УДК 004.42

Д. Р. Музичук
В. В. Войтко

ІНСТРУМЕНТИ СТАТИЧНОГО АНАЛІЗУ КОДУ ANDROID-ЗАСТОСУНКІВ

Вінницький національний технічний університет

Анотація

У роботі розглянуто сучасні підходи до статичного аналізу Android-застосунків. Проаналізовано можливості Android Lint, detekt та розширення компілятора мови Kotlin у виявленні помилок, архітектурних порушень і «запахів коду». Показано сильні та слабкі сторони існуючих інструментів, проведено їх порівняння.

Ключові слова: *статичний аналіз, Android, lint, detekt, аналіз коду.*

Abstract

The paper examines modern approaches to static analysis of Android applications. The capabilities of Android Lint, detekt, and the Kotlin compiler extension in detecting errors, architectural violations, and "code smells" are analyzed. The strengths and weaknesses of existing tools are shown and their comparison is conducted.

Keywords*: static analysis, Android, lint, detekt, code analysis.*

Вступ

Зростання складності розробки мобільних застосунків вимагає використання автоматизованих інструментів контролю якості програмного коду. В середовищі Android-розробки статичний аналіз виступає основним механізмом виявлення помилок, потенційних вразливостей і порушень архітектури ще до запуску програми. На відміну від динамічного тестування, статичний аналіз працює без запуску застосунку та дозволяє інтегрувати перевірки безпосередньо в процес збірки. Сучасні підходи до статичного аналізу Android-застосунків охоплюють кілька рівнів перевірок – від аналізу взаємодії з платформою та ресурсами до контролю якості коду й аналізу структури програми. Кожен із цих рівнів реалізується окремими інструментами та покриває різні аспекти якості програмної системи, що вимагає їх системного поєднання [1].

Інструменти статичного аналізу Android-коду

Android Lint [2] – стандартний інструмент статичного аналізу коду, інтегрований у Android Studio та систему збірки Gradle. Він виконує перевірки на рівні виклику API, ресурсів, продуктивності та безпеки. Lint підтримує створення користувацьких правил через Detector API, що дозволяє розробнику прописувати власні перевірки коду відповідно до потреб та вимог розроблюваного ним проєкту.

На рисунку 1 наведено приклад користувацької перевірки коду з використанням Lint та Detector API. Наведене правило відстежує виклики конкретного методу API, а саме Log.d, під час обходу синтаксичного дерева програми. Detector API підключається до процесу аналізу вихідного коду, перевіряє, чи належить знайдений виклик класу android.util.Log, і у разі збігу формує попередження через механізм Issue та context.report. Таким чином, створене правило автоматично сигналізує про небажане використання відлагоджувального логування, а Lint додає це правило до процесу перевірки проєкту.

```
class LogUsageDetector : Detector(), SourceCodeScanner {
    override fun getApplicableMethodNames(): List<String> {
        return listOf("d")
    }
    override fun visitMethodCall(
        context: JavaContext,
        node: UCallExpression,
        method: PsiMethod
    ) {
        val evaluator = context.evaluator
        if (evaluator.isMemberInClass(method, "android.util.Log")) {
            context.report(
                ISSUE,
                node,
                context.getLocation(node),
                "Avoid using Log.d() in production code"
            )
        }
    }
    companion object {
        val ISSUE: Issue = Issue.create(
            id = "LogUsage",
            briefDescription = "Debug logging in production",
            explanation = """
                Log.d() should not be used in release builds.
                Consider using a structured logging framework or removing debug logs.
            """.trimIndent(),
            category = Category.CORRECTNESS,
            priority = 6,
            severity = Severity.WARNING,
            implementation = Implementation(
                LogUsageDetector::class.java,
                Scope.JAVA_FILE_SCOPE
            )
        )
    }
}
```

Рисунок 1 – Реалізація функціоналу статичного аналізу коду за допомогою Lint

Detekt [3] – інструмент статичного аналізу для Kotlin, орієнтований на виявлення стилістичних порушень, складних структур коду і так званих «запахів коду» – симптомів поганого проєктування або структури коду, які не є прямими помилками, але свідчать про потенційні проблеми [4]. Він працює на основі аналізу синтаксичного дерева та підтримує створення власних правил, як і Lint.

Перевагою detekt є гнучкість конфігурації та орієнтація на підтримку чистої архітектури коду. При цьому він переважно аналізує локальні властивості програмних конструкцій і має обмежені засоби для глобального аналізу архітектури великих проєктів.

На рисунку 2 наведено приклад користувацької перевірки коду з використанням detekt.

```
class LongFunctionRule(config: Config) : Rule(config) {

    override val issue = Issue(
        javaClass.simpleName,
        Severity.CodeSmell,
        "Function is too long",
        Debt.TEN_MINS
    )

    override fun visitNamedFunction(function: KtNamedFunction) {
        super.visitNamedFunction(function)

        val lines = function.text.lines().size

        if (lines > 20) {
            report(
                CodeSmell(
                    issue,
                    Entity.from(function),
                    "Function has $lines lines (max 20)"
                )
            )
        }
    }
}


class CustomRuleSetProvider : RuleSetProvider {

    override val ruleSetId: String = "custom"

    override fun instance(config: Config): RuleSet =
        RuleSet(ruleSetId, listOf(LongFunctionRule(config)))
}
```

Рисунок 2 – Реалізація функціоналу статичного аналізу коду з використанням detekt

Detekt створює власне правило, яке реагує на оголошення функцій і оцінює їх розмір. Під час обходу PSI-дерева метод visitNamedFunction обчислює довжину функції. Якщо вона перевищує заданий поріг, формує запис про проблему через об’єкт CodeSmell. Опис правила зберігається в структурі Issue.

Ще одним напрямом статичного аналізу є використання компіляторних плагінів і систем метапрограмування, зокрема, Kotlin Symbol Processing (KSP) [5]. Такі інструменти дозволяють аналізувати структуру проєкту на етапі компіляції, генерувати код і виконувати додаткові перевірки.

Перевагою цього підходу є доступ до символьної моделі програми та можливість інтеграції аналізу безпосередньо в процес збірки. Однак, створення компіляторних розширень є складнішим завданням порівняно з використанням готових інструментів, що обмежує їх широке застосування.

На рисунку 3 наведено приклад користувацької перевірки коду з використанням KSP.

```
class AutoDtoProcessor(
    private val env: SymbolProcessorEnvironment
) : SymbolProcessor {

    private val logger = env.logger

    override fun process(resolver: Resolver): List<KSAnnotated> {
        val symbols = resolver
            .getSymbolsWithAnnotation("com.example.annotations.AutoDto")
            .filterIsInstance<KSClassDeclaration>()

        val deferred = mutableListOf<KSAnnotated>()

        for (cls in symbols) {
            if (!cls.validate()) {
                deferred += cls
                continue
            }

            val isDataClass = Modifier.DATA in cls.modifiers
            if (!isDataClass) {
                logger.error(
                    "@AutoDto можна ставити лише на data class: ${cls.qualifiedName?.asString()}",
                    cls
                )
            }
        }

        return deferred
    }
}

class AutoDtoProcessorProvider : SymbolProcessorProvider {
    override fun create(environment: SymbolProcessorEnvironment): SymbolProcessor {
        return AutoDtoProcessor(environment)
    }
}

@AutoDto
class UserDto(val id: Long)
```

Рисунок 3 – Реалізація функціоналу статичного аналізу коду за допомогою KSP

У наведеному прикладі оголошується користувацька анотація @AutoDto, після чого процесор під час компіляції знаходить усі класи, позначені цією анотацією, і аналізує їхню структуру. Для кожного такого класу перевіряється, чи оголошений він як data class, а якщо умова не виконується, то процесор генерує повідомлення про помилку і зупиняє компіляцію.

Порівняння підходів до статичного аналізу Android-коду

Попри спільну мету застосування, Android Lint, detekt і розширення компілятора мови Kotlin реалізують різні моделі аналізу. Android Lint орієнтований на аналіз особливостей використання платформи Android і тісно інтегрований з Android SDK. Він ефективний для перевірки використання API, ресурсів і помилок Android-екосистеми. Detekt перевіряє стиль, читабельність та структуру коду, виступаючи інструментом контролю якості на рівні мови програмування. Компіляторні розширення Kotlin забезпечують глибокий рівень аналізу, оскільки працюють із символьною моделлю програми під час компіляції, але потребують складнішої реалізації. Ці підходи не є взаємовиключними, оскільки кожен інструмент покриває окремий клас проблем.

Було сформовано таблицю 1 з порівнянням характеристик описаних інструментів статичного

аналізу коду у Android.

Таблиця 1 – Порівняння характеристик інструментів статичного аналізу у Android

Критерій	Android Lint	detekt	KSP
Аналіз Android API	+	–	+
Аналіз синтаксису	+	+	+
Виявлення «запахів коду»	+	+	+
Архітектурні перевірки	+	–	+
Аналіз ресурсів	+	–	–
Генерація коду	–	–	+
Глобальний аналіз проєкту	+	–	+
Розширюваність користувацькими правилами	+	+	+

Жоден із розглянутих інструментів не забезпечує універсального рішення для всіх задач статичного аналізу. Для проведення більш комплексного аналізу рекомендованим є комбіноване використання усіх цих трьох підходів, що дозволяє аналізувати одночасно як синтаксис програмної системи, так і її архітектуру.

Висновки

Було розглянуто основні інструменти статичного аналізу, що охоплюють різні рівні перевірки програмної системи – від контролю взаємодії з платформою Android до аналізу структури коду на етапі компіляції. Було продемонстровано можливості інструментів статичного аналізу коду у Android: Lint, detekt та розширення компілятора мови Kotlin. Продемонстровано їх можливості і забезпечення ними широкого спектру перевірок, що покривають різні аспекти аналізу коду та є взаємодоповнюваними інструментами.

СПИСОК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ

1. Sadowski C., Raghavan A. Software Engineering at Google: Lessons Learned from Programming Over Time. – Sebastopol : O'Reilly Media, 2021. – 602 p.
2. Android custom lint rules API guide [Електронний ресурс] / Google Samples. – Режим доступу: https://googlesamples.github.io/android-custom-lint-rules/api-guide.html
3. detekt – Static code analysis for Kotlin [Електронний ресурс] : офіційна документація. – Режим доступу: https://detekt.dev/docs/intro
4. Al-Obeidallah M. Gh., Al-Fraihat D. A systematic review on software code smells // International Journal of Electrical and Computer Engineering. – 2025. – Vol. 15, No. 3. – P. 3010–3027. – ISSN 2088-8708.
5. Kotlin Symbol Processing (KSP) Overview [Електронний ресурс] / Kotlin Documentation. – Режим доступу: https://kotlinlang.org/docs/ksp-overview.html

Музичук Дмитро Романович – аспірант кафедри програмного забезпечення, Вінницький національний технічний університет, м. Вінниця, e-mail: dmytromuzychuk2710@gmail.com.
Войтко Вікторія Володимирівна – кандидат технічних наук, доцент кафедри програмного забезпечення, Вінницький національний технічний університет, м. Вінниця, e-mail: dekanfki@i.ua.

Dmytro Muzychuk – Postgraduate student of Software Engineering department, Vinnytsia National Technical University, Vinnytsia, e-mail: dmytromuzychuk2710@gmail.com.
Viktoriia Voitko – Ph.D., Associate Professor of Software Engineering, Vinnytsia National Technical University,Vinnytsia, e-mail: dekanfki@i.ua.