transformations to automatically rejuvenate Java programs, *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pp. 497–501.

[46] de Mendonça, W. L. M., Fortes, J., Lopes, F. V., Marcilio, D., Bonifácio, R., Canedo, E. D., Lima, F. and Saraiva, J. [2020]. Understanding the impact of introducing lambda expressions in java programs, *J. Softw. Eng. Res. Dev.* **8**.
URL: *https://doi.org/10.5753/jserd.2020.744*

[47] de Moura, L. M. and Bjørner, N. [2008]. Z3: an efficient SMT solver, *in* C. R. Ramakrishnan and J. Rehof (eds), *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*, Vol. 4963 of *Lecture Notes in Computer Science*, Springer, pp. 337–340.
URL: *https://doi.org/10.1007/978-3-540-78800-3_24*

[48] de Pádua, G. B. and Shang, W. [2017]. Studying the prevalence of exception handling anti-patterns, *in* G. Scanniello, D. Lo and A. Serebrenik (eds), *Proceedings of the 25th International Conference on Program Comprehension, ICPC 2017, Buenos Aires, Argentina, May 22-23, 2017*, IEEE Computer Society, pp. 328–331.
URL: *https://doi.org/10.1109/ICPC.2017.1*

[49] Derakhshanfar, P., Devroey, X., Panichella, A., Zaidman, A. and van Deursen, A. [2020]. Botsing, a search-based crash reproduction framework for Java, *ASE*, IEEE/ACM.

[50] Deursen, A., Moonen, L. M. F., Bergh, A. and Kok, G. [2001]. *Refactoring test code*, CWI (Centre for Mathematics and Computer Science).

[51] Dhar, A., Purandare, R., Dhawan, M. and Rangaswamy, S. [2015]. CLOTHO: saving programs from malformed strings and incorrect string-handling, *in* E. D. Nitto, M. Harman and P. Heymans (eds), *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, Bergamo, Italy, August 30 - September 4, 2015*, ACM, pp. 555–566.
URL: *https://doi.org/10.1145/2786805.2786877*

[52] Dietrich, J., Pearce, D. J., Jezek, K. and Brada, P. [2017]. Contracts in the wild: A study of java programs, *in* P. Müller (ed.), *31st European Conference on Object-Oriented Programming, ECOOP 2017, June 19-23, 2017, Barcelona, Spain*, Vol. 74 of *LIPIcs*, Schloss

Dagstuhl - Leibniz-Zentrum für Informatik, pp. 9:1–9:29.
URL: *https://doi.org/10.4230/LIPIcs.ECOOP.2017.9*

[53] Digkas, G., Lungu, M., Avgeriou, P., Chatzigeorgiou, A. and Ampatzoglou, A. [2018]. How do developers fix issues and pay back technical debt in the apache ecosystem?, *in* R. Oliveto, M. D. Penta and D. C. Shepherd (eds), *25th International Conference on Software Analysis, Evolution and Reengineering, SANER 2018, Campobasso, Italy, March 20-23, 2018*, IEEE Computer Society, pp. 153–163.
URL: *https://doi.org/10.1109/SANER.2018.8330205*

[54] Dobolyi, K. and Weimer, W. [2008]. Changing java's semantics for handling null pointer exceptions, *19th International Symposium on Software Reliability Engineering (ISSRE 2008), 11-14 November 2008, Seattle/Redmond, WA, USA*, IEEE Computer Society, pp. 47–56.
URL: *https://doi.org/10.1109/ISSRE.2008.59*

[55] Durieux, T., Cornu, B., Seinturier, L. and Monperrus, M. [2017]. Dynamic patch generation for null pointer exceptions using metaprogramming, *in* M. Pinzger, G. Bavota and A. Marcus (eds), *IEEE 24th International Conference on Software Analysis, Evolution and Reengineering, SANER 2017, Klagenfurt, Austria, February 20-24, 2017*, IEEE Computer Society, pp. 349–358.
URL: *https://doi.org/10.1109/SANER.2017.7884635*

[56] Durieux, T., Madeiral, F., Martinez, M. and Abreu, R. [2019]. Empirical review of java program repair tools: a large-scale experiment on 2, 141 bugs and 23, 551 repair attempts, *in* M. Dumas, D. Pfahl, S. Apel and A. Russo (eds), *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2019, Tallinn, Estonia, August 26-30, 2019*, ACM, pp. 302–313.
URL: *https://doi.org/10.1145/3338906.3338911*

[57] Durieux, T., Soto-Valero, C. and Baudry, B. [2021]. Duets: A dataset of reproducible pairs of Java library-clients, *18th IEEE/ACM International Conference on Mining Software Repositories, MSR 2021, Madrid, Spain, May 17-19, 2021*, IEEE, pp. 545–549.

[58] Ebert, F., Castor, F. and Serebrenik, A. [2015]. An exploratory study on exception handling bugs in java programs, *J. Syst. Softw.* **106**: 82–101.
URL: *https://doi.org/10.1016/j.jss.2015.04.066*

[59] Ebert, F., Castor, F. and Serebrenik, A. [2020]. A reflection on "an exploratory study on exception handling bugs in java programs", *in* K. Kontogiannis, F. Khomh, A. Chatzigeorgiou, M. Fokaefs and M. Zhou (eds), *27th IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2020, London, ON, Canada, February 18-21, 2020*, IEEE, pp. 552–556.
URL: *https://doi.org/10.1109/SANER48275.2020.9054791*

[60] Ernst, M. D., Cockrell, J., Griswold, W. G. and Notkin, D. [1999]. Dynamically discovering likely program invariants to support program evolution, *in* B. W. Boehm, D. Garlan and J. Kramer (eds), *Proceedings of the 1999 International Conference on Software Engineering, ICSE' 99, Los Angeles, CA, USA, May 16-22, 1999*, ACM, pp. 213–224.
URL: *https://doi.org/10.1145/302405.302467*

[61] Ernst, N. A. and Bavota, G. [2022]. Ai-driven development is here: Should you worry?, *IEEE Softw.* **39**(2): 106–110.
URL: *https://doi.org/10.1109/MS.2021.3133805*

[62] *Error Prone* [2022]. [Online; accessed 3-May-2022].
URL: *https://github.com/google/error-prone*

[63] Etemadi, K., Harrand, N., Larsén, S., Adzemovic, H., Phu, H. L., Verma, A., Madeiral, F., Wikström, D. and Monperrus, M. [2023]. Sorald: Automatic patch suggestions for sonarqube static analysis violations, *IEEE Trans. Dependable Secur. Comput.* **20**(4): 2794–2810.
URL: *https://doi.org/10.1109/TDSC.2022.3167316*

[64] Fan, L., Su, T., Chen, S., Meng, G., Liu, Y., Xu, L., Pu, G. and Su, Z. [2018]. Large-scale analysis of framework-specific exceptions in Android apps, *Proceedings of the 40th International Conference on Software Engineering*, ACM, pp. 408–419.
URL: *https://doi.org/10.1145/3180155.3180222*

[65] Fan, Z., Gao, X., Mirchev, M., Roychoudhury, A. and Tan, S. H. [2023]. Automated repair of programs from large language models, *45th IEEE/ACM International Conference on Software Engineering, ICSE 2023, Melbourne, Australia, May 14-20, 2023*, IEEE, pp. 1469–1481.
URL: *https://doi.org/10.1109/ICSE48619.2023.00128*

[66] Forward, A. and Lethbridge, T. [2002]. The relevance of software documentation, tools and technologies: a survey, *Proceedings of the 2002 ACM Symposium on Document Engineering, McLean, Virginia, USA, November 8-9, 2002*, ACM, pp. 26–33.
URL: *https://doi.org/10.1145/585058.585065*

[67] Fraser, G. and Arcuri, A. [2011]. Evosuite: automatic test suite generation for object-oriented software, *in* T. Gyimóthy and A. Zeller (eds), *SIGSOFT/FSE'11 19th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-19) and ESEC'11: 13th European Software Engineering Conference (ESEC-13), Szeged, Hungary, September 5-9, 2011*, ACM, pp. 416–419.
URL: *https://doi.org/10.1145/2025113.2025179*

[68] Fraser, G. and Arcuri, A. [2015]. 1600 faults in 100 projects: automatically finding faults while achieving high coverage with EvoSuite, *Empir. Softw. Eng.* **20**(3): 611–639.
URL: *https://doi.org/10.1007/s10664-013-9288-2*

[69] Gamboa, C., Canelas, P., Timperley, C. S. and Fonseca, A. [2023]. Usability-oriented design of liquid types for java, *45th IEEE/ACM International Conference on Software Engineering, ICSE 2023, Melbourne, Australia, May 14-20, 2023*, IEEE, pp. 1520–1532.
URL: *https://doi.org/10.1109/ICSE48619.2023.00132*

[70] Gazzola, L., Micucci, D. and Mariani, L. [2019]. Automatic software repair: A survey, *IEEE Trans. Software Eng.* **45**(1): 34–67.
URL: *https://doi.org/10.1109/TSE.2017.2755013*

[71] Georges, A., Buytaert, D. and Eeckhout, L. [2007]. Statistically rigorous Java performance evaluation, *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications*, OOPSLA '07, ACM, New York, NY, USA, pp. 57–76.
URL: *http://doi.acm.org/10.1145/1297027.1297033*

[72] Ghezzi, C. and Jazayeri, M. [1998]. *Programming language concepts*, 3rd edn, John Wiley & Sons.

[73] Ginelli, D., Riganelli, O., Micucci, D. and Mariani, L. [2021]. Exception-driven fault localization for automated program repair, *21st IEEE International Conference on Software Quality, Reliability and Security, QRS 2021, Hainan, China, December 6-10, 2021*, IEEE, pp. 598–607.
URL: *https://doi.org/10.1109/QRS54544.2021.00070*

[74] Goffi, A., Gorla, A., Ernst, M. D. and Pezzè, M. [2016a]. Automatic generation of oracles for exceptional behaviors, *in* A. Zeller and A. Roychoudhury (eds), *Proceedings of the 25th International Symposium on Software Testing and Analysis, ISSTA 2016, Saarbrücken, Germany, July 18-20, 2016*, ACM, pp. 213–224.
URL: *https://doi.org/10.1145/2931037.2931061*

[75] Goffi, A., Gorla, A., Ernst, M. D. and Pezzè, M. [2016b]. Automatic generation of oracles for exceptional behaviors, *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ISSTA 2016, Association for Computing Machinery, New York, NY, USA, p. 213–224.
URL: *https://doi.org/10.1145/2931037.2931061*

[76] Goues, C. L., Nguyen, T., Forrest, S. and Weimer, W. [2012]. Genprog: A generic method for automatic software repair, *IEEE Trans. Software Eng.* **38**(1): 54–72.
URL: *https://doi.org/10.1109/TSE.2011.104*

[77] Habib, A. and Pradel, M. [2018]. How many of all bugs do we find? a study of static bug detectors, *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ASE 2018, ACM, New York, NY, USA, pp. 317–328.
URL: *http://doi.acm.org/10.1145/3238147.3238213*

[78] Harrand, N., Benelallam, A., Soto-Valero, C., Bettega, F., Barais, O. and Baudry, B. [2022]. API beauty is in the eye of the clients: 2.2 million maven dependencies reveal the spectrum of client-API usages, *J. Syst. Softw.* **184**: 111134.

[79] Hassan, F., Bansal, C., Nagappan, N., Zimmermann, T. and Awadallah, A. H. [2020]. An empirical study of software exceptions in the field using search logs, *in* M. T. Baldassarre, F. Lanubile, M. Kalinowski and F. Sarro (eds), *ESEM '20: ACM / IEEE International Symposium on Empirical Software Engineering and Measurement, Bari, Italy, October 5-7, 2020*, ACM, pp. 4:1–4:12.
URL: *https://doi.org/10.1145/3382494.3410692*

[80] Hassan, F., Mostafa, S., Lam, E. S. L. and Wang, X. [2017]. Automatic building of java projects in software repositories: A study on feasibility and challenges, *in* A. Bener, B. Turhan and S. Biffl (eds), *2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, ESEM 2017, Toronto, ON, Canada, November 9-10, 2017*, IEEE Computer Society, pp. 38–47.
URL: *https://doi.org/10.1109/ESEM.2017.11*

[81] Herbold, S., Trautsch, A., Ledel, B., Aghamohammadi, A., Ghaleb, T. A., Chahal, K. K., Bossenmaier, T., Nagaria, B., Makedonski, P., Ahmadabadi, M. N., Szabados, K., Spieker, H., Madeja, M., Hoy, N., Lenarduzzi, V., Wang, S., Rodríguez-Pérez, G., Palacios, R. C., Verdecchia, R., Singh, P., Qin, Y., Chakroborti, D., Davis, W., Walunj, V., Wu, H., Marcilio, D., Alam, O., Aldaeej, A., Amit, I., Turhan, B., Eismann, S., Wickert, A., Malavolta, I., Sulír, M., Fard, F. H., Henley, A. Z., Kourtzanidis, S., Tuzun, E., Treude, C., Shamasbi, S. M., Pashchenko, I., Wyrich, M., Davis, J., Serebrenik, A., Albrecht, E., Aktas, E. U., Strüber, D. and Erbel, J. [2022]. A fine-grained data set and analysis of tangling in bug fixing commits, *Empir. Softw. Eng.* **27**(6): 125.
URL: *https://doi.org/10.1007/s10664-021-10083-5*

[82] Hoare, C. A. R. [2009]. Null references: The billion dollar mistake, `https://www.infoq.com/presentations/Null-References-The-Billion-Dollar-Mistake-Tony-Hoare/`.

[83] Holling, D., Hofbauer, A., Pretschner, A. and Gemmar, M. [2016]. Profiting from unit tests for integration testing, *2016 IEEE International Conference on Software Testing, Verification and Validation, ICST 2016, Chicago, IL, USA, April 11-15, 2016*, IEEE Computer Society, pp. 353–363.
URL: *https://doi.org/10.1109/ICST.2016.28*

[84] Hu, X., Li, G., Xia, X., Lo, D. and Jin, Z. [2018]. Deep code comment generation, *in* F. Khomh, C. K. Roy and J. Siegmund (eds), *Proceedings of the 26th Conference on Program Comprehension, ICPC 2018, Gothenburg, Sweden, May 27-28, 2018*, ACM, pp. 200–210.
URL: *https://doi.org/10.1145/3196321.3196334*

[85] *Infer Static Analyzer* [2022]. [Online; accessed 3-May-2022].
URL: *https://fbinfer.com/*

[86] Jakobus, B., Barbosa, E. A., Garcia, A. F. and de Lucena, C. J. P. [2015]. Contrasting exception handling code across languages: An experience report involving 50 open

source projects, *26th IEEE International Symposium on Software Reliability Engineering, ISSRE 2015, Gaithersbury, MD, USA, November 2-5, 2015*, IEEE Computer Society, pp. 183–193.
URL: *https://doi.org/10.1109/ISSRE.2015.7381812*

[87] *JavaParser* [2022]. [Online; accessed 3-May-2022].
URL: *https://javaparser.org/*

[88] Jin, M., Shahriar, S., Tufano, M., Shi, X., Lu, S., Sundaresan, N. and Svyatkovskiy, A. [2023]. Inferfix: End-to-end program repair with llms, *CoRR* **abs/2303.07263**.
URL: *https://doi.org/10.48550/arXiv.2303.07263*

[89] Johnson, B., Song, Y., Murphy-Hill, E. and Bowdidge, R. [2013]. Why don't software developers use static analysis tools to find bugs?, *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, IEEE Press, Piscataway, NJ, USA, pp. 672–681.
URL: *http://dl.acm.org/citation.cfm?id=2486788.2486877*

[90] *JUnit 5* [2022]. [Online; accessed 3-May-2022].
URL: *https://junit.org/junit5/*

[91] Just, R., Jalali, D. and Ernst, M. D. [2014]. Defects4j: a database of existing faults to enable controlled testing studies for java programs, *in* C. S. Pasareanu and D. Marinov (eds), *International Symposium on Software Testing and Analysis, ISSTA '14, San Jose, CA, USA - July 21 - 26, 2014*, ACM, pp. 437–440.
URL: *https://doi.org/10.1145/2610384.2628055*

[92] Kabadi, V., Kong, D., Xie, S., Prana, G. A. A., Le, T.-D. B., Le, X.-B. D. and Lo, D. [2023]. The Future Can't Help Fix The Past: Assessing Program Repair In The Wild, *ICSME*.

[93] Kalliamvakou, E., Gousios, G., Blincoe, K., Singer, L., Germán, D. M. and Damian, D. E. [2016]. An in-depth study of the promises and perils of mining github, *Empir. Softw. Eng.* **21**(5): 2035–2071.
URL: *https://doi.org/10.1007/s10664-015-9393-5*

[94] Kechagia, M., Devroey, X., Panichella, A., Gousios, G. and van Deursen, A. [2019a]. *Effective and Efficient API Misuse Detection via Exception Propagation and Search-Based Testing*, Association for Computing Machinery, New York, NY, USA, p. 192–203.
URL: *https://doi.org/10.1145/3293882.3330552*

[95] Kechagia, M., Devroey, X., Panichella, A., Gousios, G. and van Deursen, A. [2019b]. Effective and efficient API misuse detection via exception propagation and search-based testing, *in* D. Zhang and A. Møller (eds), *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2019, Beijing, China, July 15-19, 2019*, ACM, pp. 192–203.
URL: *https://doi.org/10.1145/3293882.3330552*

[96] Kechagia, M., Mechtaev, S., Sarro, F. and Harman, M. [2022]. Evaluating automatic program repair capabilities to repair API misuses, *IEEE Trans. Software Eng.* **48**(7): 2658–2679.
URL: *https://doi.org/10.1109/TSE.2021.3067156*

[97] Kechagia, M. and Spinellis, D. [2014]. Undocumented and unchecked: exceptions that spell trouble, *in* P. T. Devanbu, S. Kim and M. Pinzger (eds), *11th Working Conference on Mining Software Repositories, MSR 2014, Proceedings, May 31 - June 1, 2014, Hyderabad, India*, ACM, pp. 312–315.
URL: *https://doi.org/10.1145/2597073.2597089*

[98] Kery, M. B., Goues, C. L. and Myers, B. A. [2016]. Examining programmer practices for locally handling exceptions, *in* M. Kim, R. Robbes and C. Bird (eds), *Proceedings of the 13th International Conference on Mining Software Repositories, MSR 2016, Austin, TX, USA, May 14-22, 2016*, ACM, pp. 484–487.
URL: *https://doi.org/10.1145/2901739.2903497*

[99] Kim, D., Nam, J., Song, J. and Kim, S. [2013]. Automatic patch generation learned from human-written patches, *in* D. Notkin, B. H. C. Cheng and K. Pohl (eds), *35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013*, IEEE Computer Society, pp. 802–811.
URL: *https://doi.org/10.1109/ICSE.2013.6606626*

[100] Kim, J., Batory, D. S., Dig, D. and Azanza, M. [2016]. Improving refactoring speed by 10x, *in* L. K. Dillon, W. Visser and L. A. Williams (eds), *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016*, ACM, pp. 1145–1156.
URL: *https://doi.org/10.1145/2884781.2884802*

[101] Kim, M., Kim, Y., Jeong, H., Heo, J., Kim, S., Chung, H. and Lee, E. [2022]. An empirical study of deep transfer learning-based program repair for kotlin projects, *in* A. Roychoudhury, C. Cadar and M. Kim (eds), *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2022, Singapore, Singapore, November 14-18, 2022*, ACM, pp. 1441–1452.
URL: *https://doi.org/10.1145/3540250.3558967*

[102] Kirbas, S., Windels, E., McBello, O., Kells, K., Pagano, M. W., Szalanski, R., Nowack, V., Winter, E. R., Counsell, S., Bowes, D., Hall, T., Haraldsson, S. and Woodward, J. R. [2021]. On the introduction of automatic program repair in bloomberg, *IEEE Softw.* **38**(4): 43–51.
URL: *https://doi.org/10.1109/MS.2021.3071086*

[103] Klint, P., van der Storm, T. and Vinju, J. J. [2009]. RASCAL: A domain specific language for source code analysis and manipulation, *Ninth IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2009, Edmonton, Alberta, Canada,*

September 20-21, 2009, IEEE Computer Society, pp. 168–177.
URL: *https://doi.org/10.1109/SCAM.2009.28*

[104] Kolak, S. D., Martins, R., Le Goues, C. and Hellendoorn, V. J. [2022]. Patch generation with language models: Feasibility and scaling behavior, *Deep Learning for Code Workshop*.

[105] Lawall, J. and Muller, G. [2022]. Automating program transformation with coccinelle, *in* J. V. Deshmukh, K. Havelund and I. Perez (eds), *NASA Formal Methods - 14th International Symposium, NFM 2022, Pasadena, CA, USA, May 24-27, 2022, Proceedings*, Vol. 13260 of *Lecture Notes in Computer Science*, Springer, pp. 71–87.
URL: *https://doi.org/10.1007/978-3-031-06773-0_4*

[106] Le Goues, C., Pradel, M. and Roychoudhury, A. [2019]. Automated program repair, *Commun. ACM* **62**(12): 56–65.
URL: *https://doi.org/10.1145/3318162*

[107] Lee, J., Hong, S. and Oh, H. [2022]. NPEX: repairing java null pointer exceptions without tests, *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022*, ACM, pp. 1532–1544.
URL: *https://doi.org/10.1145/3510003.3510186*

[108] Li, R., Chen, B., Zhang, F., Sun, C. and Peng, X. [2022]. Detecting runtime exceptions by deep code representation learning with attention-based graph neural networks, *IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2022, Honolulu, HI, USA, March 15-18, 2022*, IEEE, pp. 373–384.
URL: *https://doi.org/10.1109/SANER53432.2022.00053*

[109] Li, X., Jiang, J., Benton, S., Xiong, Y. and Zhang, L. [2021]. A large-scale study on API misuses in the wild, *14th IEEE Conference on Software Testing, Verification and Validation, ICST 2021, Porto de Galinhas, Brazil, April 12-16, 2021*, IEEE, pp. 241–252.
URL: *https://doi.org/10.1109/ICST49551.2021.00034*

[110] Lin, D., Koppel, J., Chen, A. and Solar-Lezama, A. [2017]. Quixbugs: a multi-lingual program repair benchmark set based on the quixey challenge, *in* G. C. Murphy (ed.), *Proceedings Companion of the 2017 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity, SPLASH 2017, Vancouver, BC, Canada, October 23 - 27, 2017*, ACM, pp. 55–56.
URL: *https://doi.org/10.1145/3135932.3135941*

[111] Liskov, B. H. and Wing, J. M. [1994]. A behavioral notion of subtyping, *ACM Trans. Program. Lang. Syst.* **16**(6): 1811–1841.
URL: *https://doi.org/10.1145/197320.197383*

[112] Liu, K., Kim, D., Bissyandé, T. F., Yoo, S. and Traon, Y. L. [2021]. Mining fix patterns for findbugs violations, *IEEE Trans. Software Eng.* **47**(1): 165–188.
URL: *https://doi.org/10.1109/TSE.2018.2884955*

[113] Liu, K., Koyuncu, A., Bissyandé, T. F., Kim, D., Klein, J. and Traon, Y. L. [2019]. You cannot fix what you cannot find! an investigation of fault localization bias in benchmarking automated program repair systems, *12th IEEE Conference on Software Testing, Validation and Verification, ICST 2019, Xi'an, China, April 22-27, 2019*, IEEE, pp. 102–113.
URL: *https://doi.org/10.1109/ICST.2019.00020*

[114] Liu, K., Koyuncu, A., Kim, D. and Bissyandé, T. F. [2019]. AVATAR: fixing semantic bugs with fix patterns of static analysis violations, *in* X. Wang, D. Lo and E. Shihab (eds), *26th IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2019, Hangzhou, China, February 24-27, 2019*, IEEE, pp. 456–467.
URL: *https://doi.org/10.1109/SANER.2019.8667970*

[115] Liu, K., Li, L., Koyuncu, A., Kim, D., Liu, Z., Klein, J. and Bissyandé, T. F. [2021]. A critical review on the evaluation of automated program repair systems, *J. Syst. Softw.* **171**: 110817.
URL: *https://doi.org/10.1016/j.jss.2020.110817*

[116] Liu, K., Wang, S., Koyuncu, A., Kim, K., Bissyandé, T. F., Kim, D., Wu, P., Klein, J., Mao, X. and Traon, Y. L. [2020]. On the efficiency of test suite based program repair: A systematic assessment of 16 automated repair systems for java programs, *CoRR* **abs/2008.00914**.
URL: *https://arxiv.org/abs/2008.00914*

[117] Logozzo, F. [2004]. Automatic inference of class invariants, *VMCAI*, Vol. 2937 of *LNCS*, Springer, pp. 211–222.

[118] Logozzo, F. and Ball, T. [2012]. Modular and verified automatic program repair, *in* G. T. Leavens and M. B. Dwyer (eds), *Proceedings of the 27th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2012, part of SPLASH 2012, Tucson, AZ, USA, October 21-25, 2012*, ACM, pp. 133–146.
URL: *https://doi.org/10.1145/2384616.2384626*

[119] Long, F., Amidon, P. and Rinard, M. C. [2017]. Automatic inference of code transforms for patch generation, *in* E. Bodden, W. Schäfer, A. van Deursen and A. Zisman (eds), *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017, Paderborn, Germany, September 4-8, 2017*, ACM, pp. 727–739.
URL: *https://doi.org/10.1145/3106237.3106253*

[120] Loriot, B., Madeiral, F. and Monperrus, M. [2022]. Styler: learning formatting conventions to repair checkstyle violations, *Empir. Softw. Eng.* **27**(6): 149.
URL: *https://doi.org/10.1007/s10664-021-10107-0*

[121] Madeiral, F., Urli, S., de Almeida Maia, M. and Monperrus, M. [2019]. BEARS: an extensible java bug benchmark for automatic program repair studies, *in* X. Wang, D. Lo and E. Shihab (eds), *26th IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2019, Hangzhou, China, February 24-27, 2019*, IEEE,

pp. 468–478.
URL: *https://doi.org/10.1109/SANER.2019.8667991*

[122] Mahajan, S., Abolhassani, N. and Prasad, M. R. [2020a]. *Recommending Stack Overflow Posts for Fixing Runtime Exceptions Using Failure Scenario Matching*, Association for Computing Machinery, New York, NY, USA, p. 1052–1064.
URL: *https://doi.org/10.1145/3368089.3409764*

[123] Mahajan, S., Abolhassani, N. and Prasad, M. R. [2020b]. Recommending stack overflow posts for fixing runtime exceptions using failure scenario matching, *ESEC/FSE*, ACM.

[124] Mao, K., Harman, M. and Jia, Y. [2016]. Sapienz: multi-objective automated testing for android applications, *in* A. Zeller and A. Roychoudhury (eds), *Proceedings of the 25th International Symposium on Software Testing and Analysis, ISSTA 2016, Saarbrücken, Germany, July 18-20, 2016*, ACM, pp. 94–105.
URL: *https://doi.org/10.1145/2931037.2931054*

[125] Marcilio, D., Bonifácio, R., Monteiro, E., Canedo, E. D., Luz, W. P. and Pinto, G. [2019]. Are static analysis violations really fixed?: a closer look at realistic usage of sonarqube, *in* Y. Guéhéneuc, F. Khomh and F. Sarro (eds), *Proceedings of the 27th International Conference on Program Comprehension, ICPC 2019, Montreal, QC, Canada, May 25-31, 2019*, IEEE / ACM, pp. 209–219.
URL: *https://doi.org/10.1109/ICPC.2019.00040*

[126] Marcilio, D. and Furia, C. A. [2021]. How java programmers test exceptional behavior, *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*, pp. 207–218.

[127] Marcilio, D. and Furia, C. A. [2022]. What is thrown? Lightweight precise automatic extraction of exception preconditions in Java methods, *IEEE International Conference on Software Maintenance and Evolution, ICSME 2022, Limassol, Cyprus, October 3-7, 2022*, IEEE, pp. 340–351.

[128] Marcilio, D., Furia, C. A., Bonifácio, R. and Pinto, G. [2019]. Automatically generating fix suggestions in response to static code analysis warnings, *19th International Working Conference on Source Code Analysis and Manipulation, SCAM 2019, Cleveland, OH, USA, September 30 - October 1, 2019*, IEEE, pp. 34–44.
URL: *https://doi.org/10.1109/SCAM.2019.00013*

[129] Marcilio, D., Furia, C. A., Bonifácio, R. and Pinto, G. [2020]. Spongebugs: Automatically generating fix suggestions in response to static code analysis warnings, *J. Syst. Softw.* **168**: 110671.
URL: *https://doi.org/10.1016/j.jss.2020.110671*

[130] Marginean, A., Bader, J., Chandra, S., Harman, M., Jia, Y., Mao, K., Mols, A. and Scott, A. [2019]. Sapfix: automated end-to-end repair at scale, *in* H. Sharp and M. Whalen

(eds), *Proceedings of the 41st International Conference on Software Engineering: Software Engineering in Practice, ICSE (SEIP) 2019, Montreal, QC, Canada, May 25-31, 2019*, IEEE / ACM, pp. 269–278.
URL: *https://doi.org/10.1109/ICSE-SEIP.2019.00039*

[131] Martinez, M. and Monperrus, M. [2019]. Astor: Exploring the design space of generate-and-validate program repair beyond genprog, *J. Syst. Softw.* **151**: 65–80.
URL: *https://doi.org/10.1016/j.jss.2019.01.069*

[132] Mastropaolo, A., Pascarella, L., Guglielmi, E., Ciniselli, M., Scalabrino, S., Oliveto, R. and Bavota, G. [2023]. On the robustness of code generation techniques: An empirical study on github copilot, *45th IEEE/ACM International Conference on Software Engineering, ICSE 2023, Melbourne, Australia, May 14-20, 2023*, IEEE, pp. 2149–2160.
URL: *https://doi.org/10.1109/ICSE48619.2023.00181*

[133] McConnell, S. [2004]. *Code Complete*, 2nd edn, Microsoft Press.

[134] McElreath, R. [2015]. *Statistical Rethinking: A Bayesian Course with Examples in R and Stan*, Chapman & Hall.

[135] Mechtaev, S., Yi, J. and Roychoudhury, A. [2015]. Directfix: Looking for simple program repairs, *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 1, pp. 448–458.

[136] Mechtaev, S., Yi, J. and Roychoudhury, A. [2016]. Angelix: scalable multiline program patch synthesis via symbolic analysis, *in* L. K. Dillon, W. Visser and L. A. Williams (eds), *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016*, ACM, pp. 691–701.
URL: *https://doi.org/10.1145/2884781.2884807*

[137] Melo, H., Coelho, R. and Treude, C. [2019]. Unveiling exception handling guidelines adopted by Java developers, *26th IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2019, Hangzhou, China, February 24-27, 2019*, IEEE, pp. 128–139.
URL: *https://doi.org/10.1109/SANER.2019.8668001*

[138] Mesbah, A., Rice, A., Johnston, E., Glorioso, N. and Aftandilian, E. [2019]. Deepdelta: learning to repair compilation errors, *in* M. Dumas, D. Pfahl, S. Apel and A. Russo (eds), *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2019, Tallinn, Estonia, August 26-30, 2019*, ACM, pp. 925–936.
URL: *https://doi.org/10.1145/3338906.3340455*

[139] Meurer, A., Smith, C. P., Paprocki, M., Čertík, O., Kirpichev, S. B., Rocklin, M., Kumar, A., Ivanov, S., Moore, J. K., Singh, S., Rathnayake, T., Vig, S., Granger, B. E., Muller, R. P., Bonazzi, F., Gupta, H., Vats, S., Johansson, F., Pedregosa, F., Curry, M. J., Terrel, A. R., Roučka, v., Saboo, A., Fernando, I., Kulal, S., Cimrman, R. and Scopatz, A. [2017].

Sympy: symbolic computing in python, *PeerJ Computer Science* **3**: e103.
URL: *https://doi.org/10.7717/peerj-cs.103*

[140] Meyer, B. [1997]. *Object-Oriented Software Construction*, 2nd edn, Prentice Hall.

[141] Meyer, B. [2005]. The dependent delegate dilemma, *Engineering Theories of Software Intensive Systems*, Springer.

[142] Monperrus, M. [2018]. Automatic software repair: A bibliography, *ACM Comput. Surv.* **51**(1).
URL: *https://doi.org/10.1145/3105906*

[143] Monperrus, M., Urli, S., Durieux, T., Martinez, M., Baudry, B. and Seinturier, L. [2019]. Repairnator patches programs automatically, *Ubiquity* **2019**(July): 1–12.
URL: *https://doi.org/10.1145/3349589*

[144] Motwani, M., Soto, M., Brun, Y., Just, R. and Goues, C. L. [2022]. Quality of automated program repair on real-world defects, *IEEE Trans. Software Eng.* **48**(2): 637–661.
URL: *https://doi.org/10.1109/TSE.2020.2998785*

[145] Nachtigall, M., Schlichtig, M. and Bodden, E. [2022]. A large-scale study of usability criteria addressed by static analysis tools, *in* S. Ryu and Y. Smaragdakis (eds), *ISSTA '22: 31st ACM SIGSOFT International Symposium on Software Testing and Analysis, Virtual Event, South Korea, July 18 - 22, 2022*, ACM, pp. 532–543.
URL: *https://doi.org/10.1145/3533767.3534374*

[146] Naitou, K., Tanikado, A., Matsumoto, S., Higo, Y., Kusumoto, S., Kirinuki, H., Kurabayashi, T. and Tanno, H. [2018]. Toward introducing automated program repair techniques to industrial software development, *in* F. Khomh, C. K. Roy and J. Siegmund (eds), *Proceedings of the 26th Conference on Program Comprehension, ICPC 2018, Gothenburg, Sweden, May 27-28, 2018*, ACM, pp. 332–335.
URL: *https://doi.org/10.1145/3196321.3196358*

[147] Najumudheen, E. S. F., Mall, R. and Samanta, D. [2019]. Modeling and coverage analysis of programs with exception handling, *in* R. Naik, S. Sarkar, T. T. Hildebrandt, A. Kumar and R. Sharma (eds), *Proceedings of the 12th Innovations on Software Engineering Conference (formerly known as India Software Engineering Conference), ISEC 2019, Pune, India, February 14-16, 2019*, ACM, pp. 15:1–15:11.
URL: *https://doi.org/10.1145/3299771.3299785*

[148] Nakamaru, T., Matsunaga, T., Yamazaki, T., Akiyama, S. and Chiba, S. [2020]. An empirical study of method chaining in java, *in* S. Kim, G. Gousios, S. Nadi and J. Hejderup (eds), *MSR '20: 17th International Conference on Mining Software Repositories, Seoul, Republic of Korea, 29-30 June, 2020*, ACM, pp. 93–102.
URL: *https://doi.org/10.1145/3379597.3387441*

[149] Nakshatri, S., Hegde, M. and Thandra, S. [2016]. Analysis of exception handling patterns in java projects: an empirical study, *in* M. Kim, R. Robbes and C. Bird (eds), *Proceedings of the 13th International Conference on Mining Software Repositories, MSR 2016, Austin, TX, USA, May 14-22, 2016*, ACM, pp. 500–503.
URL: *https://doi.org/10.1145/2901739.2903499*

[150] Nassif, M., Hernandez, A., Sridharan, A. and Robillard, M. P. [2022]. Generating unit tests for documentation, *IEEE Trans. Software Eng.* **48**(9): 3268–3279.
URL: *https://doi.org/10.1109/TSE.2021.3087087*

[151] Nguyen, H. A., Dyer, R., Nguyen, T. N. and Rajan, H. [2014]. Mining preconditions of apis in large-scale code corpus, *in* S. Cheung, A. Orso and M. D. Storey (eds), *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, (FSE-22), Hong Kong, China, November 16 - 22, 2014*, ACM, pp. 166–177.
URL: *https://doi.org/10.1145/2635868.2635924*

[152] Nguyen, H. A., Phan, H. D., Samantha, S. K., Nguyen, S., Yadavally, A., Wang, S., Rajan, H. and Nguyen, T. N. [2022]. A hybrid approach for inference between behavioral exception API documentation and implementations, and its applications, *37th IEEE/ACM International Conference on Automated Software Engineering, ASE 2022, Rochester, MI, USA, October 10-14, 2022*, ACM, pp. 2:1–2:13.

[153] Nguyen, H. D. T., Qi, D., Roychoudhury, A. and Chandra, S. [2013]. Semfix: program repair via semantic analysis, *in* D. Notkin, B. H. C. Cheng and K. Pohl (eds), *35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013*, IEEE Computer Society, pp. 772–781.
URL: *https://doi.org/10.1109/ICSE.2013.6606623*

[154] Nguyen, T., Vu, P. and Nguyen, T. [2020]. Code recommendation for exception handling, *in* P. Devanbu, M. B. Cohen and T. Zimmermann (eds), *ESEC/FSE '20: 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Virtual Event, USA, November 8-13, 2020*, ACM, pp. 1027–1038.
URL: *https://doi.org/10.1145/3368089.3409690*

[155] Odermatt, M., Marcilio, D. and Furia, C. A. [2022]. Static analysis warnings and automatic fixing: A replication for c# projects, *IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2022, Honolulu, HI, USA, March 15-18, 2022*, IEEE, pp. 805–816.
URL: *https://doi.org/10.1109/SANER53432.2022.00098*

[156] Pacheco, C. and Ernst, M. D. [2007]. Randoop: feedback-directed random testing for java, *in* R. P. Gabriel, D. F. Bacon, C. V. Lopes and G. L. S. Jr. (eds), *Companion to the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2007, October 21-25, 2007, Montreal, Quebec, Canada*, ACM, pp. 815–816.
URL: *https://doi.org/10.1145/1297846.1297902*

[157] Pandita, R., Xiao, X., Zhong, H., Xie, T., Oney, S. and Paradkar, A. [2012]. Inferring method specifications from natural language api descriptions, *2012 34th International Conference on Software Engineering (ICSE)*, pp. 815–825.

[158] Papi, M. M., Ali, M., Jr., T. L. C., Perkins, J. H. and Ernst, M. D. [2008]. Practical pluggable types for Java, *Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2008, Seattle, WA, USA, July 20-24, 2008*, ACM, pp. 201–212.

[159] Peng, S., Kalliamvakou, E., Cihon, P. and Demirer, M. [2023]. The Impact of AI on Developer Productivity: Evidence from Github Copilot.
URL: *https://doi.org/10.48550/arXiv.2302.06590*

[160] Peruma, A., Almalki, K., Newman, C. D., Mkaouer, M. W., Ouni, A. and Palomba, F. [2020]. tsdetect: an open source test smells detection tool, *in* P. Devanbu, M. B. Cohen and T. Zimmermann (eds), *ESEC/FSE '20: 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Virtual Event, USA, November 8-13, 2020*, ACM, pp. 1650–1654.
URL: *https://doi.org/10.1145/3368089.3417921*

[161] Pezzè, M. and Young, M. [2007]. *Software Testing and Analysis: Process, Principles and Techniques: Process, Principles, and Techniques*, Wiley.

[162] Phan, H., Nguyen, H. A., Nguyen, T. N. and Rajan, H. [2017]. Statistical learning for inference between implementations and documentation, *39th IEEE/ACM International Conference on Software Engineering: New Ideas and Emerging Technologies Results Track, ICSE-NIER 2017, Buenos Aires, Argentina, May 20-28, 2017*, IEEE Computer Society, pp. 27–30.
URL: *https://doi.org/10.1109/ICSE-NIER.2017.9*

[163] Polikarpova, N., Ciupa, I. and Meyer, B. [2009]. A comparative study of programmer-written and automatically inferred contracts, *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis*, ISSTA '09, Association for Computing Machinery, New York, NY, USA, p. 93–104.
URL: *https://doi.org/10.1145/1572272.1572284*

[164] Prenner, J. A., Babii, H. and Robbes, R. [2022]. Can openai's codex fix bugs?: An evaluation on quixbugs, *3rd IEEE/ACM International Workshop on Automated Program Repair, APR@ICSE 2022, Pittsburgh, PA, USA, May 19, 2022*, IEEE, pp. 69–75.
URL: *https://doi.org/10.1145/3524459.3527351*

[165] Ram, A., Sawant, A. A., Castelluccio, M. and Bacchelli, A. [2018]. What makes a code change easier to review: an empirical investigation on code change reviewability, *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*, pp. 201–212.
URL: *https://doi.org/10.1145/3236024.3236080*

[166] Ramanathan, M. K., Clapp, L., Barik, R. and Sridharan, M. [2020]. Piranha: reducing feature flag debt at uber, *in* G. Rothermel and D. Bae (eds), *ICSE-SEIP 2020: 42nd International Conference on Software Engineering, Software Engineering in Practice, Seoul, South Korea, 27 June - 19 July, 2020*, ACM, pp. 221–230.
URL: *https://doi.org/10.1145/3377813.3381350*

[167] Ramanathan, M. K., Grama, A. and Jagannathan, S. [2007a]. Path-sensitive inference of function precedence protocols, *Proceedings of the 29th International Conference on Software Engineering*, ICSE '07, IEEE Computer Society, USA, p. 240–250.
URL: *https://doi.org/10.1109/ICSE.2007.63*

[168] Ramanathan, M. K., Grama, A. and Jagannathan, S. [2007b]. Static specification inference using predicate mining, *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, Association for Computing Machinery, New York, NY, USA, p. 123–134.
URL: *https://doi.org/10.1145/1250734.1250749*

[169] Robillard, M. P., Bodden, E., Kawrykow, D., Mezini, M. and Ratchford, T. [2013]. Automated api property inference techniques, *IEEE Transactions on Software Engineering* **39**(5): 613–637.

[170] Robillard, M. P. and Murphy, G. C. [2000]. Designing robust java programs with exceptions, *in* J. C. Knight and D. S. Rosenblum (eds), *ACM SIGSOFT Symposium on Foundations of Software Engineering, an Diego, California, USA, November 6-10, 2000, Proceedings*, ACM, pp. 2–10.
URL: *https://doi.org/10.1145/355045.355046*

[171] Saha, R. K., Lyu, Y., Lam, W., Yoshida, H. and Prasad, M. R. [2018]. Bugs.jar: a large-scale, diverse dataset of real-world java bugs, *in* A. Zaidman, Y. Kamei and E. Hill (eds), *Proceedings of the 15th International Conference on Mining Software Repositories, MSR 2018, Gothenburg, Sweden, May 28-29, 2018*, ACM, pp. 10–13.
URL: *https://doi.org/10.1145/3196398.3196473*

[172] Seghir, M. N. and Schrammel, P. [2014]. Necessary and sufficient preconditions via eager abstraction, *in* J. Garrigue (ed.), *Programming Languages and Systems - 12th Asian Symposium, APLAS 2014, Singapore, November 17-19, 2014, Proceedings*, Vol. 8858 of *Lecture Notes in Computer Science*, Springer, pp. 236–254.
URL: *https://doi.org/10.1007/978-3-319-12736-1_13*

[173] Sena, D., Coelho, R., Kulesza, U. and Bonifácio, R. [2016]. Understanding the exception handling strategies of java libraries: an empirical study, *in* M. Kim, R. Robbes and C. Bird (eds), *Proceedings of the 13th International Conference on Mining Software Repositories, MSR 2016, Austin, TX, USA, May 14-22, 2016*, ACM, pp. 212–222.
URL: *https://doi.org/10.1145/2901739.2901757*

[174] Shoham, S., Yahav, E., Fink, S. and Pistoia, M. [2007]. Static specification mining using automata-based abstractions, *Proceedings of the 2007 International Symposium*

on Software Testing and Analysis, ISSTA '07, Association for Computing Machinery, New York, NY, USA, p. 174–184.
URL: *https://doi.org/10.1145/1273463.1273487*

[175] Soares, E., Ribeiro, M., Amaral, G., Gheyi, R., Fernandes, L., Garcia, A., Fonseca, B. and Santos, A. L. M. [2020]. Refactoring test smells: A perspective from open-source developers, *in* E. Cavalcante, F. Dantas and T. Batista (eds), *SAST 20: 5th Brazilian Symposium on Systematic and Automated Software Testing, Natal, Brazil, October 19-23, 2020*, ACM, pp. 50–59.
URL: *https://doi.org/10.1145/3425174.3425212*

[176] *SonarQube* [2022]. [Online; accessed 3-May-2022].
URL: *https://www.sonarqube.org/*

[177] Soremekun, E., Kirschner, L., Böhme, M. and Papadakis, M. [2023]. Evaluating the impact of experimental assumptions in automated fault localization, *Proceedings of the ACM/IEEE 45th International Conference on Software Engineering*, ICSE 2023, pp. 1–13.

[178] Summers, A. J. and Müller, P. [2011]. Freedom before commitment: a lightweight type system for object initialisation, *in* C. V. Lopes and K. Fisher (eds), *Proceedings of the 26th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2011, part of SPLASH 2011, Portland, OR, USA, October 22 - 27, 2011*, ACM, pp. 1013–1032.
URL: *https://doi.org/10.1145/2048066.2048142*

[179] Tan, S. H., Marinov, D., Tan, L. and Leavens, G. T. [2012]. @tcomment: Testing javadoc comments to detect comment-code inconsistencies, *in* G. Antoniol, A. Bertolino and Y. Labiche (eds), *Fifth IEEE International Conference on Software Testing, Verification and Validation, ICST 2012, Montreal, QC, Canada, April 17-21, 2012*, IEEE Computer Society, pp. 260–269.
URL: *https://doi.org/10.1109/ICST.2012.106*

[180] Tao, Y., Han, D. and Kim, S. [2014]. Writing acceptable patches: An empirical study of open source project patches, *30th IEEE International Conference on Software Maintenance and Evolution, Victoria, BC, Canada, September 29 - October 3, 2014*, IEEE Computer Society, pp. 271–280.
URL: *https://doi.org/10.1109/ICSME.2014.49*

[181] *TestNG* [2022]. [Online; accessed 3-May-2022].
URL: *https://testng.org/doc/*

[182] Thummalapenta, S. and Xie, T. [2009]. Alattin: Mining alternative patterns for detecting neglected conditions, *ASE 2009, 24th IEEE/ACM International Conference on Automated Software Engineering, Auckland, New Zealand, November 16-20, 2009*, IEEE Computer Society, pp. 283–294.
URL: *https://doi.org/10.1109/ASE.2009.72*

[183] Torres, A., Oliveira, C., Okimoto, M. V., Marcilio, D., Queiroga, P., Castor, F., Bonifácio, R., Canedo, E. D., Ribeiro, M. and Monteiro, E. [2023]. An investigation of confusing code patterns in javascript, *J. Syst. Softw.* **203**: 111731.
URL: *https://doi.org/10.1016/j.jss.2023.111731*

[184] Tschannen, J., Furia, C. A., Nordio, M. and Meyer, B. [2014]. Program checking with less hassle, *Proceedings of the 5th Working Conference on Verified Software: Theories, Tools and Experiments (VSTTE 2013)*, Vol. 8164 of *Lecture Notes in Computer Science*, Springer, pp. 149–169.

[185] Tufano, M., Palomba, F., Bavota, G., Penta, M. D., Oliveto, R., Lucia, A. D. and Poshyvanyk, D. [2016]. An empirical investigation into the nature of test smells, *in* D. Lo, S. Apel and S. Khurshid (eds), *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016, Singapore, September 3-7, 2016*, ACM, pp. 4–15.
URL: *https://doi.org/10.1145/2970276.2970340*

[186] Tómasdóttir, K. F., Aniche, M. and van Deursen, A. [2017]. Why and how JavaScript developers use linters, *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 578–589.

[187] Utture, A., Liu, S., Kalhauge, C. G. and Palsberg, J. [2022]. Striking a balance: Pruning false-positives from static call graphs, *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022*, ACM, pp. 2043–2055.
URL: *https://doi.org/10.1145/3510003.3510166*

[188] Vahabzadeh, A., Fard, A. M. and Mesbah, A. [2015]. An empirical study of bugs in test code, *in* R. Koschke, J. Krinke and M. P. Robillard (eds), *2015 IEEE International Conference on Software Maintenance and Evolution, ICSME 2015, Bremen, Germany, September 29 - October 1, 2015*, IEEE Computer Society, pp. 101–110.
URL: *https://doi.org/10.1109/ICSM.2015.7332456*

[189] Vallée-Rai, R., Co, P., Gagnon, E., Hendren, L. J., Lam, P. and Sundaresan, V. [1999]. Soot - a java bytecode optimization framework, *in* S. A. MacKay and J. H. Johnson (eds), *Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative Research, November 8-11, 1999, Mississauga, Ontario, Canada*, IBM, p. 13.
URL: *https://dl.acm.org/citation.cfm?id=782008*

[190] Vassallo, C., Panichella, S., Palomba, F., Proksch, S., Gall, H. C. and Zaidman, A. [2020]. How developers engage with static analysis tools in different contexts, *Empir. Softw. Eng.* **25**(2): 1419–1457.
URL: *https://doi.org/10.1007/s10664-019-09750-5*

[191] Wang, C., Peng, X., Liu, M., Xing, Z., Bai, X., Xie, B. and Wang, T. [2019]. A learning-based approach for automatic construction of domain glossary from source code and

documentation, *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2019, Association for Computing Machinery, New York, NY, USA, p. 97–108.
URL: *https://doi.org/10.1145/3338906.3338963*

[192] Wasylkowski, A. and Zeller, A. [2009]. Mining temporal specifications from object usage, *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*, ASE '09, IEEE Computer Society, USA, p. 295–306.
URL: *https://doi.org/10.1109/ASE.2009.30*

[193] Watson, C., Tufano, M., Moran, K., Bavota, G. and Poshyvanyk, D. [2020]. On learning meaningful assert statements for unit test cases, *in* G. Rothermel and D. Bae (eds), *ICSE '20: 42nd International Conference on Software Engineering, Seoul, South Korea, 27 June - 19 July, 2020*, ACM, pp. 1398–1409.
URL: *https://doi.org/10.1145/3377811.3380429*

[194] Wei, Y., Furia, C. A., Kazmin, N. and Meyer, B. [2011]. Inferring better contracts, *in* R. N. Taylor, H. Gall and N. Medvidović (eds), *Proceedings of the 33rd International Conference on Software Engineering (ICSE'11)*, ACM, pp. 191–200.

[195] Weimer, W. and Necula, G. C. [2004]. Finding and preventing run-time error handling mistakes, *in* J. M. Vlissides and D. C. Schmidt (eds), *Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2004, October 24-28, 2004, Vancouver, BC, Canada*, ACM, pp. 419–431.
URL: *https://doi.org/10.1145/1028976.1029011*

[196] Wen, M., Liu, Y., Wu, R., Xie, X., Cheung, S. and Su, Z. [2019]. Exposing library API misuses via mutation analysis, *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*, IEEE / ACM, pp. 866–877.
URL: *https://doi.org/10.1109/ICSE.2019.00093*

[197] Winter, E., Nowack, V., Bowes, D., Counsell, S., Hall, T., Haraldsson, S. Ó. and Woodward, J. R. [2023]. Let's talk with developers, not about developers: A review of automatic program repair research, *IEEE Trans. Software Eng.* **49**(1): 419–436.
URL: *https://doi.org/10.1109/TSE.2022.3152089*

[198] Winter, E. R., Nowack, V., Bowes, D., Counsell, S., Hall, T., Haraldsson, S. Ó., Woodward, J. R., Kirbas, S., Windels, E., McBello, O., Atakishiyev, A., Kells, K. and Pagano, M. W. [2022]. Towards developer-centered automatic program repair: findings from bloomberg, *in* A. Roychoudhury, C. Cadar and M. Kim (eds), *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2022, Singapore, Singapore, November 14-18, 2022*, ACM, pp. 1578–1588.
URL: *https://doi.org/10.1145/3540250.3558953*

[199] Xia, C. S., Wei, Y. and Zhang, L. [2023]. Automated program repair in the era of large pre-trained language models, *45th IEEE/ACM International Conference on Software Engineering, ICSE 2023, Melbourne, Australia, May 14-20, 2023*, IEEE, pp. 1482–1494.
URL: *https://doi.org/10.1109/ICSE48619.2023.00129*

[200] Xuan, J., Martinez, M., Demarco, F., Clement, M., Marcote, S. R. L., Durieux, T., Berre, D. L. and Monperrus, M. [2017]. Nopol: Automatic repair of conditional statement bugs in java programs, *IEEE Trans. Software Eng.* **43**(1): 34–55.
URL: *https://doi.org/10.1109/TSE.2016.2560811*

[201] Zeng, H., Chen, J., Shen, B. and Zhong, H. [2021]. Mining API constraints from library and client to detect API misuses, *28th Asia-Pacific Software Engineering Conference, APSEC 2021, Taipei, Taiwan, December 6-9, 2021*, IEEE, pp. 161–170.

[202] Zerouali, A. and Mens, T. [2017]. Analyzing the evolution of testing library usage in open source java projects, *in* M. Pinzger, G. Bavota and A. Marcus (eds), *IEEE 24th International Conference on Software Analysis, Evolution and Reengineering, SANER 2017, Klagenfurt, Austria, February 20-24, 2017*, IEEE Computer Society, pp. 417–421.
URL: *https://doi.org/10.1109/SANER.2017.7884645*

[203] Zhang, C., Yang, J., Zhang, Y., Fan, J., Zhang, X., Zhao, J. and Ou, P. [2012]. Automatic parameter recommendation for practical API usage, *in* M. Glinz, G. C. Murphy and M. Pezzè (eds), *34th International Conference on Software Engineering, ICSE 2012, June 2-9, 2012, Zurich, Switzerland*, IEEE Computer Society, pp. 826–836.
URL: *https://doi.org/10.1109/ICSE.2012.6227136*

[204] Zhang, J., Wang, X., Zhang, H., Sun, H., Pu, Y. and Liu, X. [2020]. Learning to handle exceptions, *35th IEEE/ACM International Conference on Automated Software Engineering, ASE 2020, Melbourne, Australia, September 21-25, 2020*, IEEE, pp. 29–41.
URL: *https://doi.org/10.1145/3324884.3416568*

[205] Zhang, P. and Elbaum, S. G. [2012]. Amplifying tests to validate exception handling code, *in* M. Glinz, G. C. Murphy and M. Pezzè (eds), *34th International Conference on Software Engineering, ICSE 2012, June 2-9, 2012, Zurich, Switzerland*, IEEE Computer Society, pp. 595–605.
URL: *https://doi.org/10.1109/ICSE.2012.6227157*

[206] Zhang, T., Upadhyaya, G., Reinhardt, A., Rajan, H. and Kim, M. [2018]. Are code examples on an online Q&A forum reliable?: a study of API misuse on stack overflow, *Proceedings of the 40th International Conference on Software Engineering, ICSE*, ACM, pp. 886–896.
URL: *https://doi.org/10.1145/3180155.3180260*

[207] Zhong, H., Meng, N., Li, Z. and Jia, L. [2020]. An empirical study on API parameter rules, *in* G. Rothermel and D. Bae (eds), *ICSE '20: 42nd International Conference on Software Engineering, Seoul, South Korea, 27 June - 19 July, 2020*, ACM, pp. 899–911.
URL: *https://doi.org/10.1145/3377811.3380922*

[208] Zhong, H. and Su, Z. [2013]. Detecting API documentation errors, *in* A. L. Hosking, P. T. Eugster and C. V. Lopes (eds), *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2013, part of SPLASH 2013, Indianapolis, IN, USA, October 26-31, 2013*, ACM, pp. 803–816.
URL: *https://doi.org/10.1145/2509136.2509523*

[209] Zhong, L. and Wang, Z. [2023]. A study on robustness and reliability of large language model code generation.

[210] Zhou, Y., Wang, C., Yan, X., Chen, T., Panichella, S. and Gall, H. C. [2020]. Automatic detection and repair recommendation of directive defects in java API documentation, *IEEE Trans. Software Eng.* **46**(9): 1004–1023.
URL: *https://doi.org/10.1109/TSE.2018.2872971*

URL References

1. An example of URL reference: `https://dvmarcilio.github.io`
2. `http://www.25hoursaday.com/CsharpVsJava.html`
3. `https://spectrum.ieee.org/the-top-programming-languages-2023`
4. `https://www.tiobe.com/tiobe-index/`
5. `http://portal.core.edu.au/conf-ranks/`
6. `https://leetcode.com/`
7. `https://www.allthingsdistributed.com/2023/04/how-ai-coding-companions-will-change-the-way-developers-work.htm`
8. `https://aws.amazon.com/blogs/aws/new-customization-capability-in-amazon-codewhisperer-generates-even-better-`
9. `https://aws.amazon.com/codewhisperer/customize/`
10. `https://www.uber.com/en-DE/blog/the-transformative-power-of-generative-ai`
11. `https://checkstyle.sourceforge.io/`
12. `https://docs.oracle.com/javase/tutorial/essential/exceptions/definition.html`
13. `https://fbinfer.com/docs/all-issue-types/#nullptr_dereference`
14. `https://scan.coverity.com/`
15. `https://spotbugs.readthedocs.io/en/stable/bugDescriptions.html`
16. `https://spotbugs.readthedocs.io/en/latest/bugDescriptions.html#np-method-may-return-null-but-is-declared-non`
17. `https://rules.sonarsource.com/java`
18. `https://twitter.com/vogella/status/1096088933144952832`
19. `https://github.com/AlDanial/cloc`
20. `https://www.gerritcodereview.com/`
21. `https://www.vogella.com/tutorials/EclipsePlatformDevelopment/article.html##gerrit-verification-failures-due-t`
22. `https://git.eclipse.org/r/##/c/140959/`
23. `http://tutor.rascal-mpl.org/Rascal/Rascal.html#/Rascal/Libraries/util/Benchmark/benchmark/benchmark.html`
24. `https://github.com/google/closure-compiler`
25. `https://commons.apache.org/proper/commons-lang/javadocs/api-2.0/org/apache/commons/lang/StringUtils.html#indexOfDifference(java.lang.String,%20java.lang.String)`
26. `https://stackoverflow.com/questions/156503`
27. `https://docs.oracle.com/javase/tutorial/essential/exceptions/runtime.html`
28. `https://docs.oracle.com/javase/8/docs/api/java/lang/Throwable.html`
29. `https://blog.jooq.org/2013/04/28/rare-uses-of-a-controlflowexception/`
30. `https://github.com/junit-team/junit4/wiki/Exception-testing`
31. `https://assertj.github.io/doc/#assertj-core-exception-assertions`
32. `https://developer.android.com/studio/test`

33. https://code.google.com/archive/p/catch-exception/
34. https://junit.org/junit5/docs/current/user-guide/#launcher-api-discovery
35. https://javaparser.org/
36. https://cran.r-project.org/
37. https://github.com/eclipse/eclipse-collections
38. https://github.com/eclipse/eclipse-collections/blob/63be239538ff2676680ff57294e5aa08ce03b602/CONTRIBUTING.md
39. https://github.com/apache/geode/blob/4b84af392df529a94a7d3163966d9b28ae9cf79c/TESTING.md
40. https://cwiki.apache.org/confluence/display/GEODE/About+Unit+Testing
41. https://github.com/hazelcast/hazelcast
42. https://sonarcloud.io/dashboard?id=hz-os-master
43. https://github.com/apilayer/restcountries
44. https://github.com/processing/processing/blob/4cc297c66908899cd29480c202536ecf749854e8/README.md
45. https://docs.oracle.com/javase/tutorial/essential/exceptions/runtime.html
46. https://docs.oracle.com/javase/8/docs/api/java/io/EOFException.html
47. https://github.com/apache/hadoop
48. https://github.com/google/ExoPlayer
49. https://github.com/apache/hadoop/blob/2ba44a73bf2bb7ef33a2259bd19ee62ef9bb5659/hadoop-hdfs-project/hadoop-hdfs/src/test/java/org/apache/hadoop/hdfs/TestFSInputChecker.java#L200
50. https://github.com/google/ExoPlayer/blob/5bfad37cd0d2917f8c62440a42e1f65aa535cac7/library/core/src/test/java/com/google/android/exoplayer2/extractor/DefaultExtractorInputTest.java#L140
51. https://github.com/spring-projects/spring-framework
52. https://docs.spring.io/spring/docs/3.0.0.M4/reference/html/ch11s02.html
53. https://docs.oracle.com/javase/8/docs/api/java/lang/Error.html
54. https://github.com/oracle/graal
55. https://github.com/eclipse/openj9
56. https://github.com/apache/commons-lang
57. https://github.com/google/guava
58. https://github.com/apache/flink
59. https://github.com/google/j2objc
60. https://github.com/apache/flink/blob/fe8625c70a710143e2e197a9ee3179d5a32e002e/flink-streaming-java/src/test/java/org/apache/flink/streaming/runtime/operators/windowing/KeyMapTest.java#L101
61. https://github.com/junit-team/junit4/wiki/Exception-testing#trycatch-idiom
62. https://github.com/vipshop/Saturn
63. https://github.com/apache/flink/blob/df525b77d29ccd89649a64e5faad96c93f61ca08/flink-core/src/test/java/org/apache/flink/core/memory/MemorySegmentUndersizedTest.java#L130
64. https://docs.oracle.com/javase/8/docs/api/java/lang/NullPointerException.html
65. https://lgtm.com/
66. https://github.com/apache/activemq

67. https://github.com/apache/activemq/blob/9abe2c6f97c92fc99c5a2ef02846f62002a671cf/activemq-unit-tests/src/test/java/org/apache/activemq/broker/region/cursors/FilePendingMessageCursorTestSupport.java#L83
68. https://github.com/apache/geode/blob/4b84af392df529a94a7d3163966d9b28ae9cf79c/geode-core/src/distributedTest/java/org/apache/geode/internal/cache/execute/OnGroupsFunctionExecutionDUnitTest.java
69. https://github.com/hibernate/hibernate-orm/commit/3489f75e1d455049cffd45694f025b97487b429f, https://lgtm.com/projects/g/hibernate/hibernate-orm/rev/1e5a8d3c434c6791b89281c4ebf04ef08181fcd7
70. https://github.com/hibernate/hibernate-orm/
71. https://github.com/apache/kafka
72. https://github.com/apache/kafka/blob/9c8f75c4b624084c954b4da69f092211a9ac4689/streams/src/test/java/org/apache/kafka/streams/kstream/WindowedSerdesTest.java#L73
73. https://github.com/junit-team/junit4/issues/706#issuecomment-21385116
74. https://github.com/JodaOrg/joda-time
75. https://github.com/pedrovgs/Algorithms
76. https://github.com/SonarSource/sonarqube
77. https://github.com/SonarSource/sonarqube/commit/14d6de3529b12ec0af367e551cf66ac6daae1ca7
78. https://github.com/SonarSource/sonarqube/commit/e4b519ed129dbc7b76eab00d6c48166a8993e35f
79. https://github.com/apache/beam
80. https://github.com/apache/beam/blob/f7b23ec69fa68f4f0b6386ecec32ab12982e4098/runners/google-cloud-dataflow-ja src/test/java/org/apache/beam/runners/dataflow/DataflowRunnerTest.java
81. https://github.com/apache/beam/pull/5150#discussion_r182212260
82. https://github.com/RoaringBitmap/RoaringBitmap
83. https://github.com/RoaringBitmap/RoaringBitmap/pull/396
84. https://github.com/neo4j/neo4j
85. https://github.com/neo4j/neo4j/commit/0ce66ab6ebd454f9dbb5a0cf36e0f2483edec413
86. https://github.com/neo4j/neo4j/pull/12444#pullrequestreview-398471953
87. https://github.com/codecentric/spring-boot-admin
88. https://github.com/codecentric/spring-boot-admin/commit/caef5a004cbbc4ba897d854094b2546efd15d52b#
89. https://github.com/spring-io/initializr
90. https://github.com/spring-io/initializr/commit/2816c216315b989c45c25c18fd9f72bb606db8ee#diff-e49dd42170d49f6c1eb73139645c48cf
91. https://rules.sonarsource.com/java/RSPEC-2698
92. https://github.com/apache/hadoop/blob/2ba44a73bf2bb7ef33a2259bd19ee62ef9bb5659/hadoop-hdfs-project/hadoop-hdfs/src/test/java/org/apache/hadoop/hdfs/server/namenode/FSXAttrBaseTest.java#L274
93. https://github.com/apache/hadoop/blob/2ba44a73bf2bb7ef33a2259bd19ee62ef9bb5659/hadoop-common-project/hadoop-common/src/main/java/org/apache/hadoop/fs/FileSystem.java#L3049-L3062
94. https://junit-pioneer.org/docs/vintage-test/
95. Apache Dubbo https://dubbo.apache.org/en/
96. Apache Commons Lang https://commons.apache.org/proper/commons-lang/
97. Apache Dubbo on GitHub https://github.com/apache/dubbo
98. https://github.com/apache/commons-lang/commit/ba607f525b842661d40195d0d4778528e2384e70

99. JavaParser: https://github.com/javaparser/javaparser
100. JGraphT: https://jgrapht.org/
101. https://commons.apache.org/proper/commons-lang/apidocs/org/apache/commons/lang3/ArrayUtils.html
102. https://jgrapht.org/javadoc-1.4.0/org/jgrapht/alg/shortestpath/AllDirectedPaths.html
103. https://github.com/apache/commons-text/blob/21fc34f17175aba66f55fb6f805e60c13055da49/src/main/java/org/apache/commons/text/RandomStringGenerator.java#L362-L366
104. https://github.com/apache/commons-lang/blob/ce477d9140f1439c44c7a852d7df1e069e21cb85/src/main/java/org/apache/commons/lang3/Validate.java#L107-L111
105. https://www.mongodb.com/
106. https://github.com/javaparser/javaparser/tree/master/javaparser-core-serialization/src/main/java/com/github/javaparser/serialization
107. https://github.com/square/moshi
108. SymPy https://www.sympy.org/en/index.html
109. https://github.com/INRIA/spoon/tree/6d157f35491eabe6e7f7505a8ebc22a9694f491f/spoon-control-flow
110. https://github.com/openjdk/jdk/tree/jdk-11%2B28
111. https://github.com/apache/accumulo/pull/2594
112. https://github.com/apache/commons-lang/pull/869
113. https://github.com/apache/commons-lang/pull/870
114. https://github.com/apache/commons-lang/pull/871
115. https://github.com/apache/commons-math/pull/206
116. https://github.com/apache/commons-math/pull/207
117. https://github.com/apache/commons-text/pull/311
118. https://github.com/apache/commons-io/pull/339
119. https://github.com/apache/commons-io/blob/2ae025fe5c4a7d2046c53072b0898e37a079fe62/src/main/java/org/apache/commons/io/EndianUtils.java
120. https://asm.ow2.io/asm4-guide.pdf#page=62
121. https://github.com/jfree/jfreechart
122. https://github.com/openjdk/jdk/blob/da75f3c4ad5bdf25167a3ed80e51f567ab3dbd01/src/java.base/share/classes/java/util/Collections.java#L1312
123. https://github.com/apache/lucene-solr/blob/7ada4032180b516548fc0263f42da6a7a917f92b/solr/solrj/src/java/org/apache/solr/client/solrj/io/sql/ResultSetImpl.java#L631
124. https://github.com/apache/logging-log4j2/blob/59f6848b70eebbaa3aa0e14f7186b9b5e1942b5a/log4j-layout-template-json/src/main/java/org/apache/logging/log4j/layout/template/json/util/TruncatingBufferedWriter.java#L160
125. https://github.com/apache/logging-log4j2/blob/59f6848b70eebbaa3aa0e14f7186b9b5e1942b5a/log4j-perf/src/main/java/org/apache/logging/log4j/perf/nogc/OpenHashStringMap.java#L476
126. https://github.com/apache/jackrabbit/blob/35d5732bc1418718f49553a81e42ac4146619dcf/jackrabbit-spi-com
src/main/java/org/apache/jackrabbit/spi/commons/name/PathFactoryImpl.java#L217
127. https://github.com/apache/commons-lang/blob/5def1c8d634f12a265662f38188cd611aa1e574b/src/main/java/org/apache/commons/lang3/ArrayUtils.java#L2807
128. https://github.com/openjdk/jdk/blob/da75f3c4ad5bdf25167a3ed80e51f567ab3dbd01/src/java.base/share/classes/java/lang/String.java#L1002

129. https://github.com/openjdk/jdk/blob/da75f3c4ad5bdf25167a3ed80e51f567ab3dbd01/src/java.base/share/classes/java/lang/String.java#L142-L153
130. https://openjdk.org/jeps/254
131. https://github.com/openjdk/jdk/blob/da75f3c4ad5bdf25167a3ed80e51f567ab3dbd01/src/java.base/share/classes/java/lang/String.java#L140
132. https://github.com/apache/pdfbox/blob/2fcdf26e400952357bef4276121bd59fb7e4040a/pdfbox/src/main/java/org/apache/pdfbox/pdmodel/interactive/annotation/PDAnnotationWidget.java#L110
133. https://github.com/openjdk/jdk/blob/da75f3c4ad5bdf25167a3ed80e51f567ab3dbd01/src/java.base/share/classes/java/lang/Double.java#L555
134. https://github.com/openjdk/jdk/blob/da75f3c4ad5bdf25167a3ed80e51f567ab3dbd01/src/java.base/share/classes/java/lang/Double.java#L50-L62
135. https://github.com/openjdk/jdk/blob/da75f3c4ad5bdf25167a3ed80e51f567ab3dbd01/src/java.base/share/classes/java/util/Stack.java#L80
136. https://github.com/openjdk/jdk/blob/da75f3c4ad5bdf25167a3ed80e51f567ab3dbd01/src/java.base/share/classes/java/util/Vector.java#L112
137. https://github.com/apache/pdfbox/blob/2fcdf26e400952357bef4276121bd59fb7e4040a/pdfbox/src/main/java/org/apache/pdfbox/pdmodel/common/function/type4/ExecutionContext.java#L65
138. https://github.com/apache/commons-text/blob/21fc34f17175aba66f55fb6f805e60c13055da49/src/main/java/org/apache/commons/text/similarity/IntersectionResult.java#L58
139. https://github.com/openjdk/jdk/blob/da75f3c4ad5bdf25167a3ed80e51f567ab3dbd01/src/java.base/share/classes/jdk/internal/org/objectweb/asm/util/ASMifier.java#L1120
140. https://github.com/apache/commons-io/blob/2ae025fe5c4a7d2046c53072b0898e37a079fe62/src/main/java/org/apache/commons/io/FileUtils.java#L1482-L1483
141. https://github.com/openjdk/jdk/blob/da75f3c4ad5bdf25167a3ed80e51f567ab3dbd01/src/java.base/share/classes/java/net/URLConnection.java#L390
142. https://github.com/openjdk/jdk/blob/da75f3c4ad5bdf25167a3ed80e51f567ab3dbd01/src/java.base/share/classes/java/net/URLConnection.java#L433
143. https://github.com/openjdk/jdk/blob/da75f3c4ad5bdf25167a3ed80e51f567ab3dbd01/src/java.base/share/classes/java/util/zip/Deflater.java#L567-L573
144. https://github.com/openjdk/jdk/blob/da75f3c4ad5bdf25167a3ed80e51f567ab3dbd01/src/java.base/share/classes/jdk/internal/reflect/UnsafeStaticShortFieldAccessorImpl.java#L39-L41
145. https://github.com/openjdk/jdk/blob/da75f3c4ad5bdf25167a3ed80e51f567ab3dbd01/src/java.base/share/classes/jdk/internal/reflect/UnsafeStaticLongFieldAccessorImpl.java#L104
146. https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/time/package-summary.html
147. https://github.com/openjdk/jdk/blob/da75f3c4ad5bdf25167a3ed80e51f567ab3dbd01/src/java.base/share/classes/java/time/Period.java#L321
148. https://github.com/openjdk/jdk/blob/339ca887835d6456da9fcccdc32fb7716cbc60bb/src/java.base/share/classes/java/io/StringBufferInputStream.java#L113
149. https://github.com/openjdk/jdk/blob/339ca887835d6456da9fcccdc32fb7716cbc60bb/src/java.base/share/classes/java/io/Reader.java#L168
150. https://github.com/openjdk/jdk/blob/da75f3c4ad5bdf25167a3ed80e51f567ab3dbd01/src/java.base/share/classes/java/lang/Integer.java#L614
151. https://github.com/openjdk/jdk/commit/564011cff0667c6d34cf6aa46eedd11f2e01862b
152. https://github.com/apache/commons-lang/commit/656d2023dcd149018cd126e283f675b4ffef9715

153. https://github.com/apache/commons-io/commit/ff387016c2d95162aa6bf6735be47c559751b530
154. https://blog.joda.org/2012/11/javadoc-coding-standards.html
155. https://bytebuddy.net/#/develop
156. https://dubbo.apache.org/en/docs/v2.7/dev/checklist/
157. https://issues.apache.org/jira/browse/LANG-1681
158. https://github.com/apache/commons-lang/pull/1047
159. https://github.com/apache/commons-lang/blob/ce477d9140f1439c44c7a852d7df1e069e21cb85/src/main/java/org/apache/commons/lang3/Conversion.java#L373
160. https://github.com/apache/commons-lang/blob/ce477d9140f1439c44c7a852d7df1e069e21cb85/src/main/java/org/apache/commons/lang3/Conversion.java#L373
161. jqwik: Property-Based Testing in Java: https://jqwik.net/
162. https://github.com/apache/commons-lang/blob/ce477d9140f1439c44c7a852d7df1e069e21cb85/src/test/java/org/apache/commons/lang3/math/FractionTest.java#L437
163. https://github.com/apache/commons-text/blob/04748ac3693163685e411167e5c689eb9ae98dac/src/main/java/org/apache/commons/text/FormattableUtils.java#L90
164. https://docs.oracle.com/en/java/javase/18/docs/api/index.html
165. https://github.com/apache/commons-text/blob/04748ac3693163685e411167e5c689eb9ae98dac/src/main/java/org/apache/commons/text/StringSubstitutor.java#L742
166. https://github.com/openjdk/jdk/blob/da75f3c4ad5bdf25167a3ed80e51f567ab3dbd01/src/java.base/share/classes/java/util/Random.java#L383-L388
167. https://github.com/AlphaAutoLeak/zelix-injection/blob/master/src/main/java/zelix/utils/Utils.java#L256
168. JavaParser: https://github.com/javaparser/javaparser
169. https://github.com/openjdk/jdk/blob/da75f3c4ad5bdf25167a3ed80e51f567ab3dbd01/src/java.base/share/classes/java/lang/StringBuilder.java#L228C40-L228C40
170. https://github.com/feathersui/feathersui-starling-sdk/blob/master/modules/swfutils/src/java/flash/swf/tools/SwfxParser.java#L173
171. https://docs.oracle.com/en/java/javase/20/jshell/introduction-jshell.html
172. https://github.com/aaiyer/SuanShu/blob/ed9829aed161112e4d5fb5e2a1ab5ae05d99a491/src/main/java/com/numericalmethod/suanshu/vector/doubles/dense/operation/Basis.java#L83
173. https://github.com/openjdk/jdk/blob/da75f3c4ad5bdf25167a3ed80e51f567ab3dbd01/src/java.base/share/classes/java/util/concurrent/LinkedBlockingDeque.java#L182
174. https://github.com/microsphere-projects/microsphere-java/blob/main/microsphere-core/src/main/java/io/microsphere/convert/multiple/StringToBlockingDequeConverter.java#L31
175. https://github.com/msdeep14/getAheadWithMe/blob/main/LowLevelDesign/Concurrency/src/practice/ratelimiter/strategy/LeakyBucket.java#L15
176. https://github.com/openjdk/jdk/blob/da75f3c4ad5bdf25167a3ed80e51f567ab3dbd01/src/java.base/share/classes/java/util/ArrayDeque.java#L194
177. https://jspecify.dev/
178. https://github.com/jfree/jfreechart/blob/5aac9ae42147d34fe175e29af3993172e9c9080a/src/main/java/org/jfree/chart/JFreeChart.java#L257
179. https://github.com/openjdk/jdk/blob/da75f3c4ad5bdf25167a3ed80e51f567ab3dbd01/src/java.base/share/classes/java/util/Properties.java#L224
180. https://github.com/shtrih-m/javapos_shtrih/blob/master/Source/Core/src/com/shtrih/util/Localizer.java#L169

181. https://github.com/jenkinsci/pipeline-groovy-lib-plugin/blob/773332a145baaa64a936eb23019e92dc110f7bc0/src/main/java/org/jenkinsci/plugins/workflow/libs/LibraryAdder.java#L172C5-L172C5
182. https://github.com/jspecify/jspecify/wiki/nullness-design-FAQ
183. https://www.sonarsource.com/blog/sonarlint-quick-fixes/
184. https://www.sonarsource.com/products/sonarqube/whats-new/sonarqube-9-5/
185. https://www.sonarsource.com/blog/deeper-sast-uncovers-hidden-security-vulnerabilities/#behind-the-scenes-of-deeper-sast
186. https://www.sonarsource.com/products/sonarcloud/
187. https://www.uber.com/ch/en/about/science/
188. https://www.uber.com/us/en/about/
189. https://github.com/uber/piranha/
190. https://go.dev/
191. https://www.uber.com/en-SE/blog/go-monorepo-bazel/
192. https://monorepo.tools/#what-is-a-monorepo
193. https://www.uber.com/en-CH/blog/devpod-improving-developer-productivity-at-uber/
194. https://blog.pragmaticengineer.com/uber-engineering-levels/
195. https://spectrum.ieee.org/the-top-programming-languages-2023
196. https://www.apache.org/