jAST: Analyzing and Modifying Java ASTs with Python

Marius Smytzek
CISPA Helmholtz Center for Information Security
Saarbrücken, Germany
marius.smytzek@cispa.de

Martin Eberlein
Humboldt-Universität zu Berlin
Berlin, Germany
martin.eberlein@hu-berlin.de

Lars Grunske
Humboldt-Universität zu Berlin
Berlin, Germany
grunske@hu-berlin.de

Andreas Zeller
CISPA Helmholtz Center for Information Security
Saarbrücken, Germany
zeller@cispa.de

Abstract

Analyzing and modifying source code at the Abstract Syntax Tree (AST) level is fundamental to numerous software engineering tasks, including program analysis, instrumentation, and transformation. While Java-specific tools exist, they often operate at the bytecode level or are tightly coupled to the Java ecosystem, limiting their flexibility and accessibility. In this paper, we present jAST, a Python-based tool that generates and manipulates ASTs for Java programs. By leveraging Python's simplicity and extensive ecosystem, jAST enables precise source-level analysis, seamless integration with Python workflows, and support for advanced tasks such as feature extraction for debugging and learning-based failure analysis. jAST is an open-source tool offering researchers, educators, and practitioners an extensible framework for working with Java ASTs.

CCS Concepts

• **Software and its engineering → Syntax**; **Software libraries and repositories**; **Software maintenance tools**; *Parsers*.

Keywords

Java, AST, Python, Program Analysis, Instrumentation

ACM Reference Format:
Marius Smytzek, Martin Eberlein, Lars Grunske, and Andreas Zeller. 2025. jAST: Analyzing and Modifying Java ASTs with Python. In *33rd ACM International Conference on the Foundations of Software Engineering (FSE Companion '25), June 23–28, 2025, Trondheim, Norway*. ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3696630.3728598

1 Introduction

Software development involves numerous tasks such as code analysis, transformation, and refactoring, which require an in-depth understanding of program structure. Abstract Syntax Trees (ASTs) are a fundamental representation of source code. They capture its syntactic structure in a tree format widely used in compilers, program analysis tools, and other software engineering tasks. ASTs enable precise and efficient code manipulation by providing a structured, hierarchical representation of its elements.

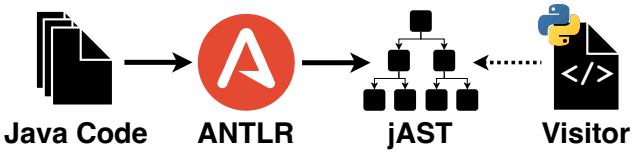

Figure 1: Overview of how jAST works. Java source code is parsed into a Parse Tree using an ANTLR-based parser and then converted into an Abstract Syntax Tree. The resulting AST, a tree of Python objects, can be traversed, analyzed, and modified using visitor and transformer patterns. Modified ASTs can then be unparsed into Java source code, enabling precise program analysis and transformations.

However, existing tools for generating ASTs for Java programs frequently have significant limitations. Many tools operate primarily on Java bytecode, which, while useful for specific analyses, does not necessarily preserve a direct relationship with the source code. This disconnect creates challenges for tasks that require source-level understanding, such as fault localization and program instrumentation. The absence of accurate and accessible source-level AST generation limits the ability of developers and researchers to perform advanced analyses or modifications.

The ability to operate directly on source code and modify it has broad applicability across various software engineering domains. For example, instrumentation is essential for fault localization and fuzzing tasks. Fault localization depends on source-level modifications to trace program behavior and pinpoint errors, while fuzzing benefits from injecting input-tracking mechanisms directly into source code. Tools enabling source-level program manipulation significantly enhance workflows in these domains, offering greater precision and flexibility than bytecode-level alternatives.

To address these challenges, we present jAST, a Python-based tool for generating and analyzing ASTs from Java programs. Python's simplicity, extensive library ecosystem, and ease of integration make it an ideal choice for building tools that support rapid prototyping and experimentation. By bridging the gap between Java program analysis and Python-based workflows, jAST facilitates a wide range of applications, including *(Static Analysis)* extracting program structure and dependencies to identify code smells or enforce coding standards, *(Program Instrumentation)* modifying source code to inject logging, tracking, or other functionality, and

FSE Companion '25, Trondheim, Norway

ACM ISBN 979-8-4007-1276-0/2025/06
https://doi.org/10.1145/3696630.3728598

(Refactoring and Transformation) automating the process of code simplification, optimization, or adaptation to new standards.

By focusing on the source code and leveraging Python's usability, jAST empowers developers and researchers to engage with Java programs in novel and productive ways. It provides the foundation for advancing software engineering practices, particularly in areas relying on accurate source-level instrumentation. Figure 1 illustrates the workflow of jAST consisting of parsing Java source code into a Parse Tree, converting it into an AST, and then allowing users to traverse, analyze, and modify the tree.

In this paper we make the following contributions:

(1) jAST, a Python tool for AST representation of Java programs.
(2) Support of AST instrumentation aiming at advanced software engineering tasks.
(3) An extensible and user-friendly framework.

The remainder of this paper is organized as follows: Section 2 provides the necessary background and Section 3 discusses related work. Section 4 outlines the specification of jAST, while Section 5 details its design and implementation. Section 6 demonstrates the usage of jAST and Section 7 explains the evaluation of out tool. Finally, Section 8 concludes with a discussion of future directions.

2 Background and Problem

ASTs are data structures that represent the syntactic structure of source code in a hierarchical tree format. They abstract away low-level syntax details, focusing on the structural and semantic relationships between code elements. For instance, in Java, an `if` statement would be represented in an AST as a parent node with child nodes for the condition and the body and an optional else. This representation enables efficient static analysis, code transformation, and optimization. ASTs play a central role in numerous software engineering tasks, including: (a) **Compilers**: ASTs form an intermediate representation in parsing and semantic analysis. (b) **Static Analysis Tools**: ASTs enable the detection of code smells, bugs, and vulnerabilities. (c) **Code Instrumentation**: Source-level ASTs are crucial for fault localization and input tracking, where bytecode-level representation often falls short. (d) **Program Transformation**: Refactoring and optimization tools rely on ASTs to ensure transformations preserve program semantics.

Many existing tools, such as ASM [1], operate on Java bytecode, enabling transformations at the compiled level. While bytecode-based instrumentation is effective for some tasks, it lacks a direct mapping to the source code, creating challenges for (a) source-level fault localization, where developers need precise correlations between source lines and execution traces and (b) advanced techniques like fuzzing, where input generation and tracking often require access to source-level constructs.

Tools that generate ASTs directly from source code provide better support for these tasks. For example, source-level transformations can ensure that injected logging or instrumentation is understandable and maintainable for developers.

Python has emerged as a popular choice for building software engineering research and development tools due to its simplicity, flexibility, and extensive library ecosystem. Existing Python-based tools like `javalang` [22] provide limited support for parsing Java code and lack the extensibility and precision required for source-level instrumentation and transformation. By providing a Python-based framework for Java AST generation, jAST enables researchers and developers to (a) leverage Python's libraries for advanced analysis (e.g., NumPy, pandas), (b) quickly prototype and experiment with AST-based workflows and (c) facilitates cross-language tool integration, bridging gaps between Java and Python ecosystems.

Unlike existing tools, jAST focuses on bridging the gap between Java source-level AST generation and Python-based workflows. By prioritizing simplicity, extensibility, and integration, jAST provides a flexible platform for research and practical applications in fault localization, fuzzing, and static analysis.

3 Related Work

Analyzing and manipulating source code at the Abstract Syntax Tree (AST) level is fundamental in software engineering, with numerous tools and frameworks developed across programming languages. Below, we survey relevant work and position jAST within the existing landscape.

Tools like Eclipse JDT [12] and Spoon [16] are prominent for Java AST analysis and transformation. Eclipse JDT provides parsing and manipulation capabilities but is tightly coupled with the Eclipse IDE, limiting its flexibility outside IDE-based workflows. Spoon offers fine-grained source-level transformations and is widely adopted in research and industry; however, it is inherently tied to the Java ecosystem, requiring expertise in Java-specific workflows.

Bytecode manipulation frameworks such as ASM [1] enable instrumentation and dynamic analysis of Java programs. These tools operate at the bytecode level, offering performance advantages but sacrificing the ability to perform precise source-level analysis, which is often critical for debugging and program comprehension.

The emergence of polyglot software systems has driven the development of cross-language AST frameworks like Tree-sitter [2], which supports efficient parsing for multiple programming languages. While Tree-sitter excels in syntax highlighting and extensibility, it provides limited functionality for in-depth program analysis or transformations specific to Java.

Python's versatility has led to tools such as LibCST [4] and astroid [11], focusing on Python-specific AST (or concrete syntax tree) manipulation with user-friendly interfaces. However, these tools do not extend their capabilities to languages like Java, creating a gap for cross-language workflows.

Beyond these, tools like ANTLR [14, 15] provide parser generation for numerous languages, including Java and Python, but lack direct AST manipulation capabilities. IntelliJ IDEA's PSI [8] offers advanced Java code insight features, but it is proprietary and confined to the IntelliJ environment.

Tools such as Babel [13] for JavaScript and Roslyn [6] for .NET languages showcase the importance of providing language-native AST frameworks. However, these tools are not designed for cross-language use or Java-specific needs. Frameworks like CheerpJ [21], which translate Java bytecode into JavaScript, also demonstrate innovative cross-platform approaches but do not address source-level manipulation. Additional tools and frameworks have emerged to address specific needs in AST analysis. JavaParser [23] offers a lightweight Java library for parsing and manipulating Java source

code, facilitating tasks like code generation and analysis. PMD [10] is a static code analyzer that uses ASTs to detect programming flaws, such as unused variables and empty catch blocks. Checkstyle [7] focuses on enforcing coding standards by analyzing the AST structure of Java code. For more advanced analysis, tools like ACER [3] to extract call graphs from ASTs and PSIMiner [20] to support machine learning assist in understanding program structure and dependencies.

These tools highlight the diverse applications of AST analysis for Java, ranging from code quality enforcement to facilitating complex program analyses. jAST differentiates itself by offering a Python-based approach to Java AST manipulation, bridging the gap between Java codebases and Python's extensive analysis ecosystem.

4 Specifaction

Figure 2 shows an excerpt of the specification of jAST. The specification defines the structure of the AST nodes that jAST generates. The AST nodes are orientated toward Python's ast module, which makes it easy to work with the generated ASTs in Python.

5 Design and Implementation

jAST is designed to provide a lightweight and extensible framework for generating and manipulating Java Abstract Syntax Trees (ASTs) in Python. We rely on ANTLR [14, 15] to parse Java source code into a Parse Tree. Using ANTLR allows us to easily update the grammar for new Java versions and generate a parser for the Java language. The Parse Tree is then converted into an AST, represented as a tree of Python objects, similar to Python's built-in AST module.

This design enables users to traverse, analyze, and modify the AST using Python's intuitive syntax and extensive library ecosystem. The resulting AST can be transformed using visitor and transformer patterns, allowing users to extract information, perform analyses, and modify the tree structure. Finally, the modified AST can be returned to the Java source code, preserving all modifications made to the tree.

6 Usage

jAST provides an intuitive and Pythonic interface for analyzing and modifying Java Abstract Syntax Trees (ASTs). The tool allows users to parse Java source code into an AST, traverse and modify the tree, and convert the modified AST back into Java source code. This section describes its installation, key functionalities, and usage.

Installation. jAST is available on the Python Package Index (PyPI) and can be installed using the following command:

```
pip install java-ast
```

Parsing Java Source Code. Java source code can be parsed into an AST using the `parse()` function. The resulting tree consists of objects whose classes inherit from `jast.JAST`. For example:

```
import jast

# Parse a Java source file
with open("HelloWorld.java") as file:
    tree = jast.parse(file.read())
```

Here, `tree` represents the AST of the Java source code.

```
-- builtin types are:
-- identifier, int, float, bool, char, string

module Java {
    mod = CompilationUnit(Package? package,
            Import* imports, declaration* body)
        | ModularUnit(Import* imports, Module body)

    declaration = EmptyDecl()
        | CompoundDecl(declaration* body)
        | Package(Annotation* annotations, qname name)
        | Import(bool? static, qname name,
            bool? on_demand)
        | Module(bool? open, qname name,
            directive* directives)
        | Field(modifier* modifiers, jtype type,
            declarator+ declarators)
        | Method(modifier* modifiers,
            typeparams? type_params,
            Annotation* annotations, jtype return_type,
            identifier id, params? parameters,
            dim* dims, qname* throws, Block? body)
        | Constructor(modifier* modifiers,
            typeparams? type_params, identifier id,
            params? parameters, Block body)
        | Class(modifier* modifiers, identifier id,
            typeparams? type_params, jtype? extends,
            jtype* implements, jtype* permits,
            declaration* body)
                                ...
        attributes(int lineno, int col_offset,
            int end_lineno, int end_col_offset)
                                ...
    stmt = Empty()
        | Block(stmt* body)
        | Compound(stmt* body)
                                ...
        | If(expr test, stmt body, stmt? orelse)
        | Switch(expr value, switchblock body)
        | While(expr test, stmt body)
                                ...
        | Expr(expr value)
        | Return(expr? value)
        | Yield(expr value)
        | Break(identifier? label)
        | Continue(identifier? label)
        | Synch(expr lock, Block body)
        attributes(int lineno, int col_offset,
            int end_lineno, int end_col_offset)

    expr = Assign(expr target, operator? op, expr value)
        | IfExp(expr test, expr body, expr orelse)
        | BinOp(expr left, operator op, expr right)
        | InstanceOf(expr value, (jtype | pattern) type)
        | UnaryOp(unaryop op, expr operand)
        | PostOp(expr operand, operator op)
                                ...
        | Constant(literal value)
        | Name(identifier id)
        | ClassExpr(jtype type)
                                ...
        attributes(int lineno, int col_offset,
            int end_lineno, int end_col_offset)
                                ...
```

Figure 2: Excerpt of the specifaction of jAST. This excerpt does not represent the Java grammar but the type structure of the AST nodes.

Visiting Nodes in the AST. jAST provides a visitor pattern for traversing the AST. For instance, the following code demonstrates how to print the names of all classes in the parsed tree:

```
class NameVisitor(jast.JNodeVisitor):
    def visit_Identifier(self, node):
        print(node.name)

visitor = NameVisitor()
visitor.visit(tree)
```

In this example, the `visit_Identifier()` method is invoked for each `Identifier` node in the tree, allowing users to extract information from specific nodes.

Modifying Nodes in the AST. jAST also supports transformations of the AST using a node transformer. For example, to rename a class:

```
class NameModifier(jast.JNodeTransformer):
    def visit_Identifier(self, node):
        if node.name == "HelloWorld":
            node.name = "HelloWorld2"
        return node

modifier = NameModifier()
tree = modifier.visit(tree)
```

This example demonstrates a transformation that renames the class `HelloWorld` to `HelloWorld2`.

Writing Modified Java Source Code. Once modifications are made, the updated AST can be written back to Java source code using the `unparse()` function:

```
with open("HelloWorld2.java", "w") as file:
    file.write(jast.unparse(tree))
```

The `unparse()` function converts the AST back into valid Java source code, preserving all modifications.

Helper Functions and Extensibility. jAST includes various helper functions to simplify AST manipulation. These features make jAST an ideal choice for tightly integrated workflows with Java, enabling tasks such as program analysis, instrumentation, and source-level transformation.

7 Evaluation

We evaluated jAST by applying it systematically to each possible Java declaration, statement, and expression, including various values and operators. We distinguished between parsing strings to ASTs and the ASTs themselves and unparsing the ASTs to verify the correctness of all tool parts. We produced unit tests for all these cases, resulting in an extensive test suite covering all aspects of the tool (indeed, we achieved 100% Moreover, we evaluated our tool on a set of real-world Java projects, including open-source libraries and applications from the Defects4J benchmark [9], to assess its usability in practical scenarios.

The evaluation demonstrated that jAST successfully parsed Java source code into accurate AST representations, allowing for precise analysis and modification. The tool's performance was satisfactory, with parsing times comparable to the existing `javalang` [22] and unparsing times even faster. However, since, to the best of our knowledge, jAST is the first tool to provide a Python-based AST unparsing for Java, we could not compare it to other tools. With this evaluation, we are convinced that jAST is a valuable addition to the software engineering tool landscape, providing a flexible framework for Java AST manipulation.

8 Conclusion and Future Work

In this paper, we presented jAST, a Python-based tool for generating Abstract Syntax Trees (ASTs) from Java source code. jAST bridges the gap between Java source-level program analysis and Python-based workflows, enabling researchers and developers to perform advanced program analysis, instrumentation, and transformation tasks efficiently. By leveraging Python's simplicity and extensive ecosystem, jAST offers a lightweight framework that facilitates cross-language workflows and supports various applications, such as fault localization and static analysis.

Our framework has several advantages over existing tools, including: (1) Precise source-level AST generation, avoiding the limitations of bytecode-level tools. (2) Integration with Python libraries, enabling rapid prototyping and advanced analysis workflows. (3) Extensibility, making it suitable for educational purposes, research, and industry applications.

The current implementation of jAST opens up numerous possibilities for future work, leveraging its capabilities to address advanced challenges in program analysis and debugging:

Feature Extraction for Debugging We are currently working on extracting execution features, such as variable usage patterns, control flow properties, and function call relationships, to assist in debugging and fault localization by integrating jAST into SFLKit [19] and our debugging pipeline [5, 17] that identifies failure patterns and learn from these features, aiding in automated failure prediction and debugging for input and execution space, which allows to run it on benchmarks like Defects4J [9] instead of Python benchmarks [18, 24].

Dynamic Instrumentation We want to expand jAST to support dynamic instrumentation, enabling real-time tracking of execution paths and runtime states for debugging and profiling.

Semantic Analysis We plan to leverage jAST to perform deeper semantic analyses of Java programs, such as type inference, dependency tracking, and detection of semantic errors.

Tool and Data Availability Statement

Our tool is available as open-source:

https://github.com/smythi93/jast

Acknowledgments

This research was partially funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) — ZE 509/7–2 and GR 3634/4–2 Emperor (261444241) and by the European Union (ERC S3, 101093186). Views and opinions expressed are however those of the authors only and do not necessarily reflect those of the European Union or the European Research Council. Neither the European Union nor the granting authority can be held responsible for them.

References

[1] Eric Bruneton, Romain Lenglet, and Thierry Coupaye. 2002. ASM: A code manipulation tool to implement adaptable systems. (01 2002).

[2] Max Brunsfeld, Amaan Qureshi, Andrew Hlynskyi, Patrick Thomson, ObserverOfTime, Will Lillis, Josh Vera, dundargoc, Phil Turnbull, Timothy Clem, Douglas Creager, Andrew Helwer, Rob Rix, Daumantas Kavolis, Hendrik van Antwerpen, Michael Davis, Christian Clason, Ika, Amin Ya, Riley Bruins, Tuấn-Anh Nguyễn, Stafford Brunk, Matt Massicotte, bfredl, Niranjan Hasabnis, Mingkai Dong, Samuel Moelius, Steven Kalt, and Kolja. 2025. *tree-sitter/tree-sitter: v0.25.3.* https://doi.org/10.5281/zenodo.14969376

[3] Andrew Chen, Yanfu Yan, and Denys Poshyvanyk. 2023. ACER: An AST-based Call Graph Generator Framework. In *2023 IEEE 23rd International Working Conference on Source Code Analysis and Manipulation (SCAM).* 254–259. https://doi.org/10.1109/SCAM59687.2023.00035

[4] LibCST Contributors. 2020. LibCST: A Concrete Syntax Tree Parser for Python. https://github.com/Instagram/LibCST.

[5] Martin Eberlein, Marius Smytzek, Dominic Steinhöfel, Lars Grunske, and Andreas Zeller. 2023. Semantic Debugging. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (San Francisco, CA, USA) *(ESEC/FSE 2023).* Association for Computing Machinery, New York, NY, USA, 438–449. https://doi.org/10.1145/3611643.3616296

[6] .NET Foundation and Contributors. 2025. Roslyn: The .NET Compiler Platform. https://github.com/dotnet/roslyn.

[7] Roman Ivanov, Richard Veach, Pavel Bludov, Andrei Paikin, Nick Mancuso, Ilja Dubinin, Andrei Selkin, Vladislav Lisetskii, Oliver Burn, Michał Kordas, Ruslan Diachenko, Baratali Izmailov, Daniil Yaroslavtsev, Ivan Sopov, Lars Kühne, Rick Giles, Oleg Sukhodolsky, Michael Studman, and Travis Schneeberger. 2025. Checkstyle. https://checkstyle.sourceforge.io/.

[8] JetBrains. 2024. Program Structure Interface (PSI). https://plugins.jetbrains.com/docs/intellij/psi.html.

[9] René Just, Darioush Jalali, and Michael D. Ernst. 2014. Defects4J: a database of existing faults to enable controlled testing studies for Java programs. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis* (San Jose, CA, USA) *(ISSTA 2014).* Association for Computing Machinery, New York, NY, USA, 437–440. https://doi.org/10.1145/2610384.2628055

[10] InfoEther, LLC. 2025. PMD: Source Code Analyzer. https://pmd.github.io/.

[11] Logilab and Astroid Contributors. 2025. Astroid. https://github.com/pylint-dev/astroid.

[12] Andrey Loskutov, Jay Arthanareeswaran, Kalyan Prasad Tatavarthi, mateusz matela, Noopur Gupta, Sarika Sinha, Sravan Kumar Lakkimsetti, and Vikas Chandra. 2003. Eclipse Java Development Tools (JDT). https://github.com/eclipse-jdt.

[13] Sebastian McKenzie, Brian Ng, Henry Zhu, Huáng Jùnliàng, Logan Smyth, Nicolò Ribaudo, Sven Sauleau, and other contributors. 2025. Babel. https://babeljs.io/.

[14] Terence Parr. 2013. *The Definitive ANTLR 4 Reference.* Pragmatic Bookshelf.

[15] Terence Parr. 2023. *ANTLR (Another Tool for Language Recognition).* ANTLR Project. https://www.antlr.org/ Accessed: 2025-01-14.

[16] Renaud Pawlak, Martin Monperrus, Nicolas Petitprez, Carlos Noguera, and Lionel Seinturier. 2016. SPOON: A library for implementing analyses and transformations of Java source code. *Softw. Pract. Exper.* 46, 9 (Sept. 2016), 1155–1179. https://doi.org/10.1002/spe.2346

[17] Marius Smytzek, Martin Eberlein, Lars Grunske, and Andreas Zeller. 2025. How Execution Features Relate to Failures: An Empirical Study and Diagnosis Approach. https://doi.org/10.48550/arXiv.2502.18664 arXiv:2502.18664 [cs.SE]

[18] Marius Smytzek, Martin Eberlein, Batuhan Serçe, Lars Grunske, and Andreas Zeller. 2024. Tests4Py: A Benchmark for System Testing. In *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering* (Porto de Galinhas, Brazil) *(FSE 2024).* Association for Computing Machinery, New York, NY, USA, 557–561. https://doi.org/10.1145/3663529.3663798

[19] Marius Smytzek and Andreas Zeller. 2022. SFLKit: a workbench for statistical fault localization. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Singapore, Singapore) *(ESEC/FSE 2022).* Association for Computing Machinery, New York, NY, USA, 1701–1705. https://doi.org/10.1145/3540250.3558915

[20] Egor Spirin, Egor Bogomolov, Vladimir Kovalenko, and Timofey Bryksin. 2021. PSIMiner: A Tool for Mining Rich Abstract Syntax Trees from Code. In *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR).* 13–17. https://doi.org/10.1109/MSR52588.2021.00014

[21] Leaning Technologies. 2024. CheerpJ: The complete Java runtime for modern browsers. https://cheerpj.com/.

[22] Chris Thunes. 2013. javalang. https://github.com/c2nes/javalang

[23] Danny van Bruggen, Federico Tomassetti, Roger Howell, Malte Langkabel, Nicholas Smith, Artur Bosch, Malte Skoruppa, Cruz Maximilien, ThLeu, Panayiotis, Sebastian Kirsch, Simon, Johann Beleites, Wim Tibackx, jean pierre L, André Rouél, edefazio, Daan Schipper, Mathiponds, Why you want to know, Ryan Beckett, ptitjes, kotari4u, Marvin Wyrich, Ricardo Morais, Maarten Coene, bresai, Implex1v, and Bernhard Haumacher. 2020. *javaparser/javaparser: Release javaparser- parent-3.16.1.* https://doi.org/10.5281/zenodo.3842713

[24] Ratnadira Widyasari, Sheng Qin Sim, Camellia Lok, Haodi Qi, Jack Phan, Qijin Tay, Constance Tan, Fiona Wee, Jodie Ethelda Tan, Yuheng Yieh, Brian Goh, Ferdian Thung, Hong Jin Kang, Thong Hoang, David Lo, and Eng Lieh Ouh. 2020. BugsInPy: a database of existing bugs in Python programs to enable controlled testing and debugging studies. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Virtual Event, USA) *(ESEC/FSE 2020).* Association for Computing Machinery, New York, NY, USA, 1556–1560. https://doi.org/10.1145/3368089.3417943