Design and Implementation of a Static Analyzer for the Kotlin Programming Language

Mehmedalija Karišik, dr Dušan Vujošević, dr Nemanja Radosavljević

***Abstract* — This paper presents the design and implementation of "Rough Analyzer," a static analysis tool for the Kotlin programming language. Modern software engineering requires writing scalable and maintainable code, and our tool assists in this by detecting code complexity, stylistic issues, and logical errors that the compiler often misses. The analyzer that we propose leverages abstract syntax tree traversal to identify issues such as high cyclomatic complexity, long functions, and magic numbers. The architecture is designed to be modular and easily extensible, allowing teams to add custom rules. We demonstrate the tool's effectiveness on a real-world Kotlin project, showcasing its potential to improve code quality and reduce technical debt.**

***Key words* — Abstract Syntax Tree, Code Quality, Compiler Design, Gradle Plugin, Kotlin, Static Analysis.**

I. Introduction

In modern software development, code quality is not a luxury; it is a necessity for long-term project survival, especially in team settings. When code is unclear or unstructured, new developers struggle to contribute, and existing developers spend more time deciphering old logic than writing new features. This friction, often dismissed in the rush for quick delivery,

Mehmedalija Karišik, Author, UNION University School of Computing, 6/VI Kneza Mihaila, 11000 Belgrade, Serbia, and Technische Universität Wien, Karlsplatz 13, 1040 Vienna, Austria; (e-mail: karisik.mehmedalija@gmail.com).

Dušan Vujošević, Author, UNION University School of Computing, 6/VI Kneza Mihaila, 11000 Belgrade, Serbia; (e-mail: dvujosevic@raf.rs).

Nemanja Radosavljević, Author, UNION University School of Computing, 6/VI Kneza Mihaila, 11000 Belgrade, Serbia; (e-mail: nradosavljevic@raf.rs).

accumulates as technical debt. At its worst, technical debt can halt progress entirely, forcing a difficult choice: a costly, full-system rewrite or the slow decline of the project [8] [9].

Static analysis tools offer a direct, automated solution to combat this problem [4] [6]. While a compiler checks for syntactic correctness, a static analyzer goes deeper, identifying structural weaknesses, stylistic inconsistencies, and potential runtime errors that the compiler misses [3]. By providing developers with early and consistent feedback, these tools help enforce coding standards and prevent the accumulation of technical debt.

This paper presents "Rough Analyzer," our answer to the need for a simple, intuitive, and extensible static analysis tool for the Kotlin language [1]. While powerful tools like Detekt exist, our goal was to create a lightweight framework that is exceptionally easy to expand with custom rules tailored to a specific team or project. We built it as a Gradle plugin for seamless integration into the standard Kotlin development workflow. In the following sections, we detail the architecture of Rough Analyzer, demonstrate the implementation of several key analysis rules, and evaluate its effectiveness on a real-world project.

II. Background and Related Work

Static code analysis is a cornerstone of modern software development, providing automated code review by analyzing source code prior to execution. Unlike dynamic analysis, which observes program behavior at runtime, static analysis focuses on the code's structure and adherence to predefined rules to uncover potential bugs, security vulnerabilities, and stylistic issues [3]. Its real power is unlocked in Continuous Integration/Continuous Deployment (CI/CD) pipelines, where automated checks can prevent flawed code from being merged into the main codebase [10].

At the heart of our analyzer, and indeed any static analysis tool, lies the Abstract Syntax Tree (AST). An AST is a tree-like representation of the source code's structure, a concept thoroughly explored by authors like Nystrom [7]. During parsing, superficial syntactic details like parentheses are discarded, leaving only the essential logical components. This format allows an analysis tool to programmatically navigate elements such as function declarations and loops. For Rough Analyzer, we leverage the JetBrains 'kotlin-compiler-embeddable' library to parse Kotlin source files directly into a semantically-rich AST.

Once the AST is constructed, we need a clean way to perform operations on its nodes. For this, we employ the Visitor design pattern [5]. This pattern allows us to separate the logic for our analysis rules (the "operations") from the AST node classes themselves (the "structure"). By doing so, we can add new rules without modifying the core AST classes, adhering to the Open/Closed Principle. This design choice is fundamental to our goal of making Rough Analyzer easily extensible.

In the Kotlin ecosystem, the most prominent related tool is Detekt [2]. Detekt is a powerful, feature-rich static analyzer with a vast set of pre-built rules. However, its comprehensive architecture can introduce a steep learning curve for teams needing to write highly specific, domain-centric rules. Rough Analyzer is therefore positioned not as a replacement for Detekt, but as a lightweight and transparent alternative designed explicitly for simplicity. Our focus is to provide a clear framework where developers can quickly implement custom checks without needing to understand a complex internal architecture.

III. System Architecture

We designed Rough Analyzer with two primary goals: seamless integration into the Kotlin development workflow and maximum flexibility for the end-user. To achieve this, the architecture is divided into three distinct components: a Gradle plugin for build system integration, a YAML-based configuration engine for rule customization, and a core analysis mechanism that orchestrates the entire process. This modular design ensures that each part of the system has a single, clear responsibility.

A. Gradle Plugin Integration

To make Rough Analyzer a natural part of any Kotlin project, we implemented it as a Gradle plugin. This allows developers to execute the analysis with a standard Gradle command (‘./gradlew roughAnalyzer‘). The integration process consists of two parts: plugin definition and task registration. First, we define the plugin within its own ‘build.gradle.kts‘ file, assigning it a unique ID and specifying the main implementation class, as shown in Listing 1:

```
gradlePlugin {
    plugins {
        create("roughAnalyzerPlugin") {
            id = "rs.raf.student.rough-analyzer"
            implementationClass =
              "rs.raf.student.roughanalyzer.
    RoughAnalyzerPlugin"
        }
    }
}
```

Listing 1. Gradle plugin definition in ‘build.gradle.kts‘

The ‘RoughAnalyzerPlugin‘ class is the entry point. When applied to a project, it registers a new Gradle task named ‘roughAnalyzer‘ and creates a configuration extension, ‘roughAnalyzerConfig‘. This extension allows users to specify the source directory to be analyzed directly in their build script.

B. Configuration Engine

A static analyzer is only useful if its rules can be adapted to a project’s specific needs. A function with 20 lines might be acceptable in a UI-heavy Android project but unacceptable in a backend service. To address this, we built a flexible configuration engine based on a single YAML file named ‘rough-analyzer.yml‘, placed in the project’s root directory. This file allows developers to enable or disable specific rules and to set custom thresholds, as shown in Listing 2:

```
longFunction:
  active: true
  threshold: 80 # Increased for a Compose-heavy
    project

magicNumber:
  active: true
```

Listing 2. Example ‘rough-analyzer.yml‘ configuration file

Our ‘ConfigLoader‘ class is responsible for parsing this YAML file. If the file is not found, the loader provides a set of sensible default values, ensuring the tool works out-of-the-box. This provides a powerful yet simple way to fine-tune the analyzer’s behavior.

C. Core Analysis Mechanism

The core of the tool is the ‘AnalyzerEngine‘ class, which executes the analysis. Its function is to iterate over source files, apply a set of active rules, and return a list of found issues. The process begins in our custom Gradle task, ‘RoughAnalyzerTask‘. The task first identifies all ‘.kt‘ files. It then uses the ‘ConfigLoader‘ to load the configuration and instantiates a ‘RuleSetProvider‘. This provider acts as a factory, creating instances of only the active rules and injecting the correct threshold values.

Finally, the ‘AnalyzerEngine‘ receives the list of files and the active rule set. For each file, it performs the following steps:

1. It reads the file’s content into a string.
2. Using the ‘kotlin-compiler-embeddable‘ library, it parses the string into a ‘KtFile‘ object, the root of the AST.
3. It iterates through the active rules and calls the ‘apply(ktFile)‘ method on each one.

This design is highly modular. The ‘AnalyzerEngine‘ does not need to know the internal logic of any rule; it only interacts with them through a common ‘Rule‘ interface. This follows the Open/Closed Principle, allowing us to add new rules without ever modifying the engine’s code.

IV. Rule Implementation

The extensibility of Rough Analyzer is best demonstrated through the implementation of its rules. Each rule is a self-contained class that implements a common ‘Rule‘ interface, which requires an ‘id‘ and an ‘apply(KtFile)‘ method. This design allows us to add new checks to the system with minimal effort. Below, we detail the implementation of two fundamental rules: detecting long functions and identifying magic numbers.

A. Detecting Long Functions

Functions that are excessively long are a classic code smell. They are often difficult to understand, test, and maintain, as they tend to have multiple responsibilities.

The ‘LongFunctionRule‘ helps enforcing a reasonable function length, encouraging developers to break down complex logic into smaller, more focused units. The implementation of this rule uses a ‘KtTreeVisitorVoid‘ to traverse the AST of a given Kotlin file. The visitor overrides the ‘visitNamedFunction‘ method, which is automatically invoked for every

function declaration in the tree. Inside this method, we access the function's body, count the number of non-empty lines of code, and compare this count against the 'threshold' value loaded from the configuration. If the line count exceeds the threshold, a new 'Issue' is created. The core logic is shown in Listing 3:

```
// Inside LongFunctionRule.kt
psiFile.accept(object : KtTreeVisitorVoid() {
    override fun visitNamedFunction(function:
    KtNamedFunction) {
        super.visitNamedFunction(function)
        val body = function.bodyExpression ?: return
        val lineCount = body.text.lines()
                                 .count { it.
    isNotBlank() }

        if (lineCount > config.threshold) {
            issues.add(/* ... create Issue ... */)
        }
    }
})
```

Listing 3. Core logic of the LongFunctionRule

B. *Detecting Magic Numbers*

Magic numbers—hardcoded, unexplained numerical literals in the source code—harm readability and make maintenance difficult. If a value used in multiple places needs to change, a developer must find and update every instance, a process which is both tedious and error-prone. The 'MagicNumberRule' flags these values and encourages their extraction into named constants.

The implementation of this rule is similar to the implementation of the previous rule. This time however, we override the 'visitConstantExpression' method. Inside, we apply a series of checks to reduce false positives. We first check if the expression's text is a number and ignore common, self-explanatory values like 0, 1, and -1. Crucially, we also inspect the parent of the AST node. If the constant expression is part of a property declaration that is marked with the 'const' keyword, we ignore it, as it has already been properly defined as a constant. If a literal passes all these checks, it is flagged as a magic number.

V. Evaluation and Results

To validate the effectiveness and practical utility of Rough Analyzer, we ran it on a real-world, medium-sized Android project written in Kotlin. The project utilizes Jetpack Compose for its UI, which often leads to functions with a higher number of parameters and lines of code. The analysis was executed from the command line using the standard Gradle wrapper. The output from the tool, a truncated version of which is shown in Listing 4, immediately identified a large number of potential issues:

```
> ./gradlew roughAnalyzer
Analysis finished. Found 296 problems:

Function 'getBreeds' is too long (21 lines). Max is
    20.
[long-function]
at /BreedsRepository.kt:63:59

Magic number '10' found. Extract it to a named
    constant.
[magic-number]
at /BreedsRepository.kt:71:30
...
BUILD SUCCESSFUL
```

Listing 4. Truncated analysis results on the sample Android project

The results demonstrate that the analyzer correctly identifies rule violations. However, the initial run reported 296 issues, which could be overwhelming for a developer. This highlights a critical aspect of static analysis: “context is key”. The default threshold of 20 lines for a function is too strict for a Jetpack Compose UI component, which can easily be much longer.

This is precisely where Rough Analyzer’s configuration engine proves its value. By creating a ‘rough-analyzer.yml‘ file and increasing the ‘longFunction‘ threshold to a more reasonable value for UI code (e.g., 80 lines), we can filter out the noise and allow the team to focus on genuinely problematic areas of the codebase. This evaluation confirms that our tool is not only functional but that its flexible configuration is an essential feature for practical use.

VI. Conclusion and Future Work

In this paper, we have presented the design and implementation of Rough Analyzer, a static analysis tool for Kotlin built from the ground up with a focus on simplicity and extensibility. We have demonstrated how its architecture, based on a Gradle plugin, a flexible YAML configuration engine, and the Visitor design pattern, allows for seamless integration and customization. The primary contribution of this work is a lightweight framework that empowers development teams to quickly implement their own coding standards and rules without the steep learning curve associated with more complex, industrial-grade tools.

Rough Analyzer serves not only as a practical proof-of-concept but also as a valuable educational resource. Its transparent design provides a clear example of applying compiler theory concepts, such as AST traversal, to solve real-world software engineering problems. It fills a practical niche between comprehensive solutions like Detekt and the day-to-day needs of teams who require a simple, custom solution to enforce their specific coding conventions.

The modular architecture of Rough Analyzer provides a solid foundation for future enhancements. Immediate next steps involve expanding the set of built-in rules to detect a wider range of issues, including overly long lines, the use of unsafe casts, unreachable code, and unused imports. Beyond simple detection, we plan to introduce more sophisticated rules that can offer concrete suggestions for code refactoring. Furthermore, we envision creating pre-defined rule sets optimized for different project types, such as Android applications or backend services.

Perhaps the most significant direction for future work is deeper integration with CI/CD pipelines. By configuring the analyzer to automatically run on every commit or pull request, it can act as a quality gate, failing the build if critical issues are detected. This would fully automate the enforcement of code quality standards and ensure that no new technical debt is introduced, solidifying Rough Analyzer's role as an essential tool in a professional development workflow.

Literature

[1] *JetBrains*, "Kotlin programming language documentation," https://kotlinlang.org/docs/home.html, 2025, accessed: July 6, 2025.

[2] *The Detekt Team*, "Detekt documentation," https://detekt.dev/docs/intro/, 2025, accessed: July 6, 2025.

[3] V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, "Compilers: Principles, Techniques, and

Tools, 2nd ed". *Pearson/Addison Wesley*, 2007.
[4] W. Charoenwet, P. Thongtanunam, V. T. Pham, C. Treude. "An empirical study of static analysis tools for secure code review." *Proceedings of the 33rd ACM SIGSOFT international symposium on software testing and analysis*. 2024.
[5] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. "Design Patterns: Elements of Reusable Object-Oriented Software". Addison-Wesley Professional, 1995.
[6] Mweu, J. Ndia. "Static Analysis Techniques for Secure Software: A Systematic Review". *Journal of Cyber Security*. 2025.
[7] R. Nystrom, "Crafting Interpreters". *Genever Benning*, 2021.
[8] H. Saeeda, M. O. Ahamd, T. Gustavsson. "A Multivocal Literature Review on Non-Technical Debt in Software Development: An Insight into Process, Social, People, Organizational, and Culture Debt". *e-Informatica Software Engineering Journal*, 18(1), 240101. 2024.
[9] E. Scott, G. Robiolo, S. Matalonga, M. Felderer, D. Pfahl. "A study on reported under-documented non-functional requirements as an indicator of technical debt". *Software Quality Journal*, 33(3), 1-31. 2025.
[10] S. Yang. "The Impact of Continuous Integration and Continuous Delivery on Software Development Efficiency". *Journal of Computer, Signal, and System Research*, 2(3), 59-68. 2025.