УДК 004.415.5
DOI: 10.18523/2617-3808.2025.8.232-237

Суліменко А. А., Франків О. О., Нагнибіда А. А.

ПРОГРАМНИЙ КОМПЛЕКС STABILITY ASSURANCE TOOL: ЕВОЛЮЦІЯ ТА РОЗВИТОК ДЛЯ АВТОМАТИЗОВАНОЇ ОЦІНКИ СТАБІЛЬНОСТІ ТА ЗРОЗУМІЛОСТІ КОДУ SWIFT

У статті розглянуто процес створення, еволюції та практичного застосування програмного комплексу Stability Assurance Tool (SAT), призначеного для статичного аналізу коду, написаного мовою Swift. Головна мета інструменту полягає у забезпеченні автоматизованої оцінки таких характеристик, як стабільність і зрозумілість програмного забезпечення, що розробляється. Описано використані метрики, архітектурні рішення, методи інтеграції з середовищем розробки Xcode та системами безперервної інтеграції (CI/CD), а також результати адаптації класичних метрик об'єктно-орієнтованого програмування до специфіки Swift. Представлені результати демонструють потенціал SAT як платформи для подальшого розвитку засобів оцінки якості ПЗ.

Ключові слова: SPM, статичний аналіз, стабільність коду, зрозумілість коду, метрики програмного забезпечення, архітектура ПЗ, автоматизована оцінка.

Вступ

Якість програмного забезпечення є ключовим фактором його довговічності, ефективності та здатності до масштабування. У сучасному контексті стрімкого розроблення мобільних і десктопних застосунків, особливо в екосистемі Apple, значну увагу приділяють мовам програмування, що дозволяють створювати безпечні, надійні та зручні у супроводі програми. Однією з таких мов є Swift [1].

Swift є мовою, яка поєднує елементи об'єктно-орієнтованого, протокольно-орієнтованого та функціонального стилів програмування. Це створює як нові можливості, так і виклики для автоматизованого аналізу коду, адже більшість сучасних метрик якості були розроблені з урахуванням класичних об'єктно-орієнтованих мов, таких як Java або C++.

Відповідно, виникає потреба в інструменті, що не лише розуміє синтаксис Swift, а й здатен коректно інтерпретувати його семантику, моделі успадкування, використання протоколів, структури, акторальну модель конкурентності та інші мовні особливості. Саме для задоволення цієї потреби і було створено Stability Assurance Tool.

Теоретичні засади та методи оцінки стабільності програмного забезпечення

Оцінка якості архітектури програмного забезпечення є невід'ємною та фундаментальною складовою процесу управління життєвим циклом будь-якого програмного продукту. Відповідно до визнаних міжнародних стандартів, зокрема ISO/IEC 9126, якість програмного забезпечення визначається через багатогранний набір характеристик [6], таких як функціональність, надійність, зручність використання, ефективність, супроводжуваність і портативність. У контексті цього дослідження центральне місце посідає характеристика супроводжуваності (Maintainability), а саме її ключові підхарактеристики: змінюваність, тестованість, зрозумілість і стабільність. Стабільність, у цьому визначенні, трактується як здатність програмної системи зберігати свою цілісність і коректність функціонування при внесенні модифікацій, що є критично важливим для довгострокового розвитку та еволюції проєкту.

Для кількісної та об'єктивної оцінки цих атрибутів у програмній інженерії застосовуються метрики програмного забезпечення — стандартизовані методи вимірювання, що дозволяють формалізувати та оцінити якість чи складність коду.

Таким чином, методологічною основою для розробленого інструменту Stability Assurance Tool (SAT) став фундаментальний набір об'єктно-орієнтованих метрик, запропонований у праці S. R. Chidamber і C. F. Kemerer (надалі — C&K) [4]. Цей набір, що містить такі відомі показники, як WMC, DIT, NOC, CBO, RFC та LCOM, довів свою ефективність і широко цитується в більшості сучасних досліджень, присвячених аналізу якості коду [7]. У своїй роботі C&K не лише запропонували, а й теоретично та емпірично валідували цей набір метрик для вимірювання якості дизайну об'єктно-орієнтованого програмного забезпечення.

Однак, як було зазначено у пізніших дослідженнях, зокрема у праці L. Etzkorn, C. Davis і J. Talburt, класичний підхід C&K не завжди повною мірою враховує динамічність сучасного процесу розроблення, де навіть незначні структурні зміни можуть суттєво вплинути на результати вимірювань. Цей фактор, у поєднанні з унікальними особливостями мови програмування Swift, такими як її мультипарадигменість, активне використання протокольно-орієнтованого підходу та сучасна модель конкурентності на базі акторів, зумовив нагальну необхідність глибокої адаптації класичних метрик. Відповідно, в рамках розробки SAT було здійснено ретельну адаптацію метрик C&K, уточнено алгоритми їх обчислення та розроблено систему гнучких шкал оцінки, що здатні враховувати масштаб і складність конкретного проєкту.

Адаптація метрик і розробка системи оцінки

Процес створення ефективної та об'єктивної методики оцінки стабільності коду, написаного мовою Swift, вимагав глибокого аналізу та адаптації кожної з обраних метрик. Для забезпечення контекстуальної коректності оцінки було впроваджено класифікацію програмних модулів за розміром (малі, середні, великі), що базується на кількості класів у проєкті. Такий підхід є обґрунтованим, оскільки показники зв'язності та складності мають різну вагу та інтерпретацію для проєктів різного масштабу. Нижче детально описано кожну метрику та її реалізацію в інструменті SAT.

WMC (Weighted Methods per Class — Зважені методи на клас)

В оригінальному визначенні ця метрика часто спрощується до простої кількості методів у класі. У розробленій адаптації запропоновано більш глибокий підхід, що фокусується на обчисленні комплексної складності класу. Значення WMC обчислюється як сума складностей (*Ci*) всіх методів (*Mi*), визначених у класі, за формулою:

$$WMC = i = 1 \sum nCi,$$

де *n* — це загальна кількість методів у класі. Якщо всі складності методів вважати рівними одиниці, то WMC дорівнюватиме *n*. У рамках SAT було запропоновано два способи обчислення складності (*Ci*):

1. **unity**: Складність кожного методу приймається за одиницю (*Ci* = 1). У цьому випадку WMC дорівнює кількості методів, що дає базову, кількісну оцінку складності класу.
2. **custom**: Складність методу обчислюється з використанням показника RFC (Response for a Class). Цей підхід дозволяє оцінити клас не лише за кількістю методів, а й за інтенсивністю та складністю його зовнішніх взаємодій. Таким чином, цей тип утилізує метрику RFC для більш точного та контекстуалізованого визначення WMC. Для метрики WMC були встановлені відносні порогові значення: відхилення до 10 % від середнього показника за проєктом вважається добрим результатом, до 15–20 % (залежно від розміру проєкту) — допустимим, тоді як більші значення сигналізують про потенційні проблеми зі складністю та супроводжуваністю класу.

NOC (Number of Children — Кількість кащадків)

Ця метрика дорівнює кількості безпосередніх дочірніх класів, що успадковуються від базового класу. Враховуючи, що мова Swift активно просуває альтернативи глибоким ієрархіям успадкування через використання протоколів та композиції, високе значення NOC може свідчити про надмірну складність і жорсткість архітектури. Для адекватної оцінки було розроблено допустимі відсоткові співвідношення кількості класів з високим NOC залежно від розміру проєкту: 10 % для малих, 30 % для середніх і 50 % для великих проєктів.

RFC (Response for a Class — Реакція на клас)

В оригінальній праці C&K максимальне значення RFC не було чітко визначено. SAT інтерпретує RFC згідно з фундаментальним визначенням авторів: як набір методів, що потенційно можуть бути виконані у відповідь на повідомлення, отримане об'єктом даного класу. Обчислення відбувається у два етапи:

1. Перший етап (прямі виклики):

$$RFC = M + R$$

де M — це кількість методів, визначених у самому класі, а R — кількість унікальних зовнішніх методів, що викликаються безпосередньо методами цього класу.

2. Повна оцінка (рекурсивні виклики):

$$RFC' = M + R'$$

де R' — це повний набір віддалених методів, викликаних рекурсивно по всьому дереву викликів, що виходять з класу. Численні дослідження, зокрема проведені NASA та іншими авторами, підтверджують, що високий показник RFC сильно корелює зі збільшенням щільності помилок та ускладненням тестування [9]. На основі цих даних було розроблено шкалу оцінки: середнє значення RFC до 50 вважається добрим, до 100 — допустимим, а вищі показники свідчать про надмірну складність та зв'язність класу.

LOC (Lines of Code — Розмірність файлів)

У розробленій системі ця метрика слугує не лише для прямої оцінки розміру програмних модулів, а і як важливий зв'язувальний фактор, що використовується для застосування коректних шкал оцінки до інших метрик. Також LOC опосередковано оцінює лаконічність архітектури в межах окреслених модулів.

LOCM (Lack of Cohesion of Methods — Показник недостатньої зчепленості методів)

Імлементація адаптації цієї метрики є важливим доповненням до першої версії аналізатора, в якій її не було реалізовано через складність обчислення та збору необхідних параметрів з архітектури програмного забезпечення.

Зчепленість є однією з ключових характеристик якісного об'єктно-орієнтованого дизайну. Вона відображає ступінь, до якого елементи всередині одного модуля взаємопов'язані та спрямовані на досягнення єдиної, чітко визначеної мети. Класи з високою зчепленістю є більш зрозумілими, легшими в супроводі та повторному використанні, оскільки їхня відповідальність є вузько сфокусованою.

LOCM, як випливає з її назви, вимірює протилежне — відсутність зчепленості. Високе значення LOCM свідчить про те, що клас, імовірно, виконує багато непов'язаних функцій. Його методи можуть бути розділені на групи, кожна з яких працює з власним набором атрибутів. Така фрагментована структура — ознака слабкого дизайну, і зазвичай вказує на необхідність рефакторингу через розбиття класу на декілька менших, більш сфокусованих одиниць [8]. Навпаки, низьке значення LOCM є ознакою сильної зчепленості: методи класу активно використовують спільні атрибути, що свідчить про якісну архітектуру.

З урахуванням синтаксичних особливостей мови Swift і ґрунтуючись на формальному визначенні метрики, в інструменті Stability Assurance Tool було реалізовано такий алгоритм обчислення LOCM.

Для класу C з множиною методів:

$$\{M1, M2, \ldots, Mn\}$$

та множиною атрибутів (змінних екземпляра)

$$\{A1, A2, \ldots, Am\}:$$

1. Визначення множин атрибутів, які використовує кожен метод.

Для кожного методу Mi визначається множина Ii — набір атрибутів, які він використовує (для читання або запису).

2. Підрахунок кількості пар методів.

Для кожної унікальної пари (*Mi*, *Mj*), *i* < *j* перевіряється перетин множин *Ii*∩*Ij*. Тоді:

– Якщо *Ii* ∩ *Ij* = ∅ — пара вважається незв'язною.
– Якщо *Ii* ∩ *Ij* ≠ ∅ — пара вважається зв'язною.

3. Обчислення LOCM.

$$LOCM = \{P - Q, 0, \text{якщо } P > Q \text{ інакше},$$

де *P* — кількість унікальних незв'язних пар методів, *Q* — кількість зв'язних пар.

Такий обрахунок фактично формалізує ступінь недостатньої зчепленості: якщо кількість незв'язаних пар перевищує кількість зв'язаних, LOCM буде позитивним і сигналізуватиме про проблему. Якщо ж навпаки, то значення LOCM буде нульовим — що вказує на належну внутрішню зв'язність.

Відповідно, інтеграція метрики LOCM у Stability Assurance Tool дозволила суттєво підвищити якість архітектурного аналізу класів, забезпечивши об'єктивний показник ступеня відповідності принципу єдиної відповідальності. LOCM слугує ефективним індикатором внутрішньої зчепленості та підґрунтям для виявлення архітектурних дефектів у дизайні об'єктно-орієнтованих систем.

Архітектура та еволюція комплексу

Архітектура Stability Assurance Tool пройшла кілька фаз трансформації: від простої CLI-програми до повноцінного багатокомпонентного інструменту, що інтегрується у виробничі процеси розроблення програмного забезпечення. Початкова версія SAT була реалізована у формі консольного застосунку, що виконував аналіз коду на основі синтаксичного дерева, сформованого за допомогою бібліотеки SwiftSyntax [2]. У ній реалізовувався обмежений набір метрик, а результати аналізу виводилися у формі звіту в терміналі або у форматі HTML.

У процесі розвитку проєкту було прийнято рішення перейти до модульної архітектури на основі Swift Package Manager. Це дозволило забезпечити розділення відповідальностей між компонентами: виконуваний модуль, бібліотека з логікою аналізу, плагін для зручного запуску з командного рядка. Однією з найважливіших переваг нового підходу стало те, що SAT може інтегруватися безпосередньо в Xcode за допомогою Build Phase Scripts [1]. Завдяки цьому аналіз коду виконується автоматично під час компіляції проєкту, а результати виводяться безпосередньо в Issue Navigator.

Ключовою зміною в архітектурі стала також підтримка паралельного обчислення метрик. Застосування механізмів Swift Concurrency дало можливість розпаралелити обчислення незалежних метрик, що значно зменшило час аналізу для великих кодових баз [10]. Використання структур Task зробило можливим незалежне виконання аналізу для кожного класу, модуля або метрики, що покращило масштабованість і продуктивність SAT.

Практичне застосування

Практичне використання Stability Assurance Tool охоплює як локальну оцінку проєктів під час розроблення, так і автоматизовану перевірку в процесах CI/CD. Інтеграція в Xcode відбувається за рахунок виконуваного скрипта у Build Phases, який викликає аналізатор перед компіляцією цільового модуля. Завдяки форматуванню виводу, що відповідає Xcode-стандартам, усі попередження та помилки автоматично відображаються в редакторі коду. Це забезпечує негайний зворотний зв'язок, дозволяє розробнику швидко виявити проблемні місця та своєчасно їх усунути.

Система конфігурації SAT основана на YAML-файлах, що дозволяє детально налаштовувати параметри аналізу. У YAML-налаштуваннях визначаються порогові значення для кожної метрики, їхні критичність, вагові коефіцієнти, активовані метрики, а також каталоги, які необхідно проаналізувати або вилучити. Такий підхід забезпечує гнучкість та адаптивність, дозволяючи застосовувати SAT до різноманітних проєктів — від невеликих open source-бібліотек до корпоративних рішень із сотнями тисяч рядків коду.

Особливо цінною є інтеграція з CI/CD-середовищами [3]. Аналізатор підтримує параметри типу maxAllowedWarnings, а також дозволяє призначати кожній метриці статус error чи warning при перевищенні порогу. Це забезпечує створення так званих quality gates, які автоматично зупиняють збірку у випадку погіршення якості коду. Таким чином, SAT виступає не лише інструментом діагностики, а й важливим елементом інфраструктури забезпечення якості програмного забезпечення.

Результати та обмеження розробленої системи

У процесі експлуатації SAT було підтверджено його ефективність у виявленні архітектурних недоліків і складних для підтримки модулів. Завдяки використанню кількісних метрик розробники отримують об'єктивну оцінку стабільності та зрозумілості коду, що дозволяє приймати обґрунтовані рішення щодо рефакторингу та покращення архітектури [5]. Інструмент продемонстрував здатність інтегруватися в реальні робочі процеси розробників, не порушуючи звичного циклу роботи.

Однак, попри всі переваги, SAT має і певні обмеження. Зокрема, поточна реалізація не підтримує глибокий аналіз конкурентного коду, зокрема взаємодії між акторами або асинхронних залежностей. Також метрики були адаптовані для класів, однак у Swift велике значення мають протоколи, структури та розширення, які не завжди охоплюються наявними правилами. У майбутньому SAT потребуватиме розширення метрик, орієнтованих саме на особливості Swift — як-от аналіз POP-конструкцій або акторальних сценаріїв.

Ще однією з проблем є продуктивність на надвеликих проєктах. Хоча паралелізація дозволяє прискорити аналіз, все ж час оброблення десятків тисяч класів або модулів може бути значним. Це створює потребу у додатковій оптимізації алгоритмів та побудові більш гнучких стратегій агрегації результатів.

Висновки

Stability Assurance Tool став важливою ініціативою у напрямі побудови надійної інфраструктури контролю якості Swift-проєктів. Його розвиток засвідчив можливість адаптації класичних метрик до сучасних мов програмування, а також продемонстрував ефективність глибокої інтеграції аналізу в процес розроблення. Архітектура SAT дозволяє зручно розширювати інструмент, додаючи нові метрики або вдосконалюючи наявні алгоритми.

Найперспективнішими напрямами подальшого розвитку є розширення підтримки для аналізу Swift Concurrency, зокрема виявлення потенційних блокувань і станів гонитви. Також доцільно розширити метрики для аналізу протокольно-орієнтованої архітектури, зокрема врахування розширень, композитних протоколів і реалізацій через розширення.

Не менш важливим є покращення взаємодії SAT з іншими інструментами. Розробка експортера у форматі SARIF дозволить інтегрувати результати SAT у загальні системи контролю якості, такі як GitHub Act, Azure DevOps або Jenkins. Підтримка Objective-C і змішаних Swift/ObjC-проєктів також відкриє можливості для аналізу складних, спадкових кодових баз.

У перспективі SAT має потенціал перетворитися не лише на інструмент перевірки, а й на рекомендаційний механізм, що не лише виявляє проблеми, а й пропонує шляхи їх вирішення. Залучення елементів машинного навчання та розроблення моделей, здатних на основі історичних даних прогнозувати стабільність змін, відкриє нові горизонти в автоматизованій оцінці якості коду.

Список літератури

1. Apple Inc. Swift Language Guide [Electronic resource]. — 2023. — Mode of access: https://docs.swift.org/swift-book.
2. Apple Inc. SwiftSyntax Documentation [Electronic resource]. — 2023. — Mode of access: https://github.com/apple/swift-syntax.
3. Beller M. How developers use static analysis tools in practice / M. Beller, G. Gousios, A. Zaidman, A. Van Deursen // Proceedings of the 37th International Conference on Software Engineering (ICSE). — IEEE, 2015. — Pp. 191–201.
4. Chidamber S. R. A Metrics Suite for Object-Oriented Design / S. R. Chidamber, C. F. Kemerer // IEEE Transactions on Software Engineering. — 1994. — Vol. 20, no. 6. — Pp. 476–493.
5. Fowler M. Refactoring: Improving the Design of Existing Code / M. Fowler. — Boston : Addison-Wesley, 2002. — 431 p.
6. ISO/IEC 9126-1:2001. Software Engineering — Product Quality. — Part 1: Quality Model. — Geneva : ISO, 2001.
7. Lanza M. Object-Oriented Metrics in Practice / M. Lanza, R. Marinescu. — Berlin : Springer, 2006. — 208 p.
8. Martin R. C. Clean Code: A Handbook of Agile Software Craftsmanship / R. C. Martin. — Upper Saddle River, NJ : Prentice Hall, 2009. — 464 p.
9. NASA Software Assurance Technology Center. Software Quality Metrics Overview [Electronic resource]. — 2003. — Mode of access: https://ntrs.nasa.gov.
10. Swift.org. Concurrency in Swift: Structured Concurrency, Actors, and Async/Await [Electronic resource]. — 2023. — Mode of access: https://www.swift.org.

References

Apple Inc. (2023). Swift Language Guide. https://docs.swift.org/swift-book.

Apple Inc. (2023). SwiftSyntax Documentation. https://github.com/apple/swift-syntax.

Beller, M., Gousios, G., Zaidman, A., & Van Deursen, A. (2015). How developers use static analysis tools in practice. In *Proceedings of the 37th International Conference on Software Engineering* (pp. 191–201). IEEE.

Chidamber, S. R., & Kemerer, C. F. (1994). A metrics suite for object-oriented design. *IEEE Transactions on Software Engineering*, *20* (6), 476–493.
Fowler, M. (2002). *Refactoring: Improving the design of existing code*. Addison-Wesley.
ISO/IEC. (2001). Software engineering — Product quality — Part 1: Quality model (ISO/IEC 9126-1:2001).
Lanza, M., & Marinescu, R. (2006). *Object-oriented metrics in practice*. Springer.
Martin, R. C. (2009). *Clean code: A handbook of agile software craftsmanship*. Prentice Hall.
NASA Software Assurance Technology Center. (2003). Software quality metrics overview. https://ntrs.nasa.gov.
Swift.org. (2023). Concurrency in Swift: Structured concurrency, actors, and async/await. https://www.swift.org.

A. Sulimenko, O. Frankiv, A. Nagnybida

SOFTWARE PACKAGE STABILITY ASSURANCE TOOL: EVOLUTION AND DEVELOPMENT FOR AUTOMATED EVALUATION OF SWIFT CODE STABILITY AND READABILITY

The development of robust and maintainable software systems is highly dependent on the architectural quality of source code. Swift, as a modern programming language developed by Apple, introduces unique challenges for static analysis due to its use of protocol-oriented and concurrent programming paradigms. Traditional quality metrics often fail to capture the nuanced characteristics of Swift codebases, creating a need for dedicated tooling.

This article presents the Stability Assurance Tool (SAT), a lightweight yet powerful static analysis system designed specifically for Swift. SAT applies a suite of object-oriented design metrics, such as those from the Chidamber & Kemerer framework, and adapts them for Swift using techniques based on abstract syntax trees generated via SwiftSyntax. The tool is engineered as a modular Swift package that integrates seamlessly with Xcode and continuous integration systems, providing developers with real-time feedback about the architectural soundness of their code.

The tool analyzes source code by computing metrics in parallel using Swift Concurrency. The results can be visualized via terminal reports or HTML dashboards and serve as input for quality gates in CI/CD pipelines. Special attention is given to YAML-based configuration that allows teams to calibrate metric weights, set custom thresholds, and fine-tune severity levels.

This article also explores SAT's extensible architecture and the practical results of its application to both open-source and enterprise Swift projects. Limitations of the current version include shallow semantic analysis and limited support for concurrency and protocol extensions, which will be addressed in future updates. Long-term plans include integration with SARIF, multi-language support, and machine learning-based prediction of unstable modules.

Keywords: static code analysis, Swift, software stability, code maintainability, CI/CD, SwiftSyntax, software metrics.

Матеріал надійшов 30.06.2025