https://doi.org/10.31891/2307-5732-2026-363-70
УДК 004.021.2:004.415

МУЗИЧУК ДМИТРО
Вінницький національний технічний університет
https://orcid.org/0009-0007-4104-0630
e-mail: dmytromuzychuk2710@gmail.com
ВОЙТКО ВІКТОРІЯ
Вінницький національний технічний університет
https://orcid.org/0000-0002-3329-7256
e-mail: dekanfki@i.ua

АНАЛІЗ ПІДХОДІВ ДО МЕТАПРОГРАМУВАННЯ ANDROID-ЗАСТОСУНКІВ

У статті проведено аналіз сучасних підходів до метапрограмування Android-застосунків з урахуванням специфіки мобільної платформи та вимог до масштабованості, продуктивності й супроводу програмного забезпечення. Розглянуто основні напрями застосування метапрограмування в Android, зокрема, генерацію коду на етапі компіляції, метапрограмування під час виконання, анотаційно-декларативні механізми, предметно-орієнтовані мови, архітектурне метапрограмування та автоматизацію збірки за допомогою Gradle. Показано, що зазначені підходи відрізняються етапом застосування, рівнем абстракції та характером впливу на програмну систему і не є альтернативними, а використовуються комплементарно в межах одного проєкту.

Виявлено ключові недоліки сучасних підходів до метапрограмування, пов'язані зі статичністю правил генерації, фрагментарністю застосування та відсутністю формалізованих моделей їх взаємодії. Обґрунтовано, що ці обмеження ускладнюють системний аналіз архітектури Android-застосунків і стримують подальшу автоматизацію процесу розробки. Показано, що метапрограмування надає формалізовану основу, придатну для інтеграції алгоритмів штучного інтелекту, які можуть бути використані для адаптивного вибору архітектурних рішень, оптимізації правил генерації коду та врахування зворотного зв'язку з експлуатації програмних систем. Отримані результати створюють передумови для розробки інтелектуалізованих методів і моделей проєктування Android-застосунків та визначають перспективні напрями подальших досліджень у цій галузі.

Ключові слова: *метапрограмування, Android, генерація коду, програмне забезпечення, автоматизація розробки, штучний інтелект.*

MUZYCHUK DMYTRO, VOITKO VIKTORIIA
Vinnytsia National Technical University

ANALYSIS OF APPROACHES TO ANDROID METAPROGRAMMING

The article presents a systematic analysis of modern approaches to metaprogramming in Android application development, considering the constraints of the mobile platform and the increasing requirements for scalability, performance, reliability, and maintainability of software systems. The study examines the principal domains of metaprogramming application, including compile-time metaprogramming, runtime metaprogramming, annotation-driven declarative mechanisms, domain-specific languages, architectural metaprogramming, and Gradle-based build automation. It is established that these approaches differ in their abstraction levels, stages of application, and impact on the software lifecycle. They should be viewed not as competing alternatives but as complementary components of a unified development methodology aimed at reducing boilerplate code, enforcing architectural consistency, and increasing development productivity.

Key shortcomings of modern approaches to metaprogramming are identified, related to the static nature of generation rules, fragmented application, and the lack of formalized models of their interaction. These factors complicate architectural transparency, restrict systematic analysis, and impede the automation of development processes. It is substantiated that these limitations complicate the systematic analysis of the architecture of Android applications and hinder further automation of the development process. It is shown that metaprogramming provides a formalized basis suitable for the integration of artificial intelligence algorithms, which can be used for adaptive selection of architectural solutions, optimization of code generation rules and consideration of feedback from the operation of software systems. The results obtained create the prerequisites for the development of intelligent methods and models for designing Android applications and identify promising areas for further research in this area.

Keywords: *metaprogramming, Android, code generation, software, development automation, artificial intelligence.*

Стаття надійшла до редакції / Received 18.02.2026
Прийнята до друку / Accepted 11.03.2026
Опубліковано / Published 26.03.2026

Постановка проблеми у загальному вигляді та її зв'язок із важливими науковими чи практичними завданнями

Сучасні Android-застосунки характеризуються високим рівнем складності, що зумовлено зростанням функціональних вимог, необхідністю підтримки різних апаратних конфігурацій, версій операційної системи та архітектурних підходів до побудови програмного забезпечення. У процесі розробки значна частина зусиль витрачається на реалізацію повторюваних структурних елементів, інтеграцію інфраструктурних компонентів та дотримання архітектурних обмежень, що негативно впливає на продуктивність розробки та якість програмного коду.

Існуючі підходи до розробки Android-застосунків переважно базуються на ручному програмуванні або частковій автоматизації, що не забезпечує системного вирішення проблеми масштабованості та супроводжуваності програмного забезпечення. Метапрограмування розглядається як перспективний напрям, здатний забезпечити формалізацію процесів створення програмних артефактів та зменшити залежність від людського фактору.

Проблема ускладнюється фрагментарністю застосування метапрограмних підходів в Android-екосистемі та відсутністю узагальненої методологічної моделі, що описує їх ролі, межі застосування та

взаємодію у межах єдиного процесу розробки. Незважаючи на активне використання окремих механізмів метапрограмування, їх поєднання здебільшого здійснюється інтуїтивно, без формалізованого опису архітектурних залежностей і контурів відповідальності. Це ускладнює системний аналіз таких розробок та обмежує можливості їх використання у поєднанні з алгоритмами штучного інтелекту, які потребують узгодженого і структурованого подання програмної системи для автоматизованого аналізу та синтезу коду.

Таким чином, дослідження підходів до метапрограмування Android-застосунків безпосередньо пов'язане з актуальними науковими та практичними завданнями програмної інженерії, зокрема, автоматизацією розробки програмного забезпечення, підвищенням його якості та створенням інтелектуалізованих інструментів підтримки життєвого циклу мобільних застосунків.

Аналіз досліджень та публікацій

Упродовж останніх років метапрограмування розглядається в наукових дослідженнях як сукупність методів компіляційного, інструментарного та інфраструктурного перетворення програмних систем, спрямованих на автоматизацію розробки та підвищення якості програмного забезпечення. Значна частина робіт зосереджена на модифікації програмного коду на етапі компіляції з використанням анотацій, обробки символів і плагінів компілятора.

Окремий напрям сучасних досліджень пов'язаний з аналізом структури програмного коду як основи для автоматизованих перетворень. У роботі [1] запропоновано інструмент для виділення залежностей у проєктах Kotlin, зокрема, у змішаних Kotlin–Java системах, що розглядається авторами як фундамент для статичного аналізу, генерації коду та підтримки архітектурних рішень. Подібні підходи є особливо актуальними для Android-застосунків, де метапрограмування тісно пов'язане з аналізом багатомодульної структури та взаємодією компонентів.

Значна кількість публікацій присвячена інструментації байткоду як формі метапрограмування. У роботах [2, 3] досліджено формалізовані механізми інструментації та їх вплив на продуктивність і стабільність програмних систем, зокрема, в контексті мобільних застосунків. Автори показують, що інструментація на рівні байткоду дозволяє провести моніторинг і аналіз без модифікації вихідного коду, проте потребує контролю накладних витрат і сумісності з інструментами оптимізації. Аналогічні висновки щодо практичного застосування інструментації в Android-середовищі наведено у роботі [4].

Окрему групу становлять емпіричні дослідження метарівня розробки Android-застосунків, зокрема, систем збірки і процесів безперервної інтеграції. У роботах [5, 6] проаналізовано еволюцію Gradle-конфігурацій і практики CI/CD у відкритих Android-проєктах, що підтверджує зростання складності метаконфігурацій та їх істотний вплив на супроводжуваність програмного забезпечення. Дослідження [7] і [8] демонструють, що скрипти систем збірки містять типові конфігурації, які можуть бути автоматично виявлені та частково виправлені, що відкриває можливості для подальшої інтелектуалізації процесів метапрограмування.

Отже, сучасні дослідження підтверджують ефективність окремих підходів метапрограмування, однак вказують на фрагментарність їх застосування та відсутність узагальнених моделей взаємодії. Це ускладнює системний аналіз архітектури Android-застосунків і стримує подальшу автоматизацію процесу розробки. Зазначені обставини обґрунтовують доцільність дослідження метапрограмування як основи для інтеграції алгоритмів штучного інтелекту.

Формулювання цілей статті

Метою роботи є аналіз та класифікація підходів до метапрограмування Android-застосунків, визначення їх переваг і обмежень, а також обґрунтування доцільності їх застосування як складової сучасних методів і моделей розробки мобільного програмного забезпечення.

Виклад основного матеріалу

Метапрограмування широко використовується в Android-розробці, що зумовлено поєднанням високої складності платформи та жорстких обмежень мобільного середовища. Android-застосунки характеризуються багатокомпонентною моделлю, необхідністю підтримки різних версій операційної системи, апаратних конфігурацій і архітектурних підходів, що призводить до написання значного обсягу повторюваного коду.

Метапрограмування дозволяє формалізувати типові шаблони взаємодії компонентів, автоматизувати їх реалізацію. Крім того, використання статичних механізмів метапрограмування зменшує залежність від рефлексії під час роботи застосунку, підвищує продуктивність, передбачуваність та спрощує супровід Android-застосунків у довготривалій перспективі.

Метапрограмування на етапі компіляції [9] – це підхід, який передбачає виконання логіки та генерацію коду під час компіляції всього проєкту. Для цього використовуються процесори анотації KSP (Kotlin Symbol processing) або Kapt (the Kotlin Annotation Processing Tool), що виконується як частина конвеєра збірки Gradle та дозволяє аналізувати структуру Kotlin-коду (класи, властивості, анотації), знаходити специфічні мітки і генерувати додаткові вихідні файли.

Основна перевага цього методу полягає у відсутності впливу на продуктивність застосунку під час його роботи, оскільки весь складний код генерується заздалегідь, тому процесору смартфона не потрібно витрачати ресурси на аналіз класів чи пошук методу через рефлексію, що є важливим для мобільних пристроїв, оскільки необхідним є швидкий запуск застосунку та економія заряду акумулятора, а також з урахуванням того, що більшість мобільних пристроїв не мають великих потужностей для обробки складних даних.

Проте цей підхід має деякі недоліки, зокрема, збільшення часу збірки проєкту. Кожен новий процесор

анотацій – це новий додатковий крок при компіляції, що сповільнює роботу розробника. Крім того, налагодження згенерованого коду є складним процесом, оскільки помилки виникають у файлах, які не редагуються вручну і приховані в глибинах папок збірки.

На рис. 1 показано мінімальний сценарій використання цього виду метапрограмування: клас, позначений анотацією @AutoFactory, автоматично отримує згенеровану фабрику, яка буде скомпільована разом із застосунком і не потребуватиме рефлексії під час роботи програми.

```
// @AutoFactory позначає клас, для якого треба згенерувати фабрику
@Target(AnnotationTarget.CLASS)
@Retention(AnnotationRetention.SOURCE)
annotation class AutoFactory

@AutoFactory
class AnalyticsTracker(private val context: android.content.Context) {
    fun track(event: String) { /*...*/ }
}

class AutoFactoryProcessor(
    private val codeGenerator: com.google.devtools.ksp.processing.CodeGenerator,
    private val logger: com.google.devtools.ksp.processing.KSPLogger
) : com.google.devtools.ksp.processing.SymbolProcessor {

    override fun process(resolver: com.google.devtools.ksp.processing.Resolver)
        : List<com.google.devtools.ksp.symbol.KSAnnotated> {

        val symbols = resolver.getSymbolsWithAnnotation(AutoFactory::class.qualifiedName!!)
        symbols.filterIsInstance<com.google.devtools.ksp.symbol.KSClassDeclaration>().forEach { cls ->
            val pkg = cls.packageName.asString()
            val name = cls.simpleName.asString()
            val file = codeGenerator.createNewFile(
                dependencies = com.google.devtools.ksp.processing.Dependencies(false, cls.containingFile!!),
                packageName = pkg,
                fileName = "${name}Factory"
            )
            file.writer().use { out ->
                out.appendLine("package $pkg")
                out.appendLine("class ${name}Factory(private val ctx: android.content.Context) {")
                out.appendLine("  fun create(): $name = $name(ctx)")
                out.appendLine("}")
            }
        }
        return emptyList()
    }
}
```

Рис. 1. Приклад метапрограмування під час компіляції в Android із використанням KSP для генерації інфраструктурного коду

У результаті роботи KSP-процесора під час збірки створюється новий файл, який потрапляє до каталогу build/generated/ksp/... і компілюється разом з основним кодом, що є важливим у Android, оскільки логіка створення допоміжних артефактів переноситься на етап компіляції, що покращує продуктивність старту застосунку і зменшує залежність від механізмів рефлексії, чутливих до оптимізації.

Метапрограмування під час виконання [10] використовує механізм рефлексії, який дозволяє програмі аналізувати власну структуру під час її виконання. Програма може отримувати інформацію про методи класу, отримувати доступ до приватних полів та створювати екземпляри об'єктів, типи яких не були відомі на етапі написання коду, що забезпечує гнучкість архітектури застосунку. Це дозволяє писати лаконічний код, який швидко адаптується до будь-яких змін.

Однак, рефлексія має недолік, оскільки перевірки відбуваються під час роботи застосунку – це споживає додаткову пам'ять та ресурси процесора смартфона, а також потребує значного часу роботи застосунку. Надмірне використання рефлексії може призводити до відчутних затримок при переході між екранами або під час ініціалізації модулів застосунку, тому цей підхід необхідно використовувати обережно. Також недоліком є безпека типів даних, оскільки дії виконуються в обхід компілятора. Помилки роботи застосунку можуть проявитися або під час тестування, або під час використання безпосередньо користувачем, що може призвести до неочікуваних збоїв. Ці недоліки є достатньо суттєвими для того, щоб розробники поступово відмовлялися від використання такого підходу до метапрограмування на користь метапрограмування на етапі компіляції.

На рис. 2 продемонстровано типову послідовність виконання при рефлексії у Android: завантаження класу за іменем, отримання конструктора і створення екземпляра.

```kotlin
object PluginLoader {

    fun create(pluginClassName: String): Any? {
        return try {
            val cls = Class.forName(pluginClassName) // або context.classLoader.loadClass(...)
            val ctor = cls.getDeclaredConstructor()
            ctor.isAccessible = true
            ctor.newInstance()
        } catch (e: Throwable) {
            android.util.Log.e("PluginLoader", "Reflection failed: $pluginClassName", e)
            null
        }
    }
}
```

Рис. 2. Приклад метапрограмування під час виконання в Android через механізм рефлексії

Цей код відображає специфіку метапрограмування під час виконання в Android-середовищі: після ввімкнення оптимізації та обфускації імена класів і членів можуть змінюватися, що може призвести до помилок ClassNotFoundException або NoSuchMethodException, тому у комерційних Android-застосунках рефлексія потребує явного опису правил збереження (-keep) або альтернативних механізмів реєстрації.

Анотаційно-декларативний підхід до метапрограмування [11] описує роботу програми, а не реалізацію процесу. Розробник використовує анотації, які надають метадані про класи, методи та змінні. Це дозволяє абстрагуватись від низькорівневої логіки, що робить код чистішим та легшим для читання.

На рис. 3 наведено приклад реалізації анотаційно-декларативного підходу до метапрограмування в Android-застосунках на основі бібліотеки Room. У цьому коді використані анотації @Dao, @Query, @Insert та @Database для декларативного опису операцій доступу до даних і конфігурації бази даних без явної реалізації інфраструктурної логіки.

```kotlin
@Dao
interface ProjectDao {
    @Insert(onConflict = OnConflictStrategy.REPLACE)
    suspend fun insertProject(project: ProjectEntity)
    @Query("SELECT * FROM projects ORDER BY createdAt DESC")
    fun getAllProjects(): Flow<List<ProjectEntity>>
    @Query("SELECT * FROM projects WHERE id = :projectId")
    suspend fun getProjectById(projectId: String): ProjectEntity?
    @Query("DELETE FROM projects WHERE id = :projectId")
    suspend fun deleteProject(projectId: String)
}

@Database(
    entities = [
        ProjectEntity::class,
        HardwareComponentEntity::class,
        FurnitureDesignEntity::class,
        HardwareSpecificationEntity::class
    ],
    version = 1
)
```

Рис. 3. Анотаційно-декларативний підхід до метапрограмування в Android на прикладі бібліотеки Room

Поданий фрагмент коду показує, що розробник визначає лише структуру сутностей, інтерфейси доступу до даних та SQL-запити у вигляді анотацій, тоді як фактична реалізація механізмів роботи з базою даних автоматично генерується під час збірки застосунку.

Цей підхід став стандартом для серіалізації даних у Android, де бібліотека бере на себе всю рутину по парсингу відповідей від сервера або з бази даних і позбавляє необхідності писати парсери вручну для кожного об'єкта. Такий підхід також дозволяє здійснювати валідацію даних, що створює код, де за зовнішнім виглядом класу відразу зрозуміло, які правила до нього застосовуються, як він взаємодіє з іншими складовими системи. Однак, цей підхід має недоліки, до яких належить те, що поведінка програми стає складною для розуміння, оскільки реальна логіка прихована за однією короткою назвою. Також, цей підхід має обмежену гнучкість, оскільки розробник обмежений тими параметрами, які передбачив автор анотації, а модифікації часто

неможливі. Якщо анотація налаштована неправильно, помилка може бути неочевидною, а повідомлення від компілятора надто загальним.

DSL у Android розробці [12] – це підхід до метапрограмування, який дозволяє створювати мікромову всередині мови програмування Kotlin для вирішення конкретних завдань завдяки особливостям мови Kotlin. Розробники можуть створювати структури, які виглядають як конфігураційні файли, але є повноцінним виконуваним кодом з перевіркою типів. Найвідомішою такою реалізацією є фреймворк Jetpack Compose, який набув широкого застосування на заміну XML-файлам для реалізації написання інтерфейсу у вигляді деревоподібної структури. Це є саме формою метапрограмування, оскільки ці структури не просто викликають функції, а будують дерево вузлів, яке реалізує інтерфейс на екрані. Така реалізація робить код менш схильним до помилок, оскільки компілятор вказує на доступні параметри всередині кожного блоку, обмежуючи область видимості функції. Головна перевага DSL полягає у високому рівні абстракції. Він дозволяє реалізувати бізнес-логіку у вигляді, схожому на мову людини, що пришвидшує та полегшує підтримку коду, оскільки складні механізми приховані за зрозумілим синтаксисом, адаптованим під предметну область.

До недоліків DSL можна віднести складність проєктування, оскільки складним є процес розробки зручного та зрозумілого DSL. Також занадто глибока вкладеність функції може створювати навантаження на стек під час виконання програми. Складною також є послідовність виконання команд, оскільки DSL приховує ітеративну природу коду.

На рис. 4 наведено приклад типового DSL-опису компонента інтерфейсу через @Composable функцію бібліотеки Jetpack Compose.

```
@Composable
fun ProfileCard(
    name: String,
    onLogout: () -> Unit
) {
    Column(modifier = Modifier.padding(16.dp)) {
        Text(text = "Профіль", style = MaterialTheme.typography.titleLarge)
        Spacer(Modifier.height(8.dp))
        Text(text = name, style = MaterialTheme.typography.bodyLarge)
        Spacer(Modifier.height(12.dp))
        Button(onClick = onLogout) { Text( text: "Вийти") }
    }
}
```

Рис. 4. DSL-підхід до метапрограмування в Android на прикладі Jetpack Compose

Специфіка DSL-підходу в Android полягає у тісному зв'язку з життєвим циклом компонента, станом (State) та подіями вводу, а також у можливості інтеграції з системними сервісами. Таким чином, DSL забезпечує високорівневий опис поведінки й структури інтерфейсу користувача та виступає формою метапрограмування, де код описує модель, що надалі інтерпретується механізмами платформи.

Архітектурне метапрограмування [13] передбачає створення системних правил та шаблонів, які керують життєвим циклом та взаємодією компонентів застосунку. Це передбачає розробку власного каркасу, де код працює за визначеною металогікою. Прикладом є реалізація сучасних архітектурних патернів, таких як MVI (model-view-intent). У таких системах розробник описує лише наміри та стани, а архітектура сама піклується про те, як ці дані будуть передаватися через потоки, як вони будуть зберігатися та як потраплять до інтерфейсу користувача. Цей підхід часто використовує "шаблонні методи" або делегування властивостей Kotlin. Наприклад, розробник може вказати намір отримати об'єкт, а внутрішня логіка Android сама вирішує, де його взяти, як створити та коли знищити, коли він уже більше не потрібний. Мета такого підходу – стандартизація розробки у великих командах розробників, коли архітектура сама визначає правила через метаконструкції, і розробникам складніше помилитися або написати код, який порушує загальні правила проєкту. Це створює фундамент для масштабування застосунків, де екрани працюють за єдиною передбаченою логікою.

До недоліків цього підходу можна віднести надлишкову складність, оскільки для малих проєктів такий підхід є занадто важким та непотрібним. Також розробникам важко реалізувати нестандартний функціонал, якщо він не вкладається в загальну концепцію архітектури застосунку. Високою тут є вартість помилки в базовому шарі архітектури, яка може призвести до нестабільної роботи всього застосунку.

На рис. 5 наведено типову MVI-структуру, де бізнес-логіка формалізується через контракти State/Event/Effect і функцію редукції.

Метапрограмування через Gradle – це рівень метапрограмування, який охоплює площину інструментарію збірки, а не написання коду.

```
sealed interface LoginEvent {
    data class EmailChanged(val value: String) : LoginEvent
    data class PasswordChanged(val value: String) : LoginEvent
    data object Submit : LoginEvent
}

data class LoginState(
    val email: String = "",
    val password: String = "",
    val loading: Boolean = false,
    val error: String? = null
)

sealed interface LoginEffect {
    data object NavigateHome : LoginEffect
    data class ShowToast(val message: String) : LoginEffect
}

fun reduce(state: LoginState, event: LoginEvent): LoginState = when (event) {
    is LoginEvent.EmailChanged -> state.copy(email = event.value, error = null)
    is LoginEvent.PasswordChanged -> state.copy(password = event.value, error = null)
    LoginEvent.Submit -> state.copy(loading = true, error = null)
}
```

Рис. 5. Архітектурне метапрограмування

Gradle [5] – це система для збору проєктів. Вона дозволяє писати скрипти, які модифікують додаток у процесі його створення. Тут відбувається маніпулювання не об'єктами, а самими файлами, ресурсами та байткодом застосунку, завдяки чому розробники можуть автоматизувати генерацію конфігурації для різних версій продукту, що дозволяє мати одну кодову базу для десятків різних варіацій застосунку. Більш просунуті техніки застосування включають маніпуляцію байткодом, за допомогою чого можна автоматично додавати код у скомпільовані класи [4]. Це використовується переважно для автоматичного вимірювання швидкості роботи методів або збору аналітики без необхідності додавати функціонал для збору в кожну функцію. Gradle дозволяє також інтегрувати зовнішні системи метаданих безпосередньо в код, що робить процес гнучким, дозволяє змінювати поведінку додатка на етапі інтеграції, не відкриваючи середовище розробки.

До недоліків такого підходу можна віднести складність скриптів, непередбачуваність збірки, що може призвести до того, що проєкт перестане збиратися, а також приховані зміни, коли код змінюється на етапі байт-коду, розробник може бачити в середовищі розробки одне, а під час виконання відбуватиметься інші процеси.

На рис. 6 наведено узагальнений приклад метапрограмування через Gradle на рівні Android-модуля. Показано, як у межах build.gradle.kts задаються правила збірки, підключаються інструменти генерації та конфігуруються параметри, що впливають на фінальні артефакти (APK/AAB) без модифікації бізнес-коду.

Наведений фрагмент демонструє, що Gradle-метапрограмування в Android реалізується через декларативну та програмовану конфігурацію збірки: визначення варіантів продукту, ін’єкцію значень у BuildConfig, параметризацію ресурсів і маніфесту, а також підключення плагінів, що виконують генерацію коду або трансформації під час збірки. Таким чином, одна кодова база може підтримувати різні середовища виконання та поведінкові режими, а логіка “налаштування застосунку” переноситься в шар інструментарію.

Розглянуті підходи до метапрограмування Android-застосунків відрізняються за етапом застосування, рівнем абстракції та характером впливу на програмну систему, що зумовлює їх використання для вирішення різних класів завдань. Метапрограмування на етапі компіляції орієнтоване на генерацію інфраструктурного коду та перенесення обчислювальної складності на етап збірки, тоді як метапрограмування під час виконання забезпечує адаптивність поведінки застосунку під час виконання. Анотаційно-декларативний підхід та DSL виконують роль формалізованих засобів опису намірів розробника та правил предметної області, а архітектурне метапрограмування і метапрограмування з використанням Gradle спрямовані на стандартизацію структури проєктів і контроль якості в масштабних системах.

Таким чином, зазначені підходи не є альтернативними або взаємовиключними, а виступають комплементарними механізмами, що застосовуються паралельно на різних рівнях процесу розробки Android-застосунків. У проєктах вони поєднуються, утворюючи багаторівневу систему, у якій декларативні описи, генерація коду та процеси збірки взаємодіють між собою.

Незважаючи на активне використання метапрограмування в Android-розробці, сучасні підходи мають низку спільних недоліків, що не дозволяють повною мірою реалізувати їх потенціал. Переважна більшість підходів до метапрограмування базується на статичних правилах і шаблонах, які формалізують типові рішення, але не враховують контекст конкретного застосунку, його масштаб, характер навантаження. У результаті метапрограмування часто зводиться до автоматизації повторюваних фрагментів коду без можливості адаптації до нетипових сценаріїв.

```
plugins {
    id("com.android.application")
    kotlin("android")
    id("com.google.devtools.ksp")
}

android {
    namespace = "com.example.app"
    compileSdk = 35

    defaultConfig {
        applicationId = "com.example.app"
        minSdk = 24
        targetSdk = 34

        val props = Properties()
        val file = rootProject.file("local.properties")
        if (file.exists()) props.load(file.inputStream())

        buildConfigField(
            type: "String",
            name: "API_KEY",
            value: "\"${props.getProperty("api.key", "")}\""
        )
    }

    buildTypes {
        debug {
            buildConfigField( type: "Boolean",  name: "LOG_ENABLED",  value: "true")
        }
        release {
            isMinifyEnabled = true
            buildConfigField( type: "Boolean",  name: "LOG_ENABLED",  value: "false")
        }
    }

    buildFeatures {
        buildConfig = true
    }
}
```

Рис. 6. Приклад метапрограмування через Gradle в Android-модулі (Kotlin DSL)

Іншим суттєвим недоліком є фрагментарність застосування підходів і відсутність узгодженої моделі їх взаємодії. Різні механізми впроваджуються ізольовано, що ускладнює аналіз архітектурних залежностей і супровід системи в довготривалій перспективі. Крім того, більшість підходів не передбачає використання зворотного зв'язку з експлуатації застосунку, внаслідок чого згенерований код залишається незмінним незалежно від реальної поведінки системи.

Зазначені недоліки вказують на необхідність переходу від статичної автоматизації до більш адаптивних й інтелектуалізованих методів підтримки процесу розробки.

Метапрограмування надає формалізовану основу, придатну для застосування алгоритмів штучного інтелекту в процесі автоматизованої розробки Android-застосунків. Декларативні описи, метадані та структурні моделі перетворюють програмну систему на об'єкт, придатний для машинного аналізу. На відміну від традиційного програмного коду такі подання мають чітко визначену структуру та явні залежності, що можуть бути використані як база для побудови інтелектуальних методів аналізу й синтезу.

Алгоритми штучного інтелекту доцільно розглядати не як заміну метапрограмування, а як засіб подолання його недоліків. Використання методів машинного навчання та аналізу даних відкриває можливості для адаптивного вибору архітектурних рішень, оптимізації правил генерації коду та врахування зворотного зв'язку з експлуатації застосунків. Таким чином, поєднання метапрограмування й алгоритмів штучного інтелекту формує передумови для переходу від статичної автоматизації до інтелектуалізованих методів проєктування Android-застосунків.

Висновки

Проведено аналіз сучасних підходів до метапрограмування Android-застосунків, що застосовуються на різних етапах і рівнях процесу розробки програмного забезпечення.

Встановлено, що існуючі підходи до метапрограмування не є альтернативними, а мають комплементарний характер і зазвичай використовуються спільно в межах одного проєкту. Водночас їх застосування здебільшого носить фрагментарний характер і не спирається на узагальнену методологічну модель, що ускладнює аналіз архітектури та обмежує можливості масштабування і супроводу Android-застосунків.

Метапрограмування створює формалізовану основу для інтеграції алгоритмів штучного інтелекту в процес розробки Android-застосунків. Декларативні описи, метадані та структурні моделі, сформовані в межах метапрограмування, є придатними для автоматизованого аналізу та синтезу програмних рішень. Поєднання метапрограмування з алгоритмами штучного інтелекту розглядається як перспективний напрям подальших досліджень, спрямований на створення адаптивних і інтелектуалізованих методів проєктування Android-застосунків.

Література

1. Feng Q., Ma X., Ji H., Song W., Liang P. Depends-Kotlin: A Cross-Language Kotlin Dependency Extractor // Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering (ASE 2024). — New York : ACM, 2024. — P. 1–5.

2. Soueidi C., Monnier M., Falcone Y. Efficient and Expressive Bytecode-Level Instrumentation for Java Programs // Software Tools for Technology Transfer. — 2023. — Vol. 25. — No. 3. — P. 487–507.

3. Auer M., Arcuschin Moreno I., Fraser G. WallMauer: Robust Code Coverage Instrumentation for Android Apps // Proceedings of the 17th International Conference on Software Testing, Verification and Validation (ICST 2024). — IEEE, 2024. — P. 1–10.

4. Samhi J., Zeller A. AndroLog: Android Instrumentation and Code Coverage Analysis. – arXiv:2404.11223, 2024. – DOI: 10.48550/arXiv.2404.11223.

5. Liu P., Li L., Liu K., McIntosh S., Grundy J. Understanding the quality and evolution of Android app build systems // Journal of Software: Evolution and Process. — 2024. — Vol. 36, No. 5. — Article e2602. — DOI: 10.1002/smr.2602.

6. Ghaleb T. A., Abduljalil O., Hassan S. CI/CD Configuration Practices in Open-Source Android Apps: An Empirical Study // ACM Transactions on Software Engineering and Methodology. — 2024.

7. Zhang C., Chen B., Hu J., Peng X., Zhao W. BuildSonic: Detecting and Repairing Performance-Related Configuration Smells for Continuous Integration Builds // Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering (ASE ’22). — Rochester, MI, USA, 2022. — Pp. 1–13. — DOI: 10.1145/3551349.3556923.

8. Tamanna M., Chandrani Y., Burrows M., Wroblewski B., Williams L., Wermke D. Your Build Scripts Stink: The State of Code Smells in Build Scripts — arXiv preprint arXiv:2506.17948 [cs.SE], 2025. — DOI: 10.48550/arXiv.2506.17948.

9. Tronetti E. Towards Aggregate Programming in Pure Kotlin through Compiler-Level Metaprogramming : магістерська дисертація / Università di Bologna. — Bologna, 2022. — 62 с. — Режим доступу: https://amslaurea.unibo.it/id/eprint/28077/1/Elisa-Tronetti-Master-Thesis-v2.1.0.pdf (дата звернення: 29.12. 2025).

10. Sun X., Li L., Li T., Bissyandé T. F., Klein J. Taming Reflection: An Essential Step Toward Whole-Program Analysis of Android Apps. Proceedings of the ACM on Programming Languages (OOPSLA). 2021. DOI: 10.1145/3440033.

11. Pfeffer D., Weninger M. On the Applicability of Annotation-Based Source Code Modification in Kotlin (Work in Progress) // Proceedings of the MPLR Workshop @ SPLASH 2023. – 2023. – P. 1–6.

12. Smyth N. Jetpack Compose 1.7 Essentials. Raleigh: The Pragmatic Bookshelf, 2024.

13. Sanchez D., Rojas A. E., Florez H. Towards a Clean Architecture for Android Apps using Model Transformations. IAENG International Journal of Computer Science. 2022. Vol. 49, No. 1. P. 270–278.

References

1. Feng Q., Ma X., Ji H., Song W., Liang P. Depends-Kotlin: A Cross-Language Kotlin Dependency Extractor // Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering (ASE 2024). — New York : ACM, 2024. — P. 1–5.

2. Soueidi C., Monnier M., Falcone Y. Efficient and Expressive Bytecode-Level Instrumentation for Java Programs // Software Tools for Technology Transfer. — 2023. — Vol. 25. — No. 3. — P. 487–507.

3. Auer M., Arcuschin Moreno I., Fraser G. WallMauer: Robust Code Coverage Instrumentation for Android Apps // Proceedings of the 17th International Conference on Software Testing, Verification and Validation (ICST 2024). — IEEE, 2024. — P. 1–10.

4. Samhi J., Zeller A. AndroLog: Android Instrumentation and Code Coverage Analysis. – arXiv:2404.11223, 2024. – DOI: 10.48550/arXiv.2404.11223.

5. Liu P., Li L., Liu K., McIntosh S., Grundy J. Understanding the quality and evolution of Android app build systems // Journal of Software: Evolution and Process. — 2024. — Vol. 36, No. 5. — Article e2602. — DOI: 10.1002/smr.2602.

6. Ghaleb T. A., Abduljalil O., Hassan S. CI/CD Configuration Practices in Open-Source Android Apps: An Empirical Study // ACM Transactions on Software Engineering and Methodology. — 2024.

7. Zhang C., Chen B., Hu J., Peng X., Zhao W. BuildSonic: Detecting and Repairing Performance-Related Configuration Smells for Continuous Integration Builds // Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering (ASE ’22). — Rochester, MI, USA, 2022. — Pp. 1–13. — DOI: 10.1145/3551349.3556923.

8. Tamanna M., Chandrani Y., Burrows M., Wroblewski B., Williams L., Wermke D. Your Build Scripts Stink: The State of Code Smells in Build Scripts — arXiv preprint arXiv:2506.17948 [cs.SE], 2025. — DOI: 10.48550/arXiv.2506.17948.

9. Tronetti E. Towards Aggregate Programming in Pure Kotlin through Compiler-Level Metaprogramming : магістерська дисертація / Università di Bologna. — Bologna, 2022. — 62 с. — Режим доступу: https://amslaurea.unibo.it/id/eprint/28077/1/Elisa-Tronetti-Master-Thesis-v2.1.0.pdf

10. Sun X., Li L., Li T., Bissyandé T. F., Klein J. Taming Reflection: An Essential Step Toward Whole-Program Analysis of Android Apps. Proceedings of the ACM on Programming Languages (OOPSLA). 2021. DOI: 10.1145/3440033.

11. Pfeffer D., Weninger M. On the Applicability of Annotation-Based Source Code Modification in Kotlin (Work in Progress) // Proceedings of the MPLR Workshop @ SPLASH 2023. – 2023. – P. 1–6.

12. Smyth N. Jetpack Compose 1.7 Essentials. Raleigh: The Pragmatic Bookshelf, 2024.

13. Sanchez D., Rojas A. E., Florez H. Towards a Clean Architecture for Android Apps using Model Transformations. IAENG International Journal of Computer Science. 2022. Vol. 49, No. 1. P. 270–278.