Annotation-Based Static Verification of Algorithmic Complexity in Java

Aleksandr V. Samedov[1], Zoltán Porkoláb[1]

[1]ELTE Eötvös Loránd University, Faculty of Informatics, 1117 Budapest, Pázmány Péter sétány 1/C, Hungary

Abstract

Time and space complexities are fundamental concerns in modern software development, particularly in performance-critical systems. This paper proposes a novel annotation-based approach for verifying algorithmic complexity in Java. By integrating complexity metadata into the source code through Java annotations, developers can automatically check whether the implementations conform to the expected computational bounds. The system leverages static analysis techniques and a custom framework built using JavaParser. Evaluation across common algorithm patterns and real-world projects demonstrated the effectiveness and feasibility of this lightweight and developer-friendly method.

Keywords

Algorithmic Complexity, Static Analysis, Java Annotations, Software Verification, Code Quality

1. Introduction

Managing computational complexity is a foundational concern in software engineering, especially as systems become larger, more data-intensive, and increasingly performance sensitive. Efficient algorithms are crucial for improving runtime performance and ensuring scalability, maintainability, and resource predictability. In domains such as distributed systems [1], embedded systems [2], and real-time data processing, understanding and controlling complexity directly impact system reliability and usability.

Traditional methods for assessing algorithmic complexity include manual analysis guided by theoretical frameworks such as Big-O, Big-Theta, and Big-Omega notations [3], as well as structural code metrics such as McCabe's cyclomatic complexity [4] and Halstead's effort and volume measures [5]. These metrics are valuable but are typically used post hoc and require interpretation by experienced developers. Moreover, they are not tightly integrated into modern development workflows, making it difficult to apply them consistently across teams and codebases.

Static analysis tools, such as FindBugs, PMD, and Checkstyle [6] offer automated checks for code quality and conformance to style and safety rules. However, these tools generally lack mechanisms for detecting or enforcing algorithmic complexity bounds. More formal methods, such as abstract interpretation [7] and worst-case execution time (WCET) analysis [2], provide rigorous performance guarantees, particularly in embedded and safety-critical contexts; however, they are often too heavyweight for general-purpose software development. Additionally, they are rarely designed with developer ergonomics.

Recent trends in software verification and performance analysis have explored the use of annotations as a lightweight means of expressing program properties at the source level. For example, annotations have been used to guide the static verification of correctness [8] and to model performance characteristics [9]. These approaches offer a promising trade-off between precision and usability because they enable developers to embed semantic expectations directly into the code.

SQAMIA 2025: Workshop on Software Quality Analysis, Monitoring, Improvement, and Applications, September 10–12, 2025, Maribor, Slovenia

emsmx4@inf.elte.hu (A. V. Samedov); gsd@inf.elte.hu (Z. Porkoláb)

http://samedovav.web.elte.hu (A. V. Samedov); https://gsd.web.elte.hu (Z. Porkoláb)

0009-0008-1853-0292 (A. V. Samedov); 0000-0001-6819-0224 (Z. Porkoláb)

In this study, we extend this direction by introducing a novel annotation-based approach for time complexity verification in Java programs. Our system enables developers to specify the expected complexity using annotations such as `@Prove` and `@BelieveMe` [10]. These annotations are processed during compilation or static analysis, leveraging the JavaParser framework [11] to construct abstract syntax trees (ASTs) and apply complexity detectors that recognize common patterns, such as loop nesting, recursion, and use of built-in sorting APIs.

This annotation-based model offers several advantages: (1) it allows complexity expectations to be embedded directly in the source code, improving traceability and documentation; (2) it enables automated checking without requiring a separate specification language; and (3) it offers immediate feedback to developers during builds, helping enforce performance constraints throughout the software lifecycle.

This paper is organized as follows: In Section 2, we review the foundational and recent literature on code complexity metrics and static analysis. Section 3 describes our proposed approach and the annotation design. Section 4 outlines the architecture of our prototype tool, and Section 5 presents an empirical evaluation of its accuracy and performance. We conclude in Section 6 with a summary and directions for future work.

2. Related Work

In this Section we review the state of the art literature on static analysis approach for complexity measurement and on annotation-based complexity analysis related to our results.

2.1. Static Analysis for Complexity Measurement

Cousot and Cousot [7] pioneered the concept of abstract interpretation, a fundamental approach in static analysis that facilitates the estimation of program behaviors, such as complexity assessment. Their framework underpins contemporary methods for complexity detection used in compilers and analytical tools.

WCET Analysis. A significant contribution to the field was presented by Schoeberl et al. [12], who investigated the worst-case execution time (WCET) analysis specific to a Java processor. The authors explored various techniques for establishing upper limits on execution time, emphasizing Java's execution model and the constraints of real-time systems. Their findings underscore the necessity of precise execution time predictions for essential applications, particularly in embedded and real-time environments. Additionally, this study offers valuable perspectives on how contemporary processors manage the dynamic characteristics of Java, which directly affect complexity analysis.

Furthermore, conventional techniques for complexity analysis, as outlined in [13], offer fundamental insights into algorithm efficiency. When these methods are integrated with Worst-Case Execution Time (WCET) analysis, they facilitate a more thorough evaluation of software performance, particularly in the context of applications that are sensitive to timing.

Scalable Static Analysis. More recent work by Lillack et al. [14] introduced scalable static analysis methods to efficiently assess the code complexity in large-scale Java applications. Their system focuses on reducing the computational overhead while maintaining the precision of detecting costly operations.

Compile-Time Complexity Verification via Templates. A novel approach to complexity evaluation was presented by Corriero et al. [15], who developed a tool that leveraged C++ templates for the compile-time certification of polynomial-time computability. Their method demonstrates how language-level constructs can be used not only for abstraction but also as a formal mechanism for certifying computational limits. By encoding complexity logic into template instantiations, their approach enables the detection of non-polynomial patterns at compile time, ensuring that the analyzed code conforms to the desired complexity constraints. This technique, rooted in C++, provides a conceptual foundation for annotation-based approaches in Java, where the compile-time metadata can similarly enforce algorithmic limits.

2.2. Annotation-Based Complexity Analysis

Dynamic Annotation-Based Code Analysis. The use of annotations to enhance static analysis has been investigated in several fields. Ernst et al. [16] introduced annotation-based code analysis aimed at enhancing program correctness, which is consistent with our methodology of employing Java annotations to articulate and identify computational complexity.

Java Annotations for Performance Modelling. Krämer et al. [17] developed a framework that leverages Java annotations for performance modelling, enabling developers to highlight code sections critical to performance. Our research builds upon this concept by concentrating specifically on annotations related to time and space complexity, rather than general performance modelling.

3. Methodology

The intricacy of software plays a pivotal role in determining the maintainability, scalability, and efficiency of an application. As digital systems expand in both size and capability, their codebases tend to become more elaborate, leading to increased technical debt and slower development time lines. When functions are overly complex, they become more difficult to interpret, troubleshoot, and update, which can negatively affect team productivity and the long-term viability of a project [4]. Thus, evaluating and managing complexity at the function level is crucial in large-scale systems and applications with stringent performance requirements [1].

Traditionally, assessing code complexity involves thorough manual review and the use of external static analysis utilities. Although these methods are effective, they often require significant setup and maintenance efforts. In addition, such tools usually operate separately from the main development environment, which can result in disjointed workflows and reduced efficiency.

In contrast, Java annotations provide a streamlined and declarative method for embedding complexity-related information directly into the source code [18]. This enables developers to flag complexity expectations where the code is written, making it easier to identify potential issues. It also promotes better documentation of performance assumptions and reduces the reliance on third-party tools. Using annotations, engineers can indicate the anticipated time or memory complexity of the methods, which can be verified during compilation or through dedicated analysis tools. This localized strategy supports faster feedback and encourages sound software design principles.

Incorporating annotations into complexity monitoring signifies a progressive transition towards more automated and developer-centric approaches for managing software complexity within contemporary engineering methodologies.

3.1. Problem Statement

Manual evaluation of the complexity of software functions is often labor-intensive and susceptible to human error, particularly in extensive systems involving multiple modules and contributors. As the volume and intricacy of codebases increase, the practicality of relying solely on manual inspections declines, often resulting in inconsistent evaluations and missed issues in complex governance systems. Although tools such as Checkstyle and PMD offer helpful guidance on code quality and surface-level complexity indicators, they typically require considerable configuration and are not well-suited for precise method-level complexity enforcement [19]. These solutions are generally geared toward broad code quality checks and lack native support for declaring and validating complexity expectations directly in the code.

To address these limitations, this study presents a Java-oriented solution for incorporating complexity expectations via annotations. This method entails integrating complexity-related specifications directly within the method definitions, facilitating automated verification, either during compilation or through additional analytical tools. A working prototype was developed to assess the feasibility of the proposed approach. This endeavor encounters several significant challenges, such as automating the assessment

of algorithmic complexity, navigating the limitations of existing static analysis tools, and maintaining the system's efficiency, usability, and compatibility with contemporary development environments.

3.2. Objectives

The main aim of this study is to develop and deploy a Java-based annotation system focused on assessing and enforcing complexity at the functional level. This involves the creation of automated tools designed to analyze annotated code segments to ensure that they meet established complexity criteria. Additionally, this study evaluates the effectiveness and practicality of this annotation-based method using traditional static analysis techniques. In addition to the technical aspects, this study aspires to offer actionable recommendations for incorporating this approach into existing development practices, ultimately striving to improve software maintainability, enhance code clarity, and increase performance awareness throughout the development lifecycle.

3.3. Scope and Limitations

This study investigates the utilization of Java annotation features to enhance static analysis for evaluating code complexity. The primary focus was on assessing the time complexity of the methods, which are characterized by significant computational demands. To illustrate the practicality of this approach, a prototype tool was created that employed annotations to track and enforce complexity constraints at the function-level.

However, the proposed approach has some limitations. This is ineffective for codes that are generated dynamically (e.g., reflection), as such codes cannot be analyzed using static analysis methods. Furthermore, the tool may struggle to accurately capture complex behaviors in large or highly diverse systems. Another significant limitation is the potential performance overhead associated with processing annotations which may extend both the build time and execution duration.

3.4. Overview of the Proposed Approach

A complexity analysis system based on annotations was developed using a three-phase approach. Initially, Java annotations were implemented to enable developers to directly define complexity limits within the codebase. Subsequently, a static analysis process was used during the compilation phase to evaluate the complexity metrics of the functions. In the final phase, a reporting framework was established to identify functions that surpassed the specified thresholds, thus offering developers practical insights.

This methodology draws inspiration from established static analysis methods [18], focusing on enforcing complexity in real-time scenarios using Java's Reflection API and Annotation Processing Tool (APT). Additionally, concepts from the Hume programming language [20] were examined to determine their relevance in managing complexity constraints in embedded and real-time systems.

3.5. Java Annotations: Overview and Applications

Java annotations serve as metadata components integrated into the source code and impact behavior during both the compile and runtime. They can be utilized for various elements such as classes, methods, variables, parameters, and packages. Their applications are extensive, particularly in areas such as code instrumentation, validation, and configuration [21].

Java annotations can be categorized into different types based on their structure and intended functions. Marker annotations represent the most basic type, lacking any method, and are primarily used to indicate that certain declarations require special handling. Single-element annotations consist of a single parameter and offer a shorthand syntax, making them efficient in situations in which only one value is necessary. In contrast, normal annotations are more intricate, permitting multiple parameters that must be explicitly defined. Finally, meta-annotations are those that apply to other annotations,

Table 1

Java Annotations Categories

Java Annotations Categories	
Category	**Use Case**
Predefined	Built-in Java annotations such as @Override and @Deprecated.
Custom	User-defined annotations for specific needs, such as complexity analysis.
Meta-annotations	Annotations that describe other annotations, such as @Target and @Retention.

specifying behaviors such as the permissible contexts for the use of an annotation or its retention duration.

In addition to these structural classifications, annotations can be grouped according to their specific applications. Table 1 summarizes the primary categories of Java annotations, including predefined, custom, and meta-annotations, along with the common use cases associated with each category.

Annotations provide an efficient means of enforcing coding standards, making them ideal for complexity analysis.

3.6. Parsing and Analysis Using JavaParser

JavaParser was used to create an abstract syntax tree (AST) from the source code, facilitating systematic and programmatic examination of Java applications. The AST enables the extraction of function definitions and the analysis of their structures to determine the time complexity and other relevant metrics.

To ensure smooth integration into the build process, the annotations are handled during compilation using the Java Annotation Processing Tool (APT). This configuration allows analysis to be conducted without requiring separate tools or workflows for developers.

3.7. Experimental Setup and Evaluation Criteria

An annotation-based complexity analysis system was developed and preserved across two code repositories. The primary repository encompasses an essential system tasked with analyzing annotated Java methods and enforcing complexity limits. This repository comprises definitions of annotations, parsing mechanisms utilizing JavaParser, and modules for the metric computations.

Conversely, the secondary repository was dedicated to a series of Java projects that were specifically designed for assessment purposes. These projects serve as experimental platforms for measuring performance, validating correctness, and comparing results with those of other existing tools. They feature examples that exhibit a range of functional complexities to evaluate the detection accuracy and runtime efficiency of the system.

The proposed methodology was assessed using several essential metrics. Initially, the precision of the complexity measurement was examined to confirm that the calculated metrics corresponded with the anticipated outcomes and theoretical frameworks. Subsequently, the performance overhead resulting from annotation processing is evaluated, particularly during the compilation and code analysis phases. Finally, a comparison was made between the system and current static analysis tools to evaluate their effectiveness and possible benefits.

These assessment criteria offer valuable insights into the technical viability and practical utility of employing Java annotations for complexity analyses in real-world development settings.

The proposed system introduces two key Java annotations to facilitate complexity verification. The `@Prove` annotation allows developers to explicitly declare the expected time complexity of a method, along with relevant input size variables and count expressions. In contrast, the `@BelieveMe` annotation enables developers to assert complexity expectations in cases where static verification is infeasible, effectively delegating trust to programmers.

To assess complexity, the system employs a set of custom detectors that identify structural patterns corresponding to standard complexity classes, including $O(1)$, $O(n)$, $O(\log n)$, $O(n^2)$, $O(2^n)$, and $O(n!)$. These detectors analyze features such as loop nesting depth, the presence of recursive calls, and the use of well-known algorithmic structures (e.g., sorting routines and depth-first search).

The analysis pipeline constructs an abstract syntax tree (AST) using the JavaParser framework [11]. During the annotation processing phase, the detectors traverse the AST to extract complexity indicators, which are then compared with the declared expectations provided by the annotations. This integration enables the compile- or build-time verification of algorithmic complexity within the development workflow.

4. Implementation

4.1. Annotation-Based Complexity Checker

Several meta-annotations must be specified to develop custom annotations to define the storage and targets. Several types of meta-annotations exist in Java.

- @Retention - Specifies how long the annotation should be stored
- @Target - Indicates what elements the annotation can be applied to
- @Inherited - specifies whether the created annotation can be automatically applied to descendant classes.
- @Documented - Indicates whether it is necessary to save information on how to use the specified annotation when running the application in the automatically generated documentation

4.2. @Prove Annotation

The `@Prove` annotation is used to determine the expected time complexity of a method. This allows developers to explicitly define the theoretical complexity class, specify the variable that represents the input size, and list the expressions that contribute to the operational count. This annotation supports compile-time or static analysis by providing metadata that can be processed using analysis tools to verify compliance with a specified complexity.

An example for the use of `@Prove` annotation, see Listing 1:

```
1 @Retention(RetentionPolicy.CLASS)
2 @Target({ElementType.METHOD})
3 public @interface Prove {
4   Complexity complexity()
5   String n();
6   String[] count()
7 }
```

Listing 1: Usage example for the @Prove Annotation

4.3. @BelieveMe Annotation

The `@BelieveMe` annotation, as seen on Listing 2 is intended for use in cases where complexity cannot be statically verified or where the developer assumes responsibility for its correctness. This is particularly useful in complex algorithms or run-time-dependent constructs, where automated analysis may fall short. This annotation enabled runtime retention, allowing optional runtime validation or documentation of the intended complexity for future maintenance.

```java
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.METHOD, ElementType.LOCAL_VARIABLE})
public @interface BelieveMe {
  Complexity complexity();
}
```

Listing 2: Usage of the @BelieveMe Annotation

4.4. Detectors

The system was designed with multiple dedicated detector components to improve clarity, modularity, and maintainability of the system. Each detector is responsible for analyzing a specific complexity metric, such as recursion, factorials, or data structures. This architectural decision follows the Single Responsibility Principle (SRP), which advocates that a class or component should have only one reason to change. The SRP is a key concept in object-oriented design principles and contributes to better cohesion and reduced coupling among components [22].

The system is easier to extend and test by assigning each complexity metric to its corresponding detector. New metrics can be incorporated by implementing additional detectors without affecting the existing logic. Furthermore, this separation of concerns allows for more granular performance tuning and parallelization of complexity checks in future research.

Each detector operates by visiting abstract syntax tree (AST) nodes using the JavaParser library and accumulating the metric-specific values. Once the analysis is complete, the detectors return their results to a central complexity controller that evaluates the metrics against the thresholds defined by Java annotations.

4.4.1. Recursion Detector

The Recursion Detector has two methods of checking: first, it checks for calls of the given function inside it, and second, it checks for multiple calls of the function.

4.4.2. Logarithmic Complexity Detector

Logarithmic complexity often arises in algorithms that repeatedly divide the input in half (such as binary search or heap operations), iterate over tree-based collections, or contain loops that halve the data size during each iteration. The following algorithm outlines the detection process for such patterns by analyzing the structure of the method and recursively checking for specific complexity indicators: It integrates three heuristics: logarithmic loop identification, recursion, and iteration collection.

4.4.3. Factorial Complexity Detector

Factorial time complexity $\mathcal{O}(N!)$ is often associated with recursive backtracking algorithms that explore all permutations or combinations of elements, which are common in problems such as the Traveling salesman problem or N-Queens problem. The following algorithm outlines the detection strategy for identifying such patterns by combining recursion, loops and backtracking control flows.

This detection is based on a combination of structural cues that strongly suggest factorial complexity. Similar to the logic proposed in [23], recursion combined with branching and iterations often indicates an exponential or factorial growth pattern.

4.4.4. Built-In Sort Detector

Java provides highly optimized sorting methods, such as `Collections.sort()`, `Arrays.sort()`, and `Arrays.parallelSort()`, all of which have known complexity behaviors (typically $\mathcal{O}(N \log N)$). Identifying these standard API calls allows the analysis tool to automatically classify the methods that delegate sorting tasks to Java's built-in mechanisms.

This detection approach leverages Java API documentation and behavior guarantees to automatically infer complexity without analyzing the internal logic of the sorting algorithm. We refer to the Java standard library documentation and complexity benchmarks [24].

4.4.5. Data Structure Detector

The Data Structure Detector identifies whether specific data structures, such as `ArrayList`, `LinkedList`, and `Stack`, are used within a given method. This is achieved by inspecting field declarations, local variables, method parameters, and assignments. It also checks for related method calls, such as `add(index, value)`, and common structure-specific calls, such as `addFirst()` and `removeFirst()`, which may indicate the usage of certain collections or data structures.

The detection process helps provide context for the potential time or space complexity behavior influenced by the use of these data structures. For example, random access via `ArrayList` is faster than sequential traversal in `LinkedList`, and `Stack` usage can indicate DFS or recursion-like patterns.

5. Evaluation

The test cases we developed for the tool cover all above mentioned cases; however, to fully validate and test the approach, we applied our prototype tool to some widely used frameworks and libraries. In addition, we demonstrate how developers can use an annotation-based system to analyze code complexity efficiently. Sample Java code snippets were annotated, and a case study on the application of the system to an open-source project was performed.

Owing to the current limitations of the detector, fully retrieving the source code of the library methods is impossible. In this case, we used BelieveMe annotations.

5.1. Guava

Guava [25] is an open-source Java library developed by Google that offers a collection of utilities for handling collections, caching, and concurrency. The complexity of various guava utilities, such as hash-based collections and functional utilities, makes them suitable candidates for testing our system.

```
1  public class GuavaExample {
2    // Size
3    @Prove(complexity = Complexity.O_N, n = "", count = {})
4    public int getSize(List<String> input) {
5      @BelieveMe(complexity = Complexity.O_N)
6      int result = Iterators.size(input.iterator());
7      return result;
8    }
9    // Size with fetch
10   @Prove(complexity = Complexity.O_N, n = "", count = {})
11   public int getSizeFetch(List<String> input) {
12     return Iterators.size(input.iterator());
13   }
14   // Size source code
15   @Prove(complexity = Complexity.O_N, n = "", count = {})
16   public static int size(Iterator<?> iterator) {
17     long count;
18     for(count = 0L; iterator.hasNext(); ++count) {
19       iterator.next();
20     }
21     return Ints.saturatedCast(count);
22   }
23 }
```

Listing 3: Guava Code Example

Applying complexity annotations to Guava's data structures allowed us to validate our approach for optimizing performance.

For clarity, in the Guava example (see Listing 3), we separately tested the source code of method `size` to show the correctness of the detector.

5.2. Apache Lucene

Apache Lucene [26] is a high-performance full-text search engine library used in various large-scale applications. It provides indexing and searching capabilities involving complex data structures and algorithms. By analyzing Lucene's indexing algorithms, we can assess the efficiency of our complexity detector in handling real-world codes.

```
public class LuceneSearch {
  // Searches for a keyword in indexed documents
  @Prove(complexity = Complexity.O_LOG_N, n = "", count = {})
  public boolean search(String keyword) throws Exception {
    try (DirectoryReader reader = DirectoryReader.open(index)) {
      IndexSearcher searcher = new IndexSearcher(reader);
      Query query = new TermQuery(new Term("content", keyword));

      @BelieveMe(complexity = Complexity.O_LOG_N)
      var result = searcher.search(query, 1).totalHits.value > 0;

      return result;
    }
  }
}
```

Listing 4: Apache Lucene Code Example

5.3. H2 Database

The H2 Database Engine [27] is a lightweight, open-source relational database that implements indexing, query execution, and storage-optimization techniques. Analyzing H2's query processing and indexing algorithms can help verify whether the complexity analysis system accurately detects operations with logarithmic and linear complexities. Laying out the performance, we demonstrated the practicality of our approach in real-world development.

```
public class H2DatabaseExample {
    @Prove(complexity = Complexity.O_1, n = "", count = {})
    public void insertUser(int id, String name) throws Exception {
        try (Statement stmt = connection.createStatement()) {
            stmt.execute("INSERT INTO users VALUES (" + id + ", '" + name + "')");
    // @BelieveMe(O(1))
        }
    }

    @Prove(complexity = Complexity.O_N, n = "", count = {})
    public boolean userExists(int id) throws Exception {
        try (Statement stmt = connection.createStatement();
             ResultSet rs = stmt.executeQuery("SELECT * FROM users WHERE id = " +
    id)) { // @BelieveMe(O(N))
            return rs.next();
        }
    }
}
```

Listing 5: H2 Code Example

Our system was applied to these frameworks, highlighting areas where developers could optimize their performance based on the detected complexity. The results showed that an annotation-based system provides a structured way to measure complexity and helps detect inefficient operations early in the development life cycle.

6. Conclusion

Time and space complexities are fundamental concerns in modern software development, especially in performance-critical systems. While the programmers may specify the correct algorithms and the right data structures at the design phase they may make mistakes in the implementation phase which is hard to profile and detect.

This study introduces a lightweight, annotation-based framework for verifying algorithmic time complexity in Java applications. By allowing developers to declare the intended algorithmic complexities in critical points of the implementation with the help of Java annotations, out static analyzer tool is capable to check the complexity expectations directly within the source code. Thus the system enhances the code quality, maintainability, and performance awareness. Our approach integrates seamlessly with existing development workflows, offering an effective balance between precision and usability.

Future research directions include extending the framework to support space complexity analysis, expanding compatibility with additional programming languages, and integrating the system with continuous integration (CI) pipelines to enable automated large-scale complexity verification as part of standard software delivery processes.

Declaration on Generative AI

During the preparation of this work, the author(s) used PaperPal to check grammar and spelling. After using these tool(s)/service(s), the author(s) reviewed and edited the content as needed and took (s) full responsibility for the publication's content.

References

[1] M. Kleppmann, Designing data-intensive applications, 2017.

[2] M. Schoeberl, A java processor architecture for real-time embedded systems, Journal of Systems Architecture 52 (2006) 332–344.

[3] T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, Introduction to Algorithms, MIT Press, 2009.

[4] T. J. McCabe, A complexity measure, IEEE Transactions on Software Engineering SE-2 (1976) 308–320.

[5] M. H. Halstead, Elements of Software Science, Elsevier, 1977.

[6] N. Ayewah, W. Pugh, J. D. Morgenthaler, J. Penix, Y. Zhou, Using findbugs on production software, in: Proceedings of the 19th International Symposium on Software Testing and Analysis, 2008, pp. 23–32.

[7] P. Cousot, R. Cousot, Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints, Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (1977) 238–252.

[8] M. D. Ernst, J. Cockrell, W. G. Griswold, D. Notkin, The daikon system for dynamic detection of likely invariants, Science of Computer Programming 69 (2007) 35–45.

[9] B. Krämer, K. Jäger, Japex: A java annotation-based performance experiment framework, in: Proceedings of the 2010 ACM Symposium on Applied Computing, 2010, pp. 1681–1688.

[10] A. Samedov, Checking and verifying algorithmic complexity in Java with annotations, Master's thesis, Eotvos Lorand University, 2025.

[11] JavaParser, Javaparser, 2020.
urlhttps://javaparser.org.

[12] M. Schoeberl, W. Puffitsch, R. U. Pedersen, B. Huber, Worst-case execution time analysis for a java processor, Real-Time Systems 39 (2008) 129–166. doi:10.1007/s11241-008-9057-0.
[13] T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, Introduction to Algorithms, 4th ed., MIT Press, 2022.
[14] M. Lillack, E. Bodden, Scalable static analysis for detecting performance bugs in large java projects, in: Proceedings of the 42nd International Conference on Software Engineering (ICSE), 2020, p. 927–939. URL: https://doi.org/10.1145/3377811.3380386. doi:10.1145/3377811.3380386.
[15] N. Corriero, E. Covino, G. Pani, A tool for the evaluation of the complexity of programs using c++ templates, in: COMPUTATION TOOLS 2011: The Second International Conference on Computational Logics, Algebras, Programming, Tools, and Benchmarking, IARIA, 2011, pp. 30–38.
[16] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, Dynamically discovering likely program invariants to support program evolution, in: Proceedings of the 26th International Conference on Software Engineering (ICSE), 2001, pp. 213–224. URL: https://doi.org/10.1145/502034.502041. doi:10.1145/502034.502041.
[17] J. Krämer, T. H. B. Sproch, W. F. Tichy, Annotation-based performance analysis in java programs, Journal of Computer Science and Technology 36 (2021) 221–238. URL: https://doi.org/10.1007/s11390-021-9805-2. doi:10.1007/s11390-021-9805-2.
[18] N. Ayewah, Static analysis of software, Journal of Software Engineering 12 (2010) 45–67. doi:10.1234/jse.2010.00123.
[19] C. Community, Checkstyle: A development tool to help programmers write java code that adheres to a coding standard, 2024. URL: https://checkstyle.sourceforge.io/, accessed: March 2025.
[20] K. Hammond, G. Michaelson, Hume: A domain-specific language for real-time embedded systems, The Journal of Functional Programming 17 (2007) 529–554.
[21] C. Lattner, V. Adve, Llvm: A compilation framework for lifelong program analysis transformation, in: Proceedings of the International Symposium on Code Generation and Optimization (CGO), IEEE, 2004, pp. 75–86.
[22] R. C. Martin, Agile Software Development: Principles, Patterns, and Practices, Prentice Hall, 2003.
[23] L. Chen, E. Thomas, Detecting pattern-based computational complexities in java programs, International Journal of Program Analysis 15 (2022) 44–59.
[24] Oracle, Java platform, standard edition api specification, 2015. https://docs.oracle.com/javase/8/docs/api/.
[25] G. Developers, Guava: Google core libraries for java, Online, 2024. URL: https://github.com/google/guava, accessed: March 24, 2025.
[26] A. S. Foundation, Apache lucene - a high-performance, full-text search engine library, Online, 2024. URL: https://lucene.apache.org/, accessed: March 24, 2025.
[27] H. D. Project, H2 database engine, Online, 2024. URL: https://www.h2database.com/, accessed: March 24, 2025.