Міністерство освіти і науки України

Національний університет «Києво-Могилянська академія»

Факультет інформатики

Кафедра інформатики

Кваліфікаційна робота

освітній ступінь – бакалавр

на тему: «**Вдосконалення інструменту автоматичної оцінки стабільності та зрозумілості програмного коду Swift з інтеграцією в середовище розробки»**

Текстова частина до кваліфікаційної роботи

за спеціальністю «Інженерія програмного забезпечення» - 121

Виконав: студент 4-го року навчання,

Спеціальності
121 «Інженерія Програмного Забезпечення»

Студента Суліменко Андрія Андрійовича

Керівник Франків О.О.

магістр комп’ютерних наук, асистент

5 травня 2025 р.

Київ – 2025

Національний університет «Києво-Могилянська академія»

Факультет інформатики

Кафедра інформатики

Освітній ступінь бакалавр

Спеціальність 121 «Інженерія Програмного Забезпечення»

Освітня програма бакалавр

ЗАТВЕРДЖУЮ

Завідувач кафедри інформатики

Гороховський С. С.

8 жовтня 2024 року

ЗАВДАННЯ

ДЛЯ КВАЛІФІКАЦІЙНОЇ РОБОТИ СТУДЕНТУ

Суліменко Андрію Андрійовичу

1. Тема роботи «**Вдосконалення інструменту автоматичної оцінки стабільності та зрозумілості програмного коду Swift з інтеграцією в середовище розробки»**, керівник роботи Франків Олександр Олександрович, магістр комп’ютерних наук, асистент

2. Строк подання роботи студентом 7 травня 2025

3. План роботи

Зміст

Висновки

Список літератури

Додатки (за необхідністю)

ГРАФІК ПІДГОТОВКИ КВАЛІФІКАЦІЙНОЇ РОБОТИ ДО ЗАХИСТУ

No п/п	Назва етапу кваліфікаційного проекту (роботи)	Термін виконання етапу	Примітка
1.	Отримання завдання на кваліфікаційну роботу.	03.10.2024	
2.	Огляд наукової літератури за темою дослідження.	11.11.2024	
3.	Огляд технічної літератури за темою дослідження.	26.11.2024	
4.	Створення плану роботи над основною частиною кваліфікаційної роботи.	03.12.2024	
5.	Створення зразку оптимізованої архітектури основного програмного модулю стабілізатора.	10.01.2025	
6.	Розробка оптимізованого алгоритму збору інформації про та аналізу програмного забезпечення.	04.01.2025	
7.	Розширення функціональних можливостей розробленого аналізатору коду.	10.01.2025	
8.	Імплементація алгоритму поширення пакунку Swift.	18.01.2025	
9.	Інтеграція розробленого іструменту з середовищем розробки Xcode.	02.03.2025	
10.	Оптимізація конфігурованих метричних обчислень.	16.03.2025	
11.	Написання пояснювальної роботи.	24.03.2025	
12.	Створення слайдів для доповіді та написання доповіді.	18.04.2025	
13.	Захист кваліфікаційної роботи (проекту).	21.05.2025	

Графік узгоджено 8 жовтня 2024 р.

Науковий керівник Франків Олександ Олександрович

Виконавець курсової роботи Суліменко Андрій Андрійович

Зміст

Анотація

У запропонованій дипломній роботі розглянуто розширення створеного середовища розробки для автоматичного оцінювання стабільності та зрозумілості програмного коду мовою Swift. Деталізовано аналіз та моделювання методів оцінки програмного коду та автоматичного аналізу програмного забезпечення. Значну увагу приділено опису моделі середовища автоматичного аналізу, його розробці та оптимізації.

У рамках цієї роботи розроблений інструмент для керування пакунками Swift було розширено з метою автоматизації та оптимізації процесу аналізу архітектури програмного модуля за допомогою метрик оцінки стабільності програмного забезпечення.

Вступ

У сучасному програмному середовищі якість програмного забезпечення є ключовим фактором, що визначає його довговічність, підтримуваність та ефективність. Одним з найважливіших аспектів забезпечення високої якості існуючих інструментів аналізу стабільності та зрозумілості коду є їхня автоматизованість та продуктивність, що дозволяє застосовувати запропоновані методи оцінки для виявлення потенційних проблем на ранніх етапах розробки з мінімальними зусиллями та технічним досвідом потенційного користувача.

Попереднє дослідження у сфері автоматизації методів оцінки стабільності та зрозумілості програмного коду та їхнього впливу на загальну якість програмного забезпечення заклали основу для подальшого вдосконалення підходів до аналізу та оптимізації розробленого інструменту. Раніше було розроблено інструмент оцінки стабільності та зрозумілості програмного забезпечення на мові програмування Swift, що базується на метриках об’єктно-орієнтованого програмування. Однак, для ефективного застосування цього інструменту в реальних проектах необхідною складовою була його інтеграція в середовище розробки та автоматизація розробленого процесу аналізу.

Ця робота присвячена суттєвому розширенню та вдосконаленню запропонованого аналізатора, що охоплює як покращення його основного аналітичного функціоналу для більш точної оцінки стабільності та зрозумілості Swift-коду, так і його автоматизацію та інтеграцію в CI/CD-процеси. Зокрема, було доопрацьовано ядро аналізу, включаючи оптимізацію процесу парсингу коду з використанням SwiftSyntax [15], впровадження паралельних обчислень для розрахунку метрик за допомогою Swift Concurrency (Task) [16] та перехід на більш гнучку протокольно-орієнтовану архітектуру для визначення та обробки метрик. Поряд із цими внутрішніми вдосконаленнями, значну увагу приділено розробці механізмів інтеграції в середовище розробки Xcode через Build Phases Scripts, гнучкому налаштуванню параметрів аналізу за допомогою YAML [17]

конфігурацій, розширенню можливостей візуалізації результатів безпосередньо у середовищі розробки, а також реалізації підтримки різних форматів виводу для адаптації до потреб розробників та команд.

Структура роботи побудована відповідно до етапів розширення та автоматизації аналізатора. У першому розділі розглянуто сучасні методи оцінки стабільності та зрозумілості програмного коду, обґрунтовано вибір метрик та їх комбінацій для створення комплексної оцінки якості. Другий розділ присвячено інтеграції аналізатора в середовище розробки та автоматизації процесу аналізу. У третьому розділі описано реалізацію та оптимізацію аналізатора, а також можливості його подальшого вдосконалення.

Результатом роботи є розширений інструмент автоматизованого аналізу, який забезпечує зручність та ефективність оцінки програмного коду у процесі розробки. Ця робота відкриває нові можливості для розробників Swift-проектів, покращуючи контроль якості коду та сприяючи створенню більш стабільних та зрозумілих програмних рішень.

Розділ 1. Аналіз методів оцінки стабільності та зрозумілості програмного коду

1.1 Загальний опис

Аналіз стабільності та зрозумілості програмного коду наразі перетворився на окрему складову розробки програмного забезпечення, що лише підтверджує динамічність розвитку цієї галузі.

Загалом, аналіз програмного забезпечення можна розділити на статичний та динамічний. Статичний аналіз програмного коду передбачає аналіз досліджуваних метрик програми без її виконання. Глибина такого аналізу цілком варіюється від застосовуваних інструментів. У свою чергу, динамічний аналіз програмного коду – це аналіз програмного забезпечення, що здійснюється за рахунок виконання програми на реальному чи віртуальному процесорі з використанням описаних умов роботи прикладної програми. Відповідно до класифікації, статичний аналіз має значно ширшу сферу використання, оскільки отримання інформації про досліджуванні метрики програмного забезпечення може бути легко поєднано з цим методом аналізу програмного забезпечення, особливо при побудові комплексних вбудованих систем. Іншою перевагою статичного методу аналізу є можливість застосування без виконання програми. Цю особливість статичного аналізу часто використовують для Static Application Security Testing (SAST) [27] – глибинного методу тестування програмного забезпечення, що зазвичай використовується в комп'ютерних системах високої надійності для пошуку потенційно вразливого коду.

Спираючись на результати попереднього дослідження у галузі автоматизації оцінки стабільності та зрозумілості програмного коду, ідея щодо поєднання здобутих результатів та статичного аналізу, що може бути здійснено автоматично у середовищі розробки Xcode, найбільш поширеного середовища розробки програмного забезпечення мовою програмування Swift, є основною складовою для створення потенційно новаторського інструменту, що надав би можливість Swift-розробникам здійснювати якісну перевірку розробленого

програмного коду на відповідність інтернаціонального стандарту визначення якості програмного продукту програмного продукту. Оскільки, в основу попередньо розробленого інструменту оцінки якості програмного забезпечення було закладено модель ISO/IEC 9126 [1], інструмент дозволяє врахувати практичність, ефективність, надійність, портативність та обслуговування, як важливі критерії стабільності програмного коду.

В даній роботі основну увагу зосереджено саме на налагодженні процесу автоматизації попередньо розробленого інструменту управління пакунками Swift з можливістю гнучкого налаштування всередині середовища розробки програмного забезпечення Xcode. Це середовище розробки програмного забезпечення, що вперше було презентовано компанією Apple у жовтні 2003 року, наразі є інтегрованим середовищем розробки для всіх підтримуваних операційних систем компанії. Остання версія, Xcode 16, випущена 16 вересня 2024 року, доступна безкоштовно в AppStore для macOS або на веб-сайті Apple Developer та надає навіть більше можливостей для вбудованої діагностики. Хоча складно визначити точну кількість користувачів Xcode, за оцінками, 34% всіх розробників використовують Xcode [2, 3] як основне інтегроване середовище розробки для створення додатків для платформ Apple, що є надзвичайно великим показником популярності програмного забезпечення.

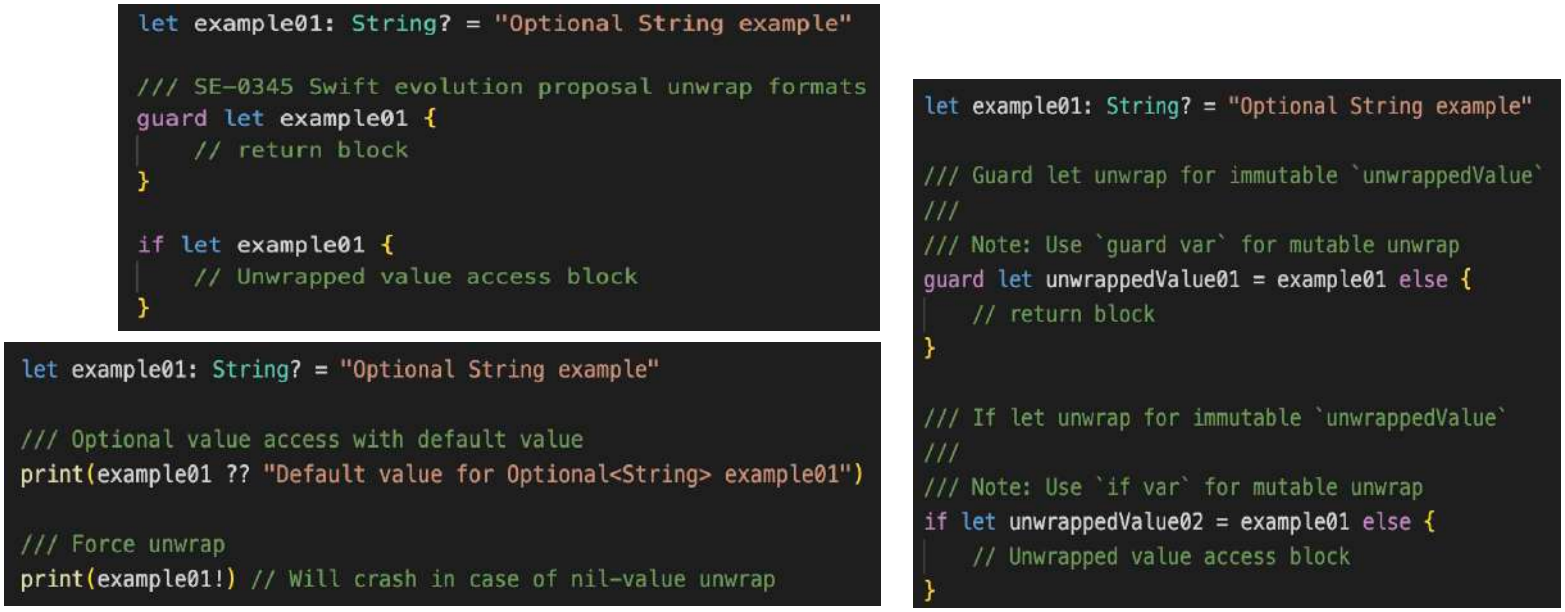
Зважаючи на популярність інтегрованого середовища розробки Xcode та важливість доступу до надійного інструменту оцінки стабільності програмного забезпечення, було вирішено сконцентруватись на побудові процесу автоматизації аналізу програмного коду, що використовує наявні можливості середовища розробки щодо інтеграції пакунків мовою програмування Swift. Таким чином, потенційний користувач матиме можливість застосування інструменту оцінки стабільності програмного забезпечення без необхідності використання сторонніх сервісів, які не інтегровані безпосередньо в інструменти розробки Apple.

1.2 Методи автоматизованого аналізу стабільності програмного забезпечення

Досліджувані параметри стабільності програмного забезпечення складають одну з найважливіших нефункціональних характеристик, що визначає його якість та надійність в експлуатації. Вона охоплює здатність системи коректно функціонувати під очікуваним навантаженням, стійкість до помилок та непередбачених умов, а також передбачуваність поведінки протягом тривалого часу. У контексті розробки на Swift, де безпека типів та управління пам'яттю є ключовими аспектами мови, автоматизований аналіз стабільності набуває особливого значення для виявлення як специфічних для платформи проблем, так і загальних помилок програмування. Так, автоматизований аналіз стабільності – це деяка сукупність методів та інструментів, що дозволяють виявляти та оцінювати потенційні проблеми розроблювального програмного забезпечення. Основна мета такого аналізу полягає у ранньому виявленні дефектів, зниженні ризиків в опублікованих версіях програмного забезпечення та підвищенні загальної надійності й стійкості програмного коду.

Статистичний аналіз вихідного коду написаного мовою програмування Swift відбувається у декілька етапів, використовуючи вбудовані фундаментальні механізми Swift. Одним з основних механізмів вбудованого статистичного аналізу є система типів та опціоналів.

Строга статична типізація запобігає багатьом помилкам невідповідності типів під час компіляції збірки, в той час, як система опціоналів, або опціональних типів, призначена для безпечної роботи зі значеннями, що можуть бути відсутні, значно знижуючи ймовірність фатальних збоїв через розіменування нульових посилань, які є частими в інших мовах програмування, як наприклад Objective-C або C++.



```swift
let example01: String? = "Optional String example"

/// SE-0345 Swift evolution proposal unwrap formats
guard let example01 {
    // return block
}

if let example01 {
    // Unwrapped value access block
}
```

```swift
let example01: String? = "Optional String example"

/// Optional value access with default value
print(example01 ?? "Default value for Optional<String> example01")

/// Force unwrap
print(example01!) // Will crash in case of nil-value unwrap
```

```swift
let example01: String? = "Optional String example"

/// Guard let unwrap for immutable `unwrappedValue`
///
/// Note: Use `guard var` for mutable unwrap
guard let unwrappedValue01 = example01 else {
    // return block
}

/// If let unwrap for immutable `unwrappedValue`
///
/// Note: Use `if var` for mutable unwrap
if let unwrappedValue02 = example01 else {
    // Unwrapped value access block
}
```

Лістинг 1.1 Приклади роботи з опціоналами: а) Розгортання опцінальних типі у SE-0345, б) Виведення значень опціоналів, в) Розгортання значення опціоналів у змінну

Іншими складовими вбудованого статистичного аналізу є Статичний Аналізатор Xcode (eng: Clang Static Analyzer) [25], що хоча й орієнтований на C/Objective-C, також аналізує взаємодію Swift з цими мовами, та Статичний Аналіз Багатопотокості (eng: Swift Concurrency) [26] – нова модель багатопотокості з async/await та actor-ами, що включає компіляторні перевірки ізоляція акторів та потікобезпечності для статичного запобігання значній частині станів гонитви даних (eng: data races).

Таким чином, для автоматизованої оцінки стабільності Swift-проектів вже існує набір вбудованих статичних можливостей мови та середовища розробки Xcode, що забезпечують хоч і обмежений у функціональних можливостях, проте комплексний підхід для раннього виявлення можливих вразливостей архітектури програмного коду, створюючи ключовий етап до створення надійного та стабільного програмного забезпечення на мові програмування Swift.

Подальший аналіз буде присвячено методам оцінки зрозумілості коду автоматизованого аналізу програмного забезпечення Swift буде присвячено

оцінці вагових коефіцієнтів метрик стабільності програмного забезпечення, що було вкладено в основу розробленого інструменту управління пакунками Swift, та існуючих рішень щодо автоматизації популярних сторонніх та вбудованих методів комплексної оцінки стабільності програмного коду.

1.3 Обмеження існуючих підходів

Незважаючи на обширність існуючих підходів оцінки стабільності програмного забезпечення, розробленого на мові програмування Swift, та методів до їх застосувань, комплексна оцінка програмного коду не є досконалою через суттєві обмеження вбудованих та розроблених сервісів. Ці обмеження створюють потребу в розробці нових або розширенні існуючих підходів для отримання більш повної та точної оцінки якості програмного забезпечення.

Однією з найбільш поширених проблем є обмеженість покриття. Використання лише статичних вбудованих перевірок обмежує здатність аналізатора виявляти вразливості, пов‘язані з конфігурацією або виконанням програмного коду. В той час, як динамічні аналізатори навпаки виявляють лише ті вразливості, що було покрито під час тустування програмного забезпечення. Так, якість оцінки суттєво залежить від вибору тестів для динамічного аналізу [28] в поєднанні з доданими статичними перевірками. Прикладами такого аналізу можуть слугувати вбудовані механізми перевірки використання пам‘яті, як: ARC, Memory Graph Debugger або використання Xcode Instruments (Leaks, Allocations, Zombies, тощо) [29, 30].

Іншими поширеними недоліками існуючих підходів є хибні спрацювання, брак контексту тестування та обмеженість масштабованості. Статичні аналізатори, хоч і пропонують глобальний аналіз, проте дуже часто генерують досить велику кількість попереджень, які насправді не є реальними недоліками розробленої архітектури, однак обмежують використання нових патернів та структур зв‘язки програмного коду. В даному контексті, такі результати

ускладнюють аналіз даних та можливих вразливостей, тим самим знижуючи довіру до інструменту. Схожим чином, автоматизовані інструменти аналізують код синтаксично та семантично, але не розуміють розробленої бізнес-логіки, архітектурних рішень чи специфічних вимог проекту.

Так, прикладами найбільш популярних сторонніх сервісів для статичного аналізу програмного коду Swift є: SwiftLint, SonarCube та SWAN.

SwiftLint [6] – це статичний аналізатор, що слугує як інструмент для забезпечення дотримання стилю Swift, що було розроблено на архівованому GitHub Swift Style Guide [7]. Цей інструмент є чудовим засобом перевірки ризикованих конструкцій у Swift коді. Проте, як було згадано, схожі засоби перевірки програмного коду зазвичай не дозволяють виконувати комплексну оцінку архітектури, оскільки не здатні розуміти семантику програмного коду. Іншою вражливість цього інструменту є ізольований аналіз кожного файлу, так аналіз залежностей не відбувається взагалі.

SonarCube [8] у свою чергу є платформою з відкритим вихідним кодом, що була розроблена у 2008 році компанією SonarSource. Цей інструмент насамперед розроблено для автоматизованого перегляду коду за допомогою статичного аналізу. SonarCube є трохи ширшим за SwiftLint, оскільки підтримує інші мови програмування та різні CI/CD конвеєри, також допомагає розробникам виявити покимлки та потенційні проблеми в програмному коді, що можуть бути вразливостями безпеки програмного забезпечення. Однією з найбільших переваг SonarCube є централізований аналіз якості коду, що також можливо відстежувати з часом. Незважаючи на широкий функціонал та глибинний аналіз, цей інструмент успадковує всі недоліки попередньо описаного SwiftLint, оскільки має пряму залежність від інших аналізаторів. Якість аналізу стилю та багатьох недоліків програмного коду Swift часто залежить від інтеграції зі SwiftLint. Іншим недоліком є зрілість SAST для Swift. Власні правила статичного аналізу безпеки SonarQube для Swift є менш вичерпними порівняно з іншими мовами програмування і можуть генерувати хибні спрацьовування, що вимагає

додаткового аналізу отриманих результатів оцінки програмного забезпечення. Також, інструмент надає лише обмежений функціонал для безкоштовної версії, що робить його менш використовуваним для власних mock-проектів.

Насамкінець, SWAN [9] – WIP (work in progress) академічний фреймворк для глибокого статичного аналізу Swift-коду, що працює на рівні SIL (Swift Intermediate Language). Цей інструмент фокусується переважно на потоках даних, відстеженні зображення даних (eng: taint analysis) та аналізі станів типів. SWAN додає значний інструментарій, що потенційно здатен виявляти складні помилки та вразливості, особливо пов'язані з безпекою даних та API, які можуть бути пропущені простішими лінтерами чи аналізаторами Xcode завдяки глибшому аналізу SIL. Проте, незважаючи на активний розвиток, це все ще академічний проект у стані розробки, що може означати брак стабільності самого інструменту, повноти документації, простоти використання та підтримки новіших версій Swift. До того ж, інструмент має досить вузьку спеціалізацію, зменшує кількість досліджуваних метрик програмного коду. Так, можна також стверджувати, що фреймворк поки бракує масштабованості та продуктивності через відсутність гарантії ефективності та швидкості на великих проектах.

1.4 Визначення та застосування вагових коефіцієнтів для метрик якості якості коду

Відповідно до проведеного аналізу інструменту керування пакунками Swift “Stability Assurance Tool”, розробленого в рамках курсової роботи, а також з огляду на обмеження існуючих засобів статичного аналізу, було ухвалено рішення про перегляд і коригування вагових коефіцієнтів, а також способу їх застосування до раніше впроваджених метрик оцінювання стабільності програмного забезпечення.

Різні метрики мають неоднакову вагу та вплив на якість кінцевого продукту й процес розробки. Саме тому виникла необхідність у введенні вагових коефіцієнтів – числових множників, які присвоюються кожній метриці з метою

відображення її відносної значущості в загальній оцінці. У результаті було розроблено оновлений механізм розрахунку інтегрального показника стабільності програмного забезпечення, що враховує вагові коефіцієнти для кожної метрики.

Ці коефіцієнти відображають ступінь суворості кожного з досліджуваних показників і є конфігурованим параметром, який може змінюватися відповідно до потреб конкретного проєкту. Такий підхід дає змогу:

- пріоритезувати зусилля з покращення коду, фокусуючись на найбільш критичних аспектах
- агрегувати різнорідні показники в єдиний інтегральний індекс якості
- адаптувати оцінювання до специфіки проєкту, команди або бізнес-вимог
- створити більш збалансовану картину, в якій окремі критичні проблеми не губляться серед численних незначних недоліків.

Ще одним удосконаленням раніше створеного інструменту оцінки стабільності стало впровадження нової метрики – LOCM (Lack of Cohesion of Methods), яка є складовою частиною моделі, описаної у праці Shyam R. Chidamber та Chris F. Kemerer “A Metrics Suite for Object-Oriented Design” [10]. Зазначена модель стала концептуальною основою Stability Assurance Tool.

LOCM – це метрика, що вимірює зв’язність методів класу стосовно атрибутів, до яких ці методи звертаються. Високе значення LOCM вказує на сильну зв’язність – тобто методи використовують багато спільних атрибутів, тоді як низьке значення свідчить про фрагментовану структуру, де методи працюють із різними наборами даних.

Беручи до уваги особливості синтаксису мови Swift, алгоритм розрахунку LOCM було реалізовано таким чином: для кожної пари методів у класі визначається перетин множин атрибутів, до яких вони звертаються. LOCM обчислюється як сума розмірів цих перетинів, нормалізована за кількістю пар методів.

Оцінювання отриманих значень здійснюється з урахуванням зазначених вагових коефіцієнтів або, за відсутності користувацької конфігурації, відповідно до типового масштабу програмного забезпечення. У середньому, задовільним вважається значення LOCM у діапазоні від 3.0 до 9.0, що свідчить про належний рівень зв’язності коду. Водночас, ці значення можуть змінюватися залежно від конкретного класу та контексту його використання.

1.5 Можливості до вдосконалення

Теперішня версія розробленого статичного аналізатора хоч і дозволяє вже виконувати якісну комплексну оцінку стабільності програмного забезпечення, що базовано на фундаментальній праці в галузі дослідження ефективності та застосування метрик оцінки програмного коду, а також пов‘язано з засобами обробки синтаксичний особливостей мови програмування Swift, все не є бездоганним рішенням щодо оцінки архітектурного модулю. Це дає змогу до розробки ряду потенційних вдосконалень для покращення роботи існуючого інструменту.

Насамперед, варто зауважити, що найбільша перевага цього інструменту – фундаментальна дослідницька база – є також деяким недоліком цього проекту, оскільки, хоч оцінка і є збалансованою за мірками об‘єктно-орієнтованого дизайну, проте не враховує дизайні особливості Swift проектів. В останні декілька років можна помітити стрімкий розвиток Swift та дизайних рішень, які є характерними для цієї мови. Тому, виникає потреба підтримки протокольно-орієнтованого стилю для якісною оцінки архітектури. Відповідно, розширення наявного інструментарію для аналізу не лише об‘єктно-орієнтованої моделі програмного забезпечення, а й моделей побудови зв‘язків між компонентами програмного коду характерних для мови Swift, є досить важливою потенційною можливостю до вдосконалення Stability Assurance Tool.

Водночас, вивчення інших існуючих рішень для статичного аналізу коду, як вбудованих, так і сторонніх, підтвердило їхню практичну користь. Навіть з урахуванням наявних обмежень, ці інструменти активно застосовуються, оскільки пропонують цінні можливості для виявлення помилок та покращення якості програмного забезпечення. Таким чином, однією з інших можливостей для вдосконалення розробленого інструментарію є поєднання найкращих якостей інший сервісів для створення глибинного аналізу, що враховував би переваги та недоліки рішень, що підтвердили свою практичну користь.

Відповідно, послуговуючись здобутими знаннями, можна зробити висновок, що аналіз методів оцінки стабільності та зрозумілості програмного коду хоч і є основною та найбільш вивченою частиною цього дослідження, проте досі має достатньо можливостей до вдосконалення для того, щоб визначити цілі майбутньої роботи над проектом.

Розділ 2. Розширення функціональності аналізатора

2.1 Загальний опис

Оперуючи інформацією з попереднього розділу, варто зауважити, що сучасна розробка програмного забезпечення на мові Swift значною мірою спирається на використання різноманітних інструментів автоматизованого аналізу коду, які допомагають підвищувати його якість, надійність та підтримуваність. Існуючий інструментарій вже надає доступ до широкого спектру функціональних можливостей: від перевірки дотримання стандартів об‘єктно-орієнтованої моделі та опціонального використання окремих метрик оцінки стабільності програмного забезпечення.

Незважаючи на цінність кожної з наявних функціональних можливостей, їх застосування на практиці часто виявляє певні обмеження, як було детально розглянуто в першому розділі дослідження. Аналіз зазвичай є фрагментованим, результати використання поодиноких метрик додає складність агрегування та інтерпретування, а комплексна автоматична оцінка, що враховує одночасно різні аспекти стабільності, залишається актуальною проблемою. Також, відсутність інтегрованого рішення щодо автоматизації оцінювання програмного забезпечення створює додаткові недоліки, що запобігають використанню системи у довгостроковій перспективі.

Саме для подолання виявлених у попередньому розділі обмежень, таких як фрагментованість аналізу, складність агрегації різнорідних метрик та недостатня увага до автоматизованої оцінки як стабільності, так і зрозумілості коду, у даній роботі було вирішено запропоновати нову систему функціональних можливостей.

Ключовий підхід запропонованого рішення полягає у застосуванні конфігурованої вагової моделі для розрахунку інтегральних показників, що можуть бути налаштовані користувачем, та інтеграцією з середовищем розробки програмного забезпечення. Система не просто збирає дані про загальну оцінку

стабільності архітектурного модуля, а й візуалізує проблемні ділянки коду, інтегрується з системою збірки проекту, що також надає можливість відстеження динаміки якості. Особлива увага приділена можливості гнучкого налаштування правил та вагових коефіцієнтів, щоб адаптувати процес оцінки до вимог конкретного проекту чи команди.

Очікується, що запропоноване розширення дозволить розробникам та командам отримувати більш цілісний, об'єктивний та дієвий зворотний зв'язок щодо стану їхнього коду, сприяючи виявленню потенційних проблем зі стабільністю в довгостроковій перспективі. Детальний опис архітектури розробленого підходу, його основних функціональних можливостей, використаних алгоритмів оцінки та способів інтеграції в процес розробки представлено у наступних підрозділах цього розділу.

2.2 Інтеграція аналізу коду у середовище розробки

Ефективність будь-якого інструменту для аналізу коду, особливо в контексті оцінки таких складних характеристик, як стабільність та зрозумілість, значною мірою залежить не лише від його аналітичних можливостей, але й від того, наскільки зручно та безшовно він інтегрований у щоденний робочий процес розробника та автоматизовані конвеєри збірки проекту CI/CD. Саме глибина та якість інтеграції визначають, чи стане аналізатор дієвим помічником, що сприяє ранньому виявленню проблем, скороченню циклу зворотного зв'язку та підвищенню загальної культури якості в команді, чи залишиться зовнішнім інструментом, який використовується епізодично або ігнорується через незручність.

Для проектів на мові програмування Swift, розробка яких переважно ведеться в інтегрованому середовищі Xcode, екосистема надає кілька поширених механізмів для вбудовування сторонньої функціональності, включаючи інструменти аналізу коду. Ці механізми дозволяють інтегрувати аналіз на різних рівнях: від надання миттєвих підказок безпосередньо в редакторі коду до

автоматичних перевірок якості при кожній збірці проекту на сервері CI/CD. Кожен з цих підходів має свої специфічні переваги та недоліки, які впливають на можливості автоматизації оцінки якості коду, швидкість отримання результатів та зручність використання. У цьому підрозділі розглянуто основні з цих механізмів інтеграції та описано підхід, обраний для розробленого аналізатора стабільності програмного коду.

Одним з основних та найбільш поширених методів розповсюдження пакунків на мові програмування Swift є менеджери залежностей, від вбудованих, як наприклад Swift Package Manager [11], до сторонніх сервісів інтегрованих для Swift розробки: CocoaPods, Carthage [20, 21]. Swift Package Manager – це інструмент для керування розповсюдженням коду Swift. Він інтегрований з системою збірки Swift для автоматизації процесу завантаження, компіляції та зв'язування залежностей. Цей менеджер залежностей доступний з версії Swift 3.0, що було випущено у вересні 2016 року.

У загальному розумінні, Swift проекти зазвичай організовують код в окремі модулі. Кожен модуль визначає простір імен і забезпечує контроль доступу до частин коду, які можуть бути використані за межами модуля. Програма може містити весь свій код в одному модулі, або ж імпортувати інші модулі як залежності. За винятком кількох системних модулів, таких як Darwin у macOS або Glibc у Linux, більшість залежностей вимагають завантаження та збірки коду для використання. Таким чином, коли користувач використовує окремий модуль для коду, який вирішує певну проблему, цей код можна повторно використовувати в інших ситуаціях. Наприклад, модуль, який надає функціональність для здійснення мережевих запитів, можна використовувати спільно з іншими додатками. Використання модулів дозволяє спиратися на код інших розробників замість того, щоб повторно реалізовувати ту саму функціональність самостійно.

Відповідно, менеджери залежностей вирішують основну проблему організації програмного коду – управління зовнішніми бібліотеками та

фреймворками, від яких власне залежить проект. Такий спосіб вбудування сторонніх програмних модулів протягом останнього часу став стандартизованим методом управлінням версіями доданих модулів, що забезпечує легку інтеграцію з розроблюваним програмним забезпеченням. До того ж, враховуючи, що SPM є нативним рішенням від Apple, така інтеграція стороннього коду має ряд вбудованих оптимізацій.

Хоча використання менеджерів залежностей, таких як Swift Package Manager та CocoaPods, значно спрощує процес додавання та оновлення інструментів аналізу в проекті, цей підхід також має свої обмеження. По-перше, не всі бажані модулі можуть бути доступні у вигляді готових пакетів для SPM чи Pods. По-друге, вбудовані механізми для запуску аналізу часто пропонують меншу гнучкість у налаштуванні параметрів та умов запуску порівняно з використанням прямих скриптів у фазах збірки проекту в Xcode. Так, менеджери залежностей є чудовим засобом розповсюдження, але їх можливостей не завжди достатньо для повноцінної та гнучкої інтеграції аналізу в процес збірки.

Ще одним підходом є використання зовнішніх інструментів та скриптів, які запускаються незалежно від процесу збірки в Xcode. Аналіз може ініціюватися вручну з командного рядка, автоматично через Git hooks, або, що найчастіше, як етап у конвеєрі CI/CD. Завдяки високій гнучкості налаштувань та незалежності від середовища розробки, цей метод особливо зручний для проведення повного аналізу проекту на CI-серверах та генерації детальних звітів. Однак, головним недоліком є відсутність миттєвого зворотного зв'язку для розробника безпосередньо під час написання коду в середовищі розробки. Крім того, автоматизація такого підходу вимагає налаштування та підтримки окремої інфраструктури.

Нарешті, ще одним механізмом інтеграції є розширення Xcode (eng. Xcode Extensions) та Xcode Plugins. Цей підхід був представлений Apple разом з Xcode 8 у вересні 2016 року, як офіційна та безпечна заміна попередній системі плагінів, яка дозволяла завантажувати довільний код безпосередньо в процес

Xcode, що створювало ризики для стабільності та безпеки середовища розробки, і тому була визнана застарілою.

Сучасні розширення редактора коду працюють як окремі, ізольовані процеси, що взаємодіють з Xcode через спеціальний фреймворк XcodeKit [22] та механізм міжпроцесної взаємодії XPC. Такий підхід гарантує, що розширення не може безпосередньо вплинути на стабільність самого Xcode. Основні можливості таких розширень включають додавання нових команд та програмну взаємодію з вмістом активного файлу в редакторі: читання буфера тексту, аналіз та модифікація виділеного тексту або всього файлу. Це робить їх привабливими для реалізації функцій форматування коду, швидких редакцій коду, генерації коду за шаблоном або інтеграції сторонніх модулів, що надають зворотний зв'язок безпосередньо в редакторі. Однак, саме через міркування безпеки та стабільності, API розширень XcodeKit має суттєві обмеження. Розширення працюють обмеженому контейнері і не мають прямого доступу таких функцій, як:

- Файлова система поза межами власного
- Загальна структура проекту чи індексу символів Xcode
- Налаштування збірки проекту
- Виконання тривалих або ресурсоємних аналітичних завдань, які можуть блокувати інтерфейс користувача.

У свою чергу, плагіни інструментів збірки, що є частиною PackagePlugin API [23] є деяким визначення всередині раніше згаданих пакунків Swift. Таким чином, пакунок отримує можливість автоматизації завдань, інтеграції зовнішніх інструментів у процес збірки або надання налаштовуваних команд для роботи з проектом. Схоже до розширень редактору коду, плагіни працюють в ізольованому середовищі з певними обмеженнями на доступ до файлової системи чи мережі, які можна налаштовувати додатково через запит дозволів [24]. Існують два основні типи плагінів – Command Plugins та Build Tool Plugins. Додатково, один пакунок може бути одночасно двома типами. Build Tool Plugins

запускають основний пакунок, поширений через Swift Package Manager, під час процесу збірки, до або паралельно з компіляцією, і зазвичай використовуються для генерації вихідного коду або виконання інших підготовчих завдань. Command Plugins запускаються на вимогу користувача через меню Xcode або через команди менеджеру пакунків Swift у терміналі і можуть виконувати ширший спектр дій, таких як форматування коду, генерація документації, аналіз залежностей тощо.

Перелічені обмеження, що є спільними для плагінів та розширень редактору коду, означають, що ці методи добре підходять для швидких операцій над текстом поточного файлу, але малопридатні для комплексного аналізу всього проекту, який потребує доступу до багатьох файлів, налаштувань збірки чи значних обчислювальних ресурсів. Таким чином, вони є цінним інструментом для покращення локального досвіду розробника, але не замінюють інструменти, інтегровані через фази збірки або CI/CD для глобального аналізу проекту.

2.3 Автоматизація процесу оцінки коду через Build Phases Scripts у Xcode

Аналізуючи досліджену інформацію та сучасні підходи до інтеграції інструментів розробки, для практичної реалізації автоматизованої оцінки коду в рамках даної роботи було вирішено суттєво переробити та розширити основну структуру поширення аналізатору стабільності програмного забезпечення Stability Assurance Tool. Початкова версія, що розповсюджувалася виключно як стандартний пакунок через Swift Package Manager (SPM), мала певні обмеження щодо гнучкості інтеграції у процес розробки та взаємодії з середовищем Xcode, особливо враховуючи особливості стандартних плагінів SPM. Розуміючи важливість швидкого налаштування, легкої конфігурації метричних даних та негайного зворотного зв'язку для розробника, аналізатор було доповнено окремим, динамічно оновлюваним, виконуваним файлом (Executable Target) в межах SPM пакунку. Цей виконуваний файл містить збудований бінарний файл аналізатора, що включає усі актуальні зміни кожної останньої збірки пакунку.

Такий підхід дозволяє інструменту успадковувати концептуальні переваги плагінів збірки (Build Tool Plugins), як-от автоматизація аналізу під час компіляції, але водночас долає ключове обмеження закритого контейнеру виконання, характерне для таких плагінів. На відміну від ізольованого середовища Build Tool Plugin з обмеженим доступом до ресурсів, бінарний файл аналізатора, викликаний через скрипт у фазі збірки (Build Phases) Xcode, виконується як стандартний системний процес. Це надає йому повний доступ до файлової системи проекту, змінних середовища та інших ресурсів системи в межах дозволів користувача, що є критично важливим для проведення комплексного аналізу стабільності, який може вимагати доступу до різних файлів проекту чи конфігурацій.

Інтеграція в процес збірки Xcode реалізується шляхом додавання простого скрипта запуску до секції "Build Phases" цільового проекту. Основне завдання скрипта – викликати виконуваний файл аналізатора з відповідного пакунку Swift. Цей механізм забезпечує значну конфігураційну гнучкість, оскільки параметри аналізу, порогові значення метрик чи специфічні шляхи можна передавати як аргументи командного рядка безпосередньо у скрипті. Крім того, скрипт має доступ до стандартних змінних середовища Xcode, таких як $SRCROOT чи $TARGET_NAME, що дозволяє аналізатору автоматично адаптуватися до контексту конкретної збірки та конфігурації проекту.

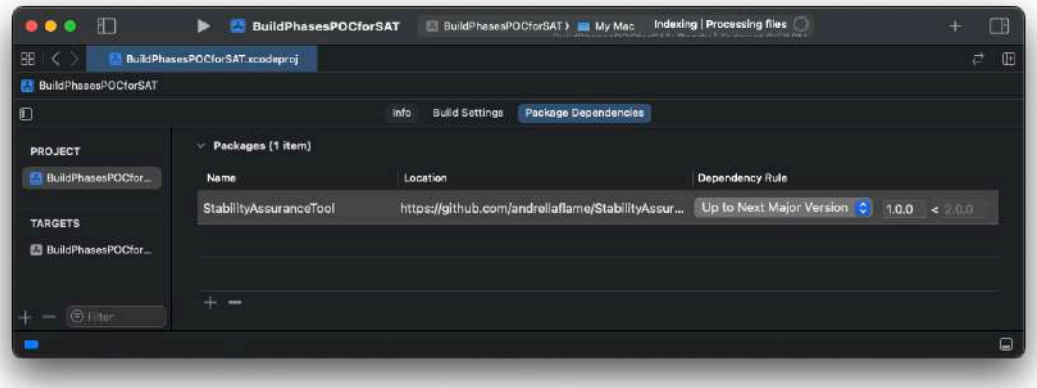


Рис. 2.1 Приклад інтеграції пакунку Swift з вирористанням SPM

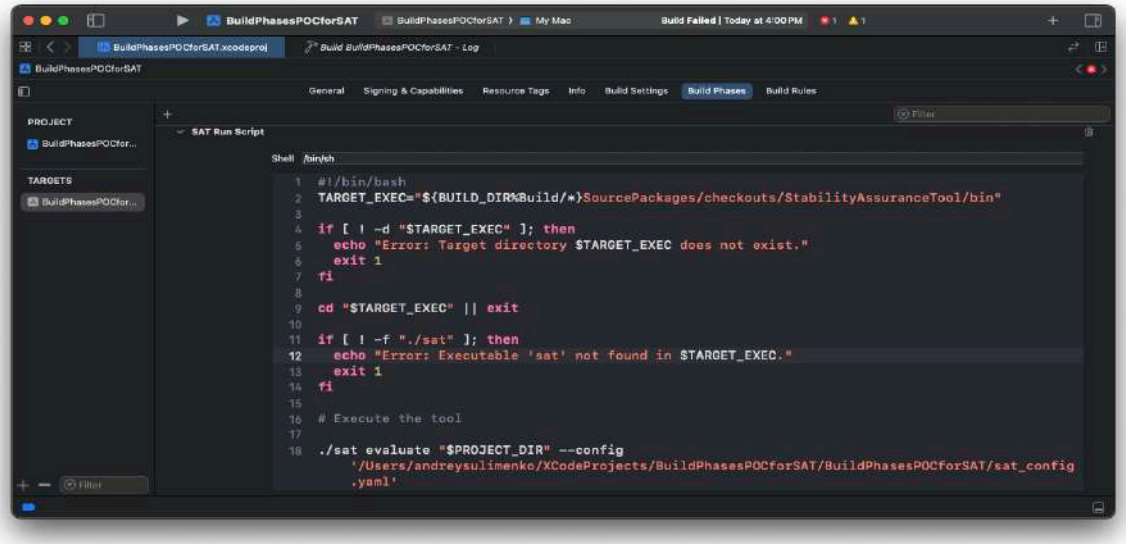


Рис. 2.2 Приклад використання Build Phases scripting

Ключовою перевагою використання виконуваного файлу у Build Phases є можливість безпосередньої інтеграції результатів аналізу в діагностичну систему та інтерфейс Xcode. Для цього аналізатор форматує свій вивід – повідомлення про потенційні проблеми стабільності, порушення визначених метрик тощо – відповідно до специфічного шаблону, який розпізнається середовищем розробки для відображення помилок та попереджень. Завдяки такому форматуванню, розробники отримують миттєвий зворотний зв'язок: попередження та помилки з'являються у навігаторі проблем (eng. Issue Navigator), а відповідні рядки коду підсвічуються безпосередньо в редакторі. Це дозволяє оперативно реагувати на потенційні проблеми стабільності архітектури під час написання коду та компіляції, не покидаючи звичного робочого середовища.

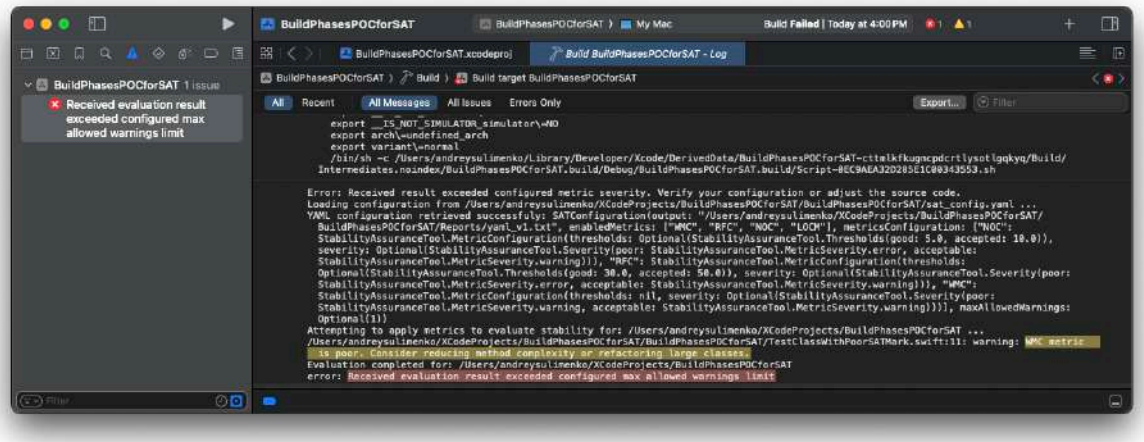


Рис. 2.3 Приклад виконання Stability Assurance Tool з виводом результатів у Issue Navigator

Водночас, попри переваги інтеграції через виконуваний файл та Build Phases, аналізатор зберіг відповідність до стандартного Command Tool Plugin. Це забезпечує альтернативний шлях використання інструменту для розробників, які працюють переважно з командним рядком, або для інтеграції аналізу в системи безперервної інтеграції та розгортання, де виклик відбувається через команди терміналу.

Таким чином, обрана гібридна архітектура поєднує гнучкість та потужність виконання аналізу як повноцінного процесу з тісною інтеграцією в робочий процес розробника в Xcode, зберігаючи при цьому сумісність зі стандартними інструментами екосистеми Swift Package Manager.

2.4 Налаштування гнучкості конфігурації аналізатора. Використання YAML-конфігурацій для налаштування процесу оцінки

Ефективна система оцінки якості коду має бути не лише потужною, а й гнучкою, адже різні проекти, команди та навіть окремі модулі в межах одного проекту можуть мати різні вимоги до стабільності, зрозумілості й пріоритетів аналізу програмного забезпечення. Універсальний підхід із правилами та

порогами заданими за замовчуванням часто виявляється неефективним, оскільки може створювати надлишковий шум для одних проектів і бути недостатньо суворим для інших. Таким чином, критично важливою є можливість гнучкого налаштування процесу аналізу відповідно до конкретного контексту. Для забезпечення такої адаптивності в розробленому аналізаторі Stability Assurance Tool було впроваджено механізм конфігурації на основі YAML-файлів.

YAML Ain't Markup Language (YAML) є мовою серіалізації даних [17], яка постійно входить до списку найпопулярніших мов програмування. YAML часто використовується як формат для конфігураційних файлів та у додатках, що зберігають або передають дані структуровані. Так, її можливості серіалізації об'єктів роблять її життєздатною заміною таким мовам, як JSON.

Власні типи даних також дозволені, але YAML за замовчуванням кодує скаляри, списки та асоціативні масиви для оптимізації збережених даних. Синтаксис YAML було створено за стандартами RFC. Відповідно, YAML призначений для читання та запису в потоках.

Завдяки простому синтаксису на основі відступів, що забезпечує легкість читання й розуміння, нативній підтримці складних структур даних, можливості додавання коментарів для пояснення параметрів або тимчасового вимкнення налаштувань, а також широкій поширеності та доступності парсерів, формат YAML було обрано для створення конфігураційних файлів Stability Assurance Tool. Це рішення дозволяє уникнути громіздкості XML та нестачі структури в інших форматах.

За допомогою YAML-файлу користувачі розробленого статичного аналізатора Stability Assurance Tool отримують можливість гнучко керувати різними аспектами процесу оцінки програмного забезпечення, адаптуючи його під специфічні потреби своїх проєктів та команд. Ця гнучкість реалізується через низку конфігурованих параметрів.

По-перше, надається можливість вибірково активувати чи деактивувати певні перевірки стабільності або цілі групи попередньо визначених правил. Це дозволяє користувачам фокусувати аналіз на тих аспектах, які є найбільш релевантними для поточного етапу розробки або типу проєкту, ігноруючи менш значущі попередження.

По-друге, ключовим елементом конфігурації є визначення граничних значень для різноманітних метрик коду. Наприклад, розробники можуть встановити максимальну допустиму цикломатичну складність для функцій, ліміти на рівень зв'язаності класів (eng. coupling) або мінімальну кількість класів-нащадків для базових класів. Важливо, що перевищення цих встановлених порогів може генерувати попередження або помилки, причому тип реакції системи, попередження чи помилка, також може бути врегульований через параметри у файлі конфігурації, що безпосередньо впливає на спрацювання порогів якості (eng. quality gates) у CI/CD.

Крім того, система дозволяє задавати вагові коефіцієнти для різних метрик або категорій проблем, як це було детально розглянуто в першому розділі. Це забезпечує можливість адаптувати важливість окремих аспектів якості під потреби конкретного проєкту при розрахунку інтегральних показників стабільності програмного забезпечення, надаючи більш зважену фінальну оцінку. Для деяких специфічних правил також передбачена можливість налаштування їхніх внутрішніх параметрів, а окремі метрики можуть підтримувати різні формати або моделі оцінки, що дозволяє ще точніше адаптувати загальний результат під вимоги аналізованого програмного забезпечення.

Нарешті, для зручності та точності аналізу, користувачі мають можливість вказувати шляхи до файлів або директорій з використанням glob-патернів. Це дозволяє як визначати конкретні модулі програми, що підлягають оцінці, так і виключати з аналізу сторонні бібліотеки чи згенерований код, а також вказувати директорії для збереження результатів та звітів. Така деталізація

налаштувань перетворює аналізатор на потужний та адаптивний інструмент контролю якості.

Механізм завантаження та застосування конфігураційного файлу реалізовано з фокусом на автоматизоване використання заданих параметрів. Виконуваний файл аналізатора при запуску автоматично аналізує задані параметри конфігураційного файлу YAML у визначеній директорії, що задана як параметр запуску команди аналізатора. Якщо файл знайдено, конфігураційні дані автоматично оброблено модулем конфігурації всередині збудованого виконуваного файлу Stability Assurance Tool. Отримані налаштування перезаписують або доповнюють стандартні параметри аналізатора. У випадку помилок у синтаксисі YAML або некоректних значень параметрів, аналізатор може видавати відповідне повідомлення про помилку та, залежно від налаштувань, або припинити роботу, або продовжити аналіз зі стандартними налаштуваннями.

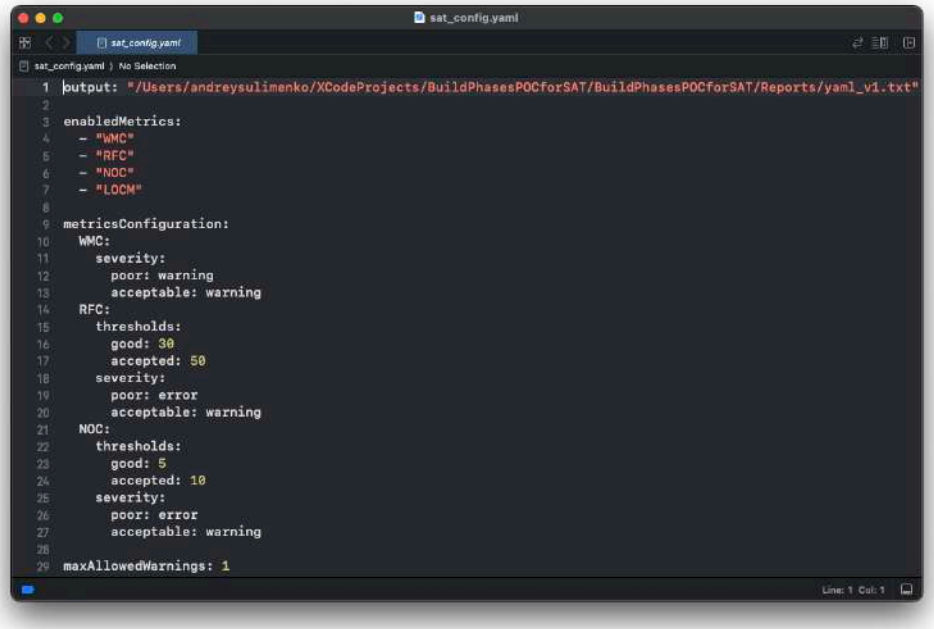


Рис. 2.4 Приклад YAML конфігурації для Stability Assurance Tool

Таким чином, використання YAML-конфігурацій надає розробникам потужний та водночас простий і зрозумілий механізм для адаптації процесу

автоматизованої оцінки стабільності та зрозумілості коду під специфічні потреби їхніх проектів та команд. Така гнучкість є ключовою для практичного впровадження системи контролю якості та отримання релевантних і дієвих результатів.

2.5 Можливості до вдосконалення

Незважаючи на реалізовані вдосконалення та розширення функціональності аналізатора Stability Assurance Tool, описані в цьому розділі, існує значний потенціал для його подальшого розвитку. Подальша робота може бути націлена на кілька ключових напрямків, що передбачають потенційне підвищення практичної цінності інструменту та покращення досвіду його використання розробниками Swift.

Перспективним напрямком є подальше вдосконалення інтеграції інструменту в екосистему розробки та доставки програмного забезпечення. Хоча поточна реалізація забезпечує інтеграцію через Build Phases та SPM Command Plugins, розробка готових плагінів для поширених платформ безперервної інтеграції та доставки, таких як GitHub Actions чи Xcode Cloud, суттєво спростила б процес налаштування автоматизованих перевірок якості, конфігурації порогів якості (eng. Quality Gates) та генерації звітності в рамках командних робочих процесів. Окрім цього, для забезпечення максимальної сумісності, особливо з проектами, що використовують альтернативні системи управління залежностями, може бути розглянута підтримка розповсюдження аналізатора через CocoaPods або Carthage.

Критично важливим напрямком є підвищення дієвості результатів аналізу. Замість прямолінійного звітування про виявлені проблеми чи відхилення метрик, система може бути вдосконалена для надання автоматизованих, контекстно-обґрунтованих рекомендацій щодо рефакторингу або виправлення коду. Для складних проблем стабільності цінним було б впровадження елементів допомоги у визначенні потенційних першопричин.

Подальша інтеграція з системами відстеження завдань, такими як Jira або GitHub Issues, дозволила б автоматизувати процес реєстрації критичних дефектів, виявлених аналізатором, для їх подальшої обробки командою розробників.

Нарешті, фундаментальним напрямком залишається вдосконалення та розширення самого аналітичного ядра інструменту. Це включає постійне розширення набору метрик для більш глибокої оцінки стабільності, особливо аспектів Swift Concurrency, та зрозумілості коду.

Зазначені напрямки вдосконалення демонструють, що розширені функціональні можливості системи перетворюють її на гнучку платформу з потенціалом для майбутнього розвитку. Подальша робота в цих напрямках дозволить створити ще більш потужний та зручний інструмент для забезпечення високої якості перевірки стабільності програмного коду, розробленого на мові Swift.

Розділ 3. Реалізація та оптимізація аналізатора

3.1 Загальний опис

Розробка інструментів для автоматизованого аналізу програмного коду, таких як аналізатори стабільності чи зрозумілості, зазвичай спирається на подібні архітектурні принципи та проходить через кілька ключових етапів.

Типовий процес починається з синтаксичного аналізу, або парсингу, вихідного коду для побудови його проміжного представлення, найчастіше у вигляді абстрактного синтаксичного дерева AST (eng. abstract syntax tree) [4], що є представленням структурованих даних системи або її фрагменту, або графу потоку керування CFG (eng. Control-flow graph) [5], що у свою чергу є представленням у вигляді нотації графів, яке може бути пройдено програмою під час її виконання. На основі цих структур виконуються наступні стадії аналізу: застосування набору правил для виявлення специфічних патернів, проведення аналізу потоків даних для відстеження розповсюдження значень змінних та виявлення таких проблем, як використання неініціалізованих змінних чи потенційні витоки ресурсів, або аналізу потоку керування для розуміння можливих шляхів виконання програми. Результати цих етапів агрегуються, обробляються та представляються користувачу у вигляді звітів, попереджень чи інтегральних оцінок якості.

Однак, практична реалізація таких аналізаторів стикається з низкою суттєвих викликів. По-перше, це складність повного та коректного парсингу сучасних мов програмування, як Swift, що постійно розвиваються. По-друге, забезпечення масштабованості та прийнятної продуктивності аналізу для великих кодових баз є нетривіальною задачею, особливо для глибоких методів аналізу потоків даних чи керування. По-третє, лінійна залежність між точністю аналізу, як мінімізацією хибних спрацьовувань, та повнотою, тобто мінімізацією пропусків реальних помилок. Нарешті, важливим викликом є ефективна інтеграція результатів аналізу в робочі процеси розробників та системи CI/CD таким чином, щоб зворотний зв'язок був своєчасним, зрозумілим та дієвим.

Враховуючи ці загальні архітектурні підходи та виклики імплементації, було здійснено перепланування практичної реалізації аналізатора Stability Assurance Tool, описаного концептуально в попередньому розділі, спираючись на досліджену інформацію на результати здобуті у першій версії статичного аналізатору програмного коду. Таким чином, цей розділ детально висвітлює конкретні рішення, застосовані при розробці інструменту: програмну реалізацію його архітектури як SPM-інструменту, методи, використані для оптимізації процесу аналізу та обробки результатів, реалізацію механізму гнучкої конфігурації рівнів критичності метрик, що було згадано у попередньому розділі, як важливе рішення оптимізації попередньої версії, а також підхід до формування загальної оцінки якості та її впровадження в типові сценарії розробки програмного забезпечення.

3.2 Архітектурний дизайн SPM-інструменту для аналізу стабільності коду

Перехід від загальних принципів та викликів реалізації аналізаторів коду, розглянутих у попередньому підрозділі, до конкретного втілення розробленої системи, визначає ефективну реалізації функціоналу оцінки стабільності програмного коду. Концепція розробленого аналізатору була представлена в розділі 2, де аналізатор Stability Assurance Tool було спроектовано як модульний пакунок Swift Package Manager. Така архітектура дозволяє чітко розмежувати логічні частини системи та сприяє її підтримуваності й розширюваності. Далі буде детально розглянуто основні елементи цієї архітектури та їхню взаємодію.

В основі архітектурного дизайну аналізатора Stability Assurance Tool лежить структура пакунку Swift Package Manager, визначена у файлі маніфесту Package.swift. Такий підхід забезпечує модульність, керованість залежностей та стандартизоване розповсюдження інструменту в екосистемі Swift. Ключовими компонентами розробленого пакунку є кілька цілей (eng. targets), що взаємодіють між собою.

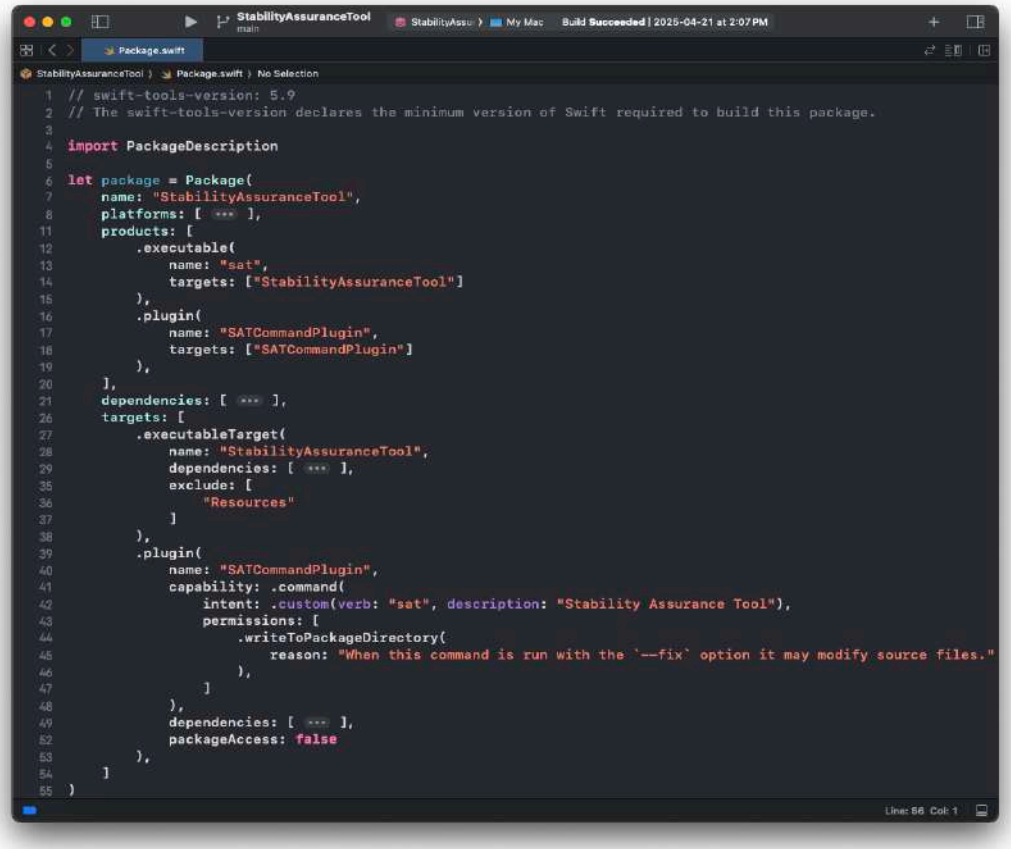


Рис. 3.1 Конфігурація маніфест файлу інструменту Stability Assurance Tool

По-перше, було визначено виконувану ціль (eng. Executable Target). Цей модуль є основним програмним продуктом пакунку та містить точку входу. Він відповідає за оркестрацію всього процесу аналізу: парсинг аргументів командного рядка, з використанням, наприклад, бібліотеки ArgumentParser [12], зчитування та валідацію конфігурації, виклик ядра аналізу, обробку результатів та їх форматування для виводу. Саме цей скомпільований виконуваний файл викликається скриптами у фазах збірки Xcode, що дозволяє обійти обмеження файлової системи щодо плагінів.

По-друге, основна аналітична логіка винесена в окрему бібліотечну ціль (eng. Library Target). Такий підхід сприяє модульності та тестопридатності. Ця бібліотека інкапсулює класи та методи для парсингу Swift-коду із застосуванням інтегрованого модулю SwiftSyntax [13], реалізацію алгоритмів розрахунку метрик стабільності та зрозумілості, логіку застосування правил та порогових

значень, а також роботу з YAML-конфігурацією через Yams [14]. Виконувана ціль імпортує та використовує цю бібліотеку.

По-третє, для зручності ручного запуску та інтеграції з інструментами командного рядка, пакунок також визначає плагінну ціль (eng. Plugin Target), що реалізує SPM Command Plugin. Цей плагін фактично виступає як зручна обгортка, що зчитує параметри та викликає основну виконувану ціль з відповідними аргументами. Це забезпечує альтернативний шлях використання інструменту для розробників через команди командного рядка або меню Xcode.

Як вже було зазначено, пакунки Swift – це багаторазові компоненти коду, що можуть об'єднувати ресурси, надавати свій код у вигляді двійкових файлів або залежати від інших пакунків. Використання пакунків Swift для компонування виконуваного коду у вигляді виконуваного продукту або бібліотечного продукту є поширеним методом практичної реалізації таких проектів [15]. Пакунки, які надають бібліотечні продукти, сприяють модульності коду, полегшують обмін кодом з іншими користувачами та дають змогу іншим розробникам додавати функціональність до своїх програм.

Таким чином, багатокомпонентна архітектура в рамках єдиного SPM-пакунку Stability Assurance Tool дозволяє поєднати переваги різних підходів: потужність та гнучкість окремого виконуваного файлу для глибокого аналізу, модульність та тестопридатність основної логіки завдяки бібліотеці, та зручність використання через стандартні механізми SPM Command Plugins.

3.3 Оптимізація механізму аналізу та обробки результатів оцінки

Реалізація функціональних можливостей та архітектури аналізатора стабільності коду є лише одієб зі складових процесу створення якісного інструменту. Не менш важливою є його оптимізація для забезпечення практичного застосування в реальних умовах розробки Swift-проектів. Так цей підрозділ присвячено детальному розгляду застосованих методів оптимізації на

різних етапах роботи статичного аналізатора програмного коду Stability Assurance Tool, включаючи оптимізацію процесу парсингу вихідного коду, підвищення алгоритмічної ефективності ядра аналізу, зокрема, через розпаралелювання обчислень там, а також пришвидшення агрегації та фінальної обробки результатів оцінки якості для їх подальшого представлення.

Одним із ключових удосконалених компонентів розробленого інструменту є оптимізований механізм парсингу вихідного коду програмного модуля. У початковій реалізації аналізатор функціонував із прямою залежністю від повного доступу до всіх вихідних файлів, що призводило до передчасного завершення процесу аналізу у випадках виявлення частково або повністю недоступних елементів, зокрема зовнішніх пакетів, модулів або залежностей. Така поведінка значно обмежувала застосування інструменту в умовах розподілених або частково закритих архітектур програмного забезпечення.

Для вирішення цієї проблеми було реалізовано глибшу інтеграцію з бібліотекою SwiftSyntax, що надала змогу виконувати побудову та обхід синтаксичних дерев без необхідності компіляції всього проекту або повного доступу до всіх залежностей. Нова архітектура дозволяє інструменту надійно аналізувати лише доступні сегменти коду, автоматично ідентифікуючи та маркуючи недоступні чи обмежені області як потенційно закриті або зовнішні. При цьому процес оцінки стабільності не переривається, а результати містять відповідні попередження, що підвищує гнучкість та надійність інструменту.

Такий підхід суттєво розширює можливості використання системи для часткової або модульної оцінки стабільності програмного забезпечення, зокрема в умовах розробки великих систем з обмеженим доступом до деяких компонентів або під час аналізу сторонніх бібліотек, інтегрованих на етапі пізнього зв’язування.

Іншою важливою оптимізацією є підвищення алгоритмічної ефективності ядра аналізу. Модифікований компонент складає основу функціональних

можливостей розробленого інструменту, тому якість оптимізації та перевірка регресії були надзвичайно важливими.

Під час аналізу попередньої версії інструменту та виявлених у ній обмежень було встановлено, що всі процеси метричних обчислень виконувались синхронно. Це означало, що розрахунок нової характеристики починався лише після завершення обробки попередньої, що призводило до послідовного блокування потоків обробки. Такий підхід мав певне обґрунтування на етапі початкової розробки: загальна оцінка стабільності програмного коду визначалась як сукупність усіх обчислюваних метрик, деякі з яких повторно використовували результати попередніх обчислень.

Попри відсутність суттєвих затримок у загальному часі виконання під час тестування, аналіз логів демонстрував нерівномірність у завершенні окремих процесів – деякі з них завершувались із помітним запізненням, навіть після того, як необхідні дані вже були доступні. Це вказувало на неефективність обраної моделі в умовах розширення функціональності або збільшення кількості метрик.

У зв’язку з цим було прийнято рішення про паралельну організацію обчислень у рамках статичного аналізу. Зокрема, було виокремлено критичні точки синхронізації: побудова синтаксичного дерева, фіналізація результатів окремих метрик та обчислення загальної оцінки стабільності. Усі інші етапи, включно з обчисленням окремих метрик, що не мають залежності від результатів інших, були переведені в асинхронний режим із використанням структури Task – стандартної одиниці асинхронної роботи у Swift [31]. Такий підхід дозволив істотно підвищити продуктивність інструменту та забезпечити масштабованість у випадку додавання нових метричних показників.

Насамкінець, серед важливих удосконалень слід відзначити оптимізацію процесу обробки фінальних результатів аналізу програмного коду. Внесені зміни до структурної моделі проекту суттєво спростили й пришвидшили обчислення загальної оцінки стабільності в межах системи Stability Assurance Tool. Зокрема, кожну метрику, яка використовувалась як показник стабільності програмного

забезпечення, було винесено в окремий протокол, що, у свою чергу, реалізує інтерфейс ParsableCommand.

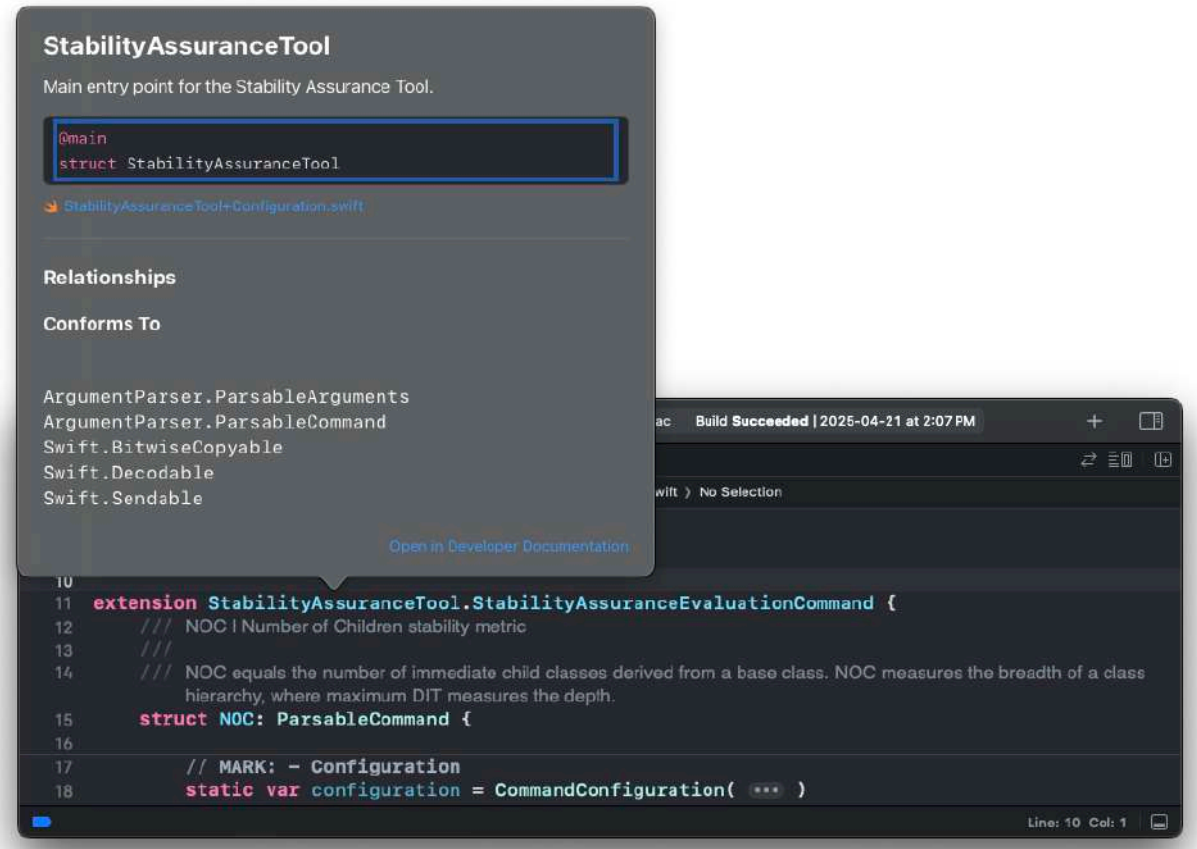


Рис. 3.2 Документація протоколу StabilityAssuranceEvaluationCommand

Завдяки цьому було впроваджено протокольно-орієнтований підхід, який дозволяє пов’язувати схожі за структурою компоненти спільним типом. Це дало змогу забезпечити уніфікований доступ до полів метрик через абстракцію протоколу, зменшити накладні витрати на обробку та зробити механізм масштабованим і придатним до автоматизації. Така архітектура також підвищує гнучкість інструменту в контексті додавання нових метрик та підтримки командного інтерфейсу без значних змін у загальній логіці аналізу.

3.4 Конфігурація критичних рівнів метрик (severity levels) та їх вплив на загальну оцінку

На додаток до гнучкої конфігурації правил та порогових значень метрик за допомогою YAML-файлів, розглянутих у другому розділі цієї роботи, суттєвим компонентом адаптивності інструменту Stability Assurance Tool є механізм налаштування рівнів критичності (eng. severity levels) для кожної інтегрованої метрики оцінки стабільності програмного забезпечення. Цей механізм дозволяє командам розробників не лише інтерпретувати результати аналізу відповідно до внутрішніх вимог, а й автоматизувати ухвалення рішень щодо подальших дій на основі метрик.

Рівні критичності визначаються як інтервали допустимих значень для кожної метрики, що зіставляються з умовно еталонними параметрами. Наприклад, низький рівень зв’язності модуля чи високий рівень зчеплення може бути віднесено до критичного рівня тільки в контексті певного типу архітектури або специфіки системи, яка розробляється. Саме тому інструмент не застосовує жорстко зафіксовані пороги оцінювання. Натомість, за замовчуванням пропонуються рекомендовані діапазони, які можуть бути змінені відповідно до потреб конкретного проекту.

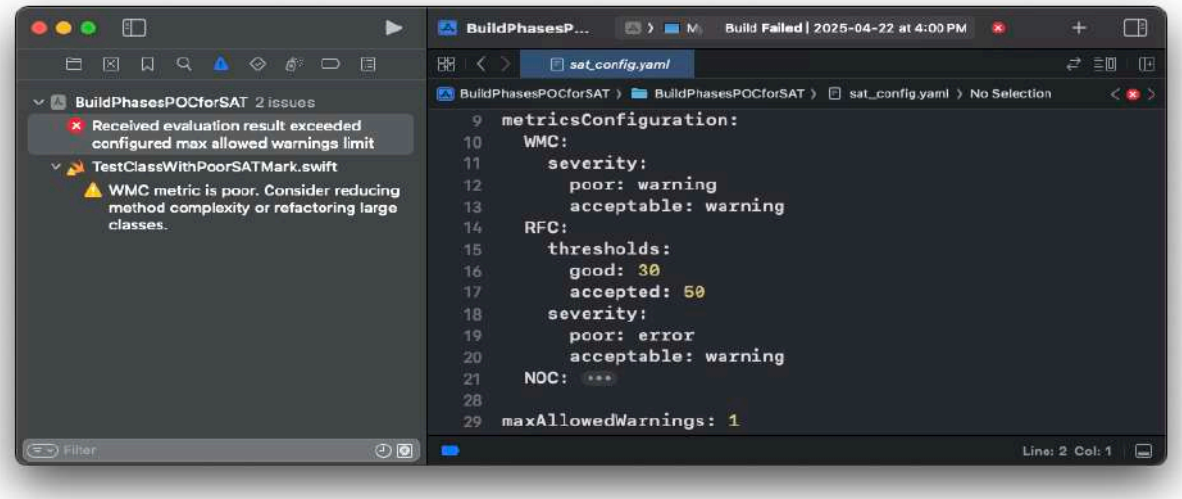


Рис. 3.3 Приклад виводу результатів з заданими рівнями критичності метрик

Додатковою перевагою є підтримка інтегрованої системи реагування на порушення встановлених рівнів критичності. Користувач має можливість самостійно визначити, яку дію повинен ініціювати аналізатор у разі перевищення чи недотримання заданих порогів: від генерації попередження (eng. warning), що не блокує подальший процес аналізу, до фіксації критичної помилки (eng. error), яка призводить до припинення оцінки й сигналізує про необхідність негайного втручання. Такий підхід дозволяє адаптувати інструмент під різні етапи життєвого циклу програмного продукту, як наприклад, встановлювати більш м’які обмеження на етапі прототипування і жорсткіші при підготовці до релізу.

Загалом, механізм рівнів критичності виконує роль своєрідної політики контролю якості, яка забезпечує баланс між гнучкістю аналізу та дотриманням високих стандартів стабільності програмного забезпечення.

3.5 Впровадження загальної оцінки якості в AppDelegate та її інтеграція в CI/CD

Після визначення архітектури статичного аналізатора програмного коду Swift, його оптимізації та налаштування конфігурованих параметрів, постає завдання агрегації всіх зібраних даних в єдиний, осмислений показник якості. Саме загальна оцінка якості коду, яка інтегрує різноманітні метрики стабільності та зрозумілості з урахуванням їхньої ваги та серйозності виявлених проблем, дозволяє приймати обґрунтовані рішення щодо стану проекту та ефективності процесу розробки. У цьому підрозділі детально описано реалізований розрахунок інтегральної оцінки в Stability Assurance Tool та її впровадження в процеси розробки, зокрема через інтеграцію із середовищем розробки Xcode, централізований вивід результатів перевірки в AppDelegate та інтеграцію з CI/CD.

У контексті iOS застосунків, AppDelegate – це основна точка входу до застосунку, яка реалізує обробку глобальних подій життєвого циклу застосунку,

таких як запуск, перехід у фоновий режим або завершення роботи [18]. Саме тому AppDelegate є зручним місцем для ініціалізації загальних служб, зокрема тих, що пов’язані з відстеженням стабільності коду або виводом результатів перевірки в зручному форматі.

Для сприйняття результатів перевірки середовищем розробки Xcode як попередження або помилки, Stability Assurance Tool форматує вивід у спеціальному форматі, що відповідає синтаксису повідомлень компілятора. Зокрема, повідомлення мають вигляд:

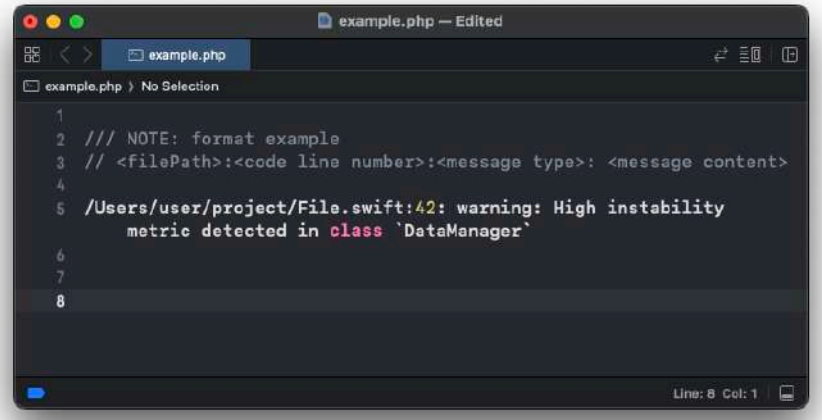


Рис. 3.4 Приклад форматування для інтеграції з інтерфейсом Xcode

Такий формат забезпечує автоматичну інтеграцію з інтерфейсом Xcode. Повідомлення з’являються в секції Issue Navigator, дозволяють переходити до відповідного рядка коду та позначаються жовтими або червоними маркерами, відповідно до рівня критичності [19].

Враховуючи особливості побудови архітектури вдосконаленої версії Stability Assurance Tool, а саме відкритий доступ до збірки пакунку за версією інструменту, наразі користувач має змогу використання цього файлу безпосередньо в процесах розроблюваного програмного забезпечення. Так, розробники мають можливість спрощеного процесу запуску перевірки

стабільності програмного забезпечення через виклик команд аналізатора разом з посиланням на вбудований виконуваний файл у скрипті Build Phases проекту.

Таким чином, оцінка стабільності автоматично відбувається під час збірки проекту. А за рахунок конфігурованих рівнів критичності також регулюється дозвіл на поширення програмного забезпечення за допомогою CI/CD систем, що вимагають попередню збірку проекту.

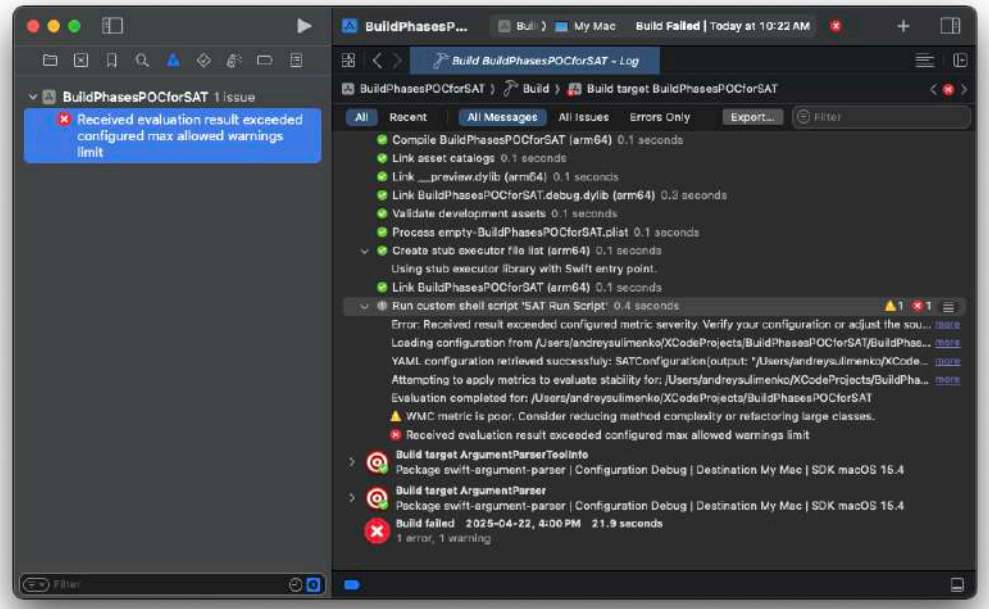


Рис. 3.5 Приклад зупинки виконання програми через перебільшення рівня критичності для метрик Stability Assurance Tool

Не менш важливими атрибутами агрегації Stability Assurance Tool є централізований вивід загальної оцінки в AppDelegate та підтримка формату виводу, сумісного з Xcode. Так користувачі розробленого інструменту мають можливість отримати оброблені результати у звичному вигляді, що автоматично розпізнається як стандартне повідомлення про попередження, значно покращуючи інтеграцію інструменту у щоденний робочий процес розробника.

3.6 Можливості до вдосконалення

Попри досягнуту функціональну повноту реалізації Stability Assurance Tool на поточному етапі, кожен із розглянутих у цьому розділі компонентів зберігає потенціал до подальшого вдосконалення. Зокрема, формалізований підхід до оцінювання стабільності коду може бути доповнено новими сценаріями використання, що охоплюють ширший спектр типів проектів та архітектур.

Одним із ключових компонентів аналізатора програмного коду, що демонструє значний потенціал для подальшого розвитку, є його архітектурний дизайн. Поточна архітектура інструменту вже враховує можливість масштабування, однак водночас створює умови для впровадження глибшої модульності. Це, у свою чергу, відкриває шлях до гнучкої заміни або розширення окремих елементів системи відповідно до потреб конкретного проекту чи середовища використання.

Серед перспективних напрямків вдосконалення архітектури можна виділити реалізацію адаптерів для підтримки різних форматів вхідних даних, що забезпечить ширшу сумісність з різними середовищами розробки. Також доцільним виглядає інтегрування конфігурованих модулів, які застосовують додаткові методики оцінювання окремих характеристик програмного забезпечення – таких як читабельність, відповідність архітектурним патернам або складність модульних залежностей.

Іншим важливим вектором розвитку є покращення інтерактивної взаємодії з розробником безпосередньо в середовищі розробки Xcode. Можливим кроком є створення додаткового розширення редактора коду, яке б доповнювало існуючий механізм зворотного зв'язку через Build Phases. Таке розширення могло б надавати контекстно-залежні команди, наприклад, для автоматичного застосування рекомендованих виправлень (eng. quick fixes) для певних типів виявлених проблем або для запуску аналізу лише виділеного фрагменту коду на вимогу, що прискорило б локальний цикл розробки та виправлення помилок.

Описані можливості до вдосконалення формують лише базову структуру для подальшого розвитку інструменту, яка не обмежується визначеними характеристиками. Так, узагальнюючи викладене, можна стверджувати, що аналізатор програмного коду має значний потенціал для еволюції як на архітектурному, так і на функціональному рівнях. Запропоновані напрямки покращення, як: модульність, адаптивність до різних форматів вхідних даних, розширення методик оцінювання та гнучке налаштування параметрів – сприяють підвищенню універсальності інструменту та його здатності до інтеграції у сучасні процеси розробки. Такий підхід забезпечує стійку основу для подальшої адаптації системи до змінних вимог середовища розробки та очікувань користувачів.

Висновки

У результаті виконання цієї роботи було поглиблено дослідження та розширено функціональні можливості раніше реалізованого інструменту для автоматизованого аналізу стабільності програмного коду. Проведено обґрунтовану модернізацію архітектурних та функціональних компонентів системи з урахуванням сучасних вимог до якості програмного забезпечення. Запропоновані технічні рішення базуються на адаптації та практичному застосуванні ключових метрик об'єктно-орієнтованого дизайну, а також методів статичного аналізу коду.

У межах дослідження було також проаналізовано наявні підходи до оцінювання стабільності та зрозумілості програмного коду, визначено їх сильні сторони й обмеження. Окрему увагу приділено специфіці аналізу проектів, написаних мовою Swift, включно з особливостями роботи компілятора, середовища розробки Xcode та інтеграції інструменту в CI/CD-процеси. Значне місце відведено модульності системи, налаштуванню вагових коефіцієнтів метрик, формуванню YAML-конфігурацій та реалізації гнучкого механізму обробки результатів.

Запропоновані в роботі підходи сприяють глибшому розумінню проблематики аналізу архітектури ПЗ та закладають підґрунтя для подальшого розвитку інструменту з урахуванням актуальних тенденцій у галузі розробки програмного забезпечення.

Незважаючи на суттєві досягнення, описані в межах цієї роботи щодо реалізації, оптимізації та практичного впровадження аналізатора стабільності програмного коду Stability Assurance Tool, залишається очевидним, що навіть найефективніші інструменти потребують постійного вдосконалення. Особливо це актуально для сфери статичного аналізу коду, яка є однією з найбільш динамічних і стрімко зростаючих галузей у сучасному програмному забезпеченні. Зміни в парадигмах розробки, поява нових архітектур, бібліотек та

мовних конструкцій постійно породжують нові виклики, що, своєю чергою, формують потребу у гнучких, масштабованих та інтелектуально адаптивних інструментах аналізу. У свою чергу, глибоке розуміння поточних технічних рішень, їх обмежень та переваг, отримане в результаті цього дослідження, створює надійну основу для окреслення векторів подальшого розвитку Stability Assurance Tool.

Таким чином, інструмент має всі передумови для еволюції в повноцінну екосистему оцінки якості програмного забезпечення, здатну надійно підтримувати процес розробки в умовах зростаючої складності та масштабності сучасних програмних систем.

Список використаних джерел

1. ISO/IEC. (2001). Software engineering – Product quality – Part 1: Quality model. ISO/IEC 9126-1:2001.
2. Apple Inc. (n.d.). Xcode Overview. Apple Developer Documentation. Режим доступу до ресурсу: https://developer.apple.com/xcode/
3. Apple Developer Documentation. (n.d.). Xcode Release Notes. Режим доступу до ресурсу: https://developer.apple.com/documentation/xcode-release-notes
4. Aho, A. V., Lam, M. S., Sethi, R., & Ullman, J. D. (2007). Compilers: Principles, Techniques, & Tools (2nd ed.). Addison-Wesley.
5. Allen, F. E. (1970). Control flow analysis. SIGPLAN Notices, 5(7), 1–19
6. SwiftLint Contributors. (n.d.). realm/SwiftLint. GitHub Repository. Режим доступу до ресурсу: https://github.com/realm/SwiftLint
7. GitHub. (n.d.). github/swift-style-guide (Archived). GitHub Repository. Режим доступу до ресурсу: https://github.com/github/swift-style-guide
8. SonarSource. (n.d.). SonarQube. Product Website. Режим доступу до ресурсу: https://www.sonarsource.com/products/sonarqube/
9. The Maple Lab (University of Alberta). (n.d.). themaplelab/swan. GitHub Repository. Режим доступу до ресурсу: https://github.com/themaplelab/swan
10. Chidamber, S. R., & Kemerer, C. F. (1994). A metrics suite for object oriented design. IEEE Transactions on Software Engineering, 20(6), 476–493.
11. Swift.org. (n.d.). Package Manager. Swift Documentation. Режим доступу до ресурсу: https://www.swift.org/documentation/package-manager/
12. Apple. (n.d.). apple/swift-argument-parser. GitHub Repository. Режим доступу до ресурсу: https://github.com/apple/swift-argument-parser
13. Apple. (n.d.). apple/swift-syntax. GitHub Repository. Режим доступу до ресурсу: https://github.com/apple/swift-syntax
14. Simón, J. P. (n.d.). jpsim/Yams. GitHub Repository. Режим доступу до ресурсу: https://github.com/jpsim/Yams
15. Apple Developer Documentation. (n.d.). Creating a standalone Swift package with Xcode.

16. Apple Developer Documentation. (n.d.). Task. Swift Documentation. Режим доступу до ресурсу: https://developer.apple.com/documentation/swift/task
17. YAML.org. (n.d.). YAML Ain’t Markup Language (YAML™) Version 1.2 Specification. Режим доступу до ресурсу: https://yaml.org/spec/1.2.2/
18. Apple Developer Documentation. (n.d.). UIApplicationDelegate. UIKit Documentation. Режим доступу до ресурсу: https://developer.apple.com/documentation/uikit/uiapplicationdelegate
19. Apple Developer Documentation. (n.d.). Running custom scripts during a build. Xcode Documentation. Режим доступу до ресурсу: https://developer.apple.com/documentation/xcode/running-custom-scripts-during-a-build
20. CocoaPods. (n.d.). CocoaPods.
21. Carthage. (n.d.). Carthage. GitHub Repository. Режим доступу до ресурсу: https://github.com/Carthage/Carthage
22. Apple Developer Documentation. (n.d.). XcodeKit. Режим доступу до ресурсу: https://developer.apple.com/documentation/xcodekit
23. Apple. (n.d.). SE-0303: Package Manager Extensible Build Tools. Swift Evolution Proposal. Режим доступу до ресурсу: https://github.com/apple/swift-evolution/blob/main/proposals/0303-swiftpm-extensible-build-tools.md
24. Apple. (n.d.). SE-0332: Package Manager Command Plugins. Swift Evolution Proposal.
25. Apple Developer Documentation. (n.d.). Running the static analyzer. Режим доступу до ресурсу: https://www.google.com/search?q=https://developer.apple.com/documentation/xcode/running-the-static-analyzer
26. The Swift Programming Language (Swift 5.7). (n.d.). Concurrency. Apple Books.
27. OWASP. (n.d.). Mobile Security Project.
28. Apple Developer Documentation. (n.d.). Testing with Xcode. Режим доступу до ресурсу: https://www.google.com/search?q=https://developer.apple.com/documentation/xcode/testing-with-xcode

29. Apple Developer Documentation. (n.d.). Diagnosing memory, thread, and crash issues quickly. Режим доступу до ресурсу: https://www.google.com/search?q=https://developer.apple.com/documentation/xcode/diagnosing-memory-thread-and-crash-issues-quickly
30. Apple Developer Documentation. (n.d.). MetricKit. Режим доступу до ресурсу: https://developer.apple.com/documentation/metrickit
31. Apple Developer Documentation. (n.d.). Task. Режим доступу до ресурсу: https://developer.apple.com/documentation/swift/task
32. Muchnick, S. S. (1997). Advanced Compiler Design and Implementation. Morgan Kaufmann.