НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «КИЄВО-МОГИЛЯНСЬКА АКАДЕМІЯ»
МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «КИЄВО-МОГИЛЯНСЬКА АКАДЕМІЯ»
МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

Кваліфікаційна наукова
праця на правах рукопису

ФРАНКІВ ОЛЕКСАНДР ОЛЕКСАНДРОВИЧ

УДК 004.415.5(043.5)

ДИСЕРТАЦІЯ

АВТОМАТИЗОВАНИЙ АНАЛІЗ ТА ВІЗУАЛІЗАЦІЯ В ДОПОВНЕНІЙ РЕАЛЬНОСТІ АРХІТЕКТУРИ ПРОГРАМНИХ МОДУЛІВ ДЛЯ ВИЯВЛЕННЯ АНТИПАТЕРНІВ

122 «Комп’ютерні науки»
12 «Інформаційні технології»

Подається на здобуття наукового ступеня доктора філософії
Дисертація містить результати власних досліджень. Використання ідей, результатів і текстів інших авторів мають посилання на відповідне джерело

_____ О. О. Франків

Науковий керівник **Глибовець Микола Миколайович**, доктор фізико-математичних наук, професор

Київ – 2026

АНОТАЦІЯ

Франків О. О. Автоматизований аналіз та візуалізація в доповненій реальності архітектури програмних модулів для виявлення антипатернів. – Кваліфікаційна наукова праця на правах рукопису.

Дисертація на здобуття наукового ступеня доктора філософії за спеціальністю 122 «Комп'ютерні науки» (12 — Інформаційні технології). — Національний університет «Києво-Могилянська академія», Київ, 2026.

Дисертаційна робота присвячена розв'язанню актуальної науково-прикладної задачі аналізу архітектури програмного забезпечення на основі автоматизованої побудови графової моделі програмного модуля та її подальшої візуалізації. Основною метою роботи є розроблення підходу, який дозволяє формалізувати структуру програмної системи, виконувати аналіз її архітектури та виявляти потенційні проблеми проєктування виключно на основі вихідного коду без необхідності використання додаткової документації, UML-діаграм або ручного опису архітектури.

На сьогодні одним із важливих викликів у галузі розроблення програмного забезпечення є забезпечення належної якості програмних систем та підтримка їх архітектурної цілісності в процесі розвитку проєкту. Незважаючи на значну кількість наявних засобів автоматизованого аналізу програмного коду, більшість із них зосереджені переважно на виявленні локальних синтаксичних або семантичних помилок. Водночас аналіз загальної структури програмної системи, взаємодії її компонентів та оцінювання архітектурних рішень часто залишаються складними задачами, що потребують значних ручних зусиль з боку розробників. Додатковою проблемою є обмежені можливості наявних інструментів щодо представлення результатів аналізу у формі, зручній для сприйняття людиною. У складних програмних системах текстові звіти або набори окремих метрик нерідко є недостатніми для формування цілісного уявлення про структуру архітектури та локалізацію потенційних проблем.

У дисертації проведено аналіз сучасних підходів до оцінювання архітектури програмного забезпечення, зокрема методів статичного аналізу коду, використання програмних метрик, засобів виявлення архітектурних вад, а також методів візуального представлення програмних систем. Показано, що існуючі підходи часто розглядають окремі характеристики програмного забезпечення ізольовано та недостатньо враховують глобальну структуру залежностей між компонентами системи. Встановлено, що традиційні метрики зв'язності, згуртованості або складності є корисними для оцінювання окремих фрагментів програмного коду, однак не забезпечують достатнього рівня наочності для аналізу складних архітектурних взаємозв'язків.

Запропоновано підхід до представлення архітектури програмного забезпечення у вигляді орієнтованого зваженого графа. У межах запропонованої моделі вершини графа відповідають структурним елементам програмної системи, зокрема типам даних, полям та методам, а ребра — різним видам залежностей між ними. Модель враховує як декларативні зв'язки, такі як належність або наслідування, так і поведінкові взаємодії, зокрема виклики методів, використання типів та доступ до полів. Використання ваг ребер дозволяє відобразити інтенсивність взаємодій між компонентами системи та забезпечує можливість подальшого аналізу структури програмного забезпечення.

Показано, що використання графової моделі дозволяє перейти від текстового представлення вихідного коду до формалізованої структури, придатної для алгоритмічної обробки, аналізу та візуалізації. На відміну від спрощених моделей залежностей, запропонований підхід враховує напрям взаємодії між компонентами та дозволяє аналізувати структуру системи на різних рівнях деталізації. Запропонована модель є достатньо універсальною для адаптації до різних мов програмування.

У роботі розроблено методи автоматизованого аналізу архітектури програмного забезпечення на основі дослідження структури графа. Зокрема, запропоновано підходи до виявлення потенційних архітектурних вад, пов'язаних із порушенням зв'язності, згуртованості компонентів та наявністю циклічних

залежностей. Аналіз виконується без використання додаткової інформації про програмну систему та базується виключно на структурі побудованої графової моделі.

У роботі обґрунтовано використання просторового розміщення вузлів графа та особливості локальної структури залежностей для визначення потенційних проблем архітектури програмного забезпечення. Зокрема, надмірна концентрація зв'язків, ізольовані групи компонентів або аномально велика кількість взаємодій між окремими елементами можуть свідчити про порушення принципів проєктування. Запропонований підхід дозволяє локалізувати проблемні області програмного коду та спростити подальший процес аналізу й рефакторингу програмної системи.

Окрему увагу приділено задачі візуалізації графової моделі архітектури програмного забезпечення. У дисертації розглянуто використання просторового представлення графа як засобу підвищення наочності та зменшення когнітивного навантаження під час аналізу складних залежностей між компонентами програмної системи. Для побудови візуального представлення використано силовий алгоритм розміщення графа, що моделює вершини як систему взаємодіючих частинок. Такий підхід дозволяє автоматично формувати естетично прийнятні конфігурації графа, придатні для подальшого аналізу користувачем.

У роботі ґрунтовно досліджена можливість застосування методів машинного навчання для оптимізації процесу побудови візуального представлення графової моделі. Зокрема, розглянуто використання графових нейронних мереж як допоміжного інструменту покращення початкового розміщення вузлів графа та зменшення обчислювальної складності класичних силових алгоритмів. Запропонований комбінований підхід дозволяє підвищити швидкодію побудови візуалізації без суттєвого погіршення її естетичних характеристик та читабельності.

Показано, що використання методів машинного навчання не може повністю замінити класичні алгоритми візуалізації, а виконує роль допоміжного інструменту оптимізації. Поєднання графових нейронних мереж із силовими алгоритмами

дозволяє враховувати структурні особливості графа та забезпечує адаптивність підходу до різних типів програмних систем.

Практичним результатом дисертаційної роботи є розроблення програмного комплексу A.D.A.R. (Architecture Displayer in Augmented Reality), призначеного для автоматизованої генерації, аналізу та візуалізації архітектури програмного забезпечення. Розроблена система виконує побудову графової моделі програмного модуля, підтримує аналіз архітектурних залежностей та дозволяє відображати структуру програмної системи у тривимірному просторі. Використання просторової візуалізації та засобів доповненої реальності дозволяє спростити сприйняття складних залежностей між компонентами програмного забезпечення на інтуїтивному рівні.

Запропонований підхід може бути використаний у задачах аналізу якості програмного забезпечення, виявлення архітектурних проблем, підтримки процесу супроводу програмних систем, рефакторингу та дослідження структури великих програмних проєктів. Окрім практичного застосування у процесі розроблення програмного забезпечення, результати роботи можуть бути використані у навчальному процесі під час вивчення дисциплін, пов’язаних із архітектурою програмного забезпечення, статичним аналізом коду та методами візуалізації даних.

У результаті проведеного дослідження отримано комплексний підхід до аналізу архітектури програмного забезпечення, який поєднує графове представлення структури програмної системи, автоматизований аналіз залежностей, методи візуалізації та елементи машинного навчання. Запропонований підхід забезпечує можливість більш наочного та ефективного дослідження структури програмних систем і може бути основою для подальшого розвитку засобів автоматизованого аналізу програмного забезпечення.

Дисертаційна робота складається з п'яти розділів.

У першому розділі проведено аналітичний огляд підходів до контролю якості та аналізу архітектури програмного забезпечення, методів візуалізації програмних систем та існуючих інструментів. Виявлено, що жоден із наявних засобів не

забезпечує одночасно статичного аналізу вихідного коду, зваженого графа залежностей, адаптивного виявлення аномалій та тривимірної просторової візуалізації, що обґрунтовує задачу дослідження.

У другому розділі розроблено графову модель архітектури програмного модуля у вигляді орієнтованого зваженого графа з трьома категоріями вузлів і шістьма класами ребер, що охоплюють як декларативні, так і поведінкові залежності між компонентами. Описано метод автоматизованої побудови моделі для мови Swift на основі графа символів SymbolKit та продемонстровано її придатність для виявлення конкретних проблем проєктування.

У третьому розділі розроблено підхід до автоматизованого виявлення архітектурних аномалій на основі статистичного аналізу характеристик графа залежностей. Детектор визначає відхилення відносно типових значень конкретної системи, що забезпечує адаптивність без жорстко заданих порогів. Підхід апробовано на реальному Swift-застосунку із підтвердженням виявлених аномалій аналізом вихідного коду.

У четвертому розділі досліджено метрики якості візуалізації графів та встановлено їхні обмеження з точки зору когнітивного сприйняття. Запропоновано використання графових нейронних мереж як евристики для формування початкового розміщення вершин у силових алгоритмах. На вибірці з 44 Swift-проєктів підтверджено повну стабілізацію результатів та скорочення часу побудови layout до 58 % для графів середнього розміру. Розглянуто інтеграцію графової моделі у середовище доповненої реальності як засіб просторової візуалізації.

У п'ятому розділі описано програмний комплекс A.D.A.R., реалізований як набір модулів Swift Package Manager із рівнем покриття тестами 95 %. Наведено деталі інтеграції в інструментарій розробника через SPM-плагіни та описано застосунок-переглядач з підтримкою багатокористувацьких AR-сесій і механізмами зменшення когнітивного навантаження.

Дослідження проводилося в рамках виконання наукових тем кафедри інформатики Національного університету «Києво-Могилянська академія»: «Моделі та методи машинного навчання в прикладних задачах» (УДК: 004.85,

004.49; 004.056.57, № держреєстрації 0123U102430) та «Аналіз великих об'ємів даних в реальному режимі часу» (УДК: 004.4:061.68, 004.67.775, № держреєстрації 0118U000647).

Ключові слова: архітектура програмного забезпечення, програмне забезпечення, граф залежностей, статичний аналіз, виявлення антипатернів, графова нейронна мережа, доповнена реальність, автоматизована візуалізація архітектури, нейронні мережі, штучний інтелект, обчислювальна складність, діаметр графа, абстрактне синтаксичне дерево, когнітивне навантаження, силовий алгоритм розміщення.

ABSTRACT

Frankiv O. O. Automated Analysis and Augmented Reality Visualization of Software Module Architecture for Anti-Pattern Detection. – Qualification scientific work submitted as a manuscript.

PhD thesis to obtain the degree of Doctor of Philosophy in the Programme Subject Area 122 «Computer Science» (12 — Information Technologies). — National University of Kyiv-Mohyla Academy, Kyiv, 2026.

The dissertation is devoted to solving the relevant scientific and applied problem of software architecture analysis based on the automated construction of a graph model of a software system and its further visualization. The primary objective of the research is the development of an approach that enables formalization of software structure, architectural analysis, and detection of potential design problems using only source code without requiring additional documentation, UML diagrams, or manually created architectural descriptions.

One of the major challenges in modern software engineering is ensuring software quality and maintaining architectural integrity during software evolution. Despite the large number of existing automated code analysis tools, most of them are primarily focused on detecting local syntactic or semantic errors. At the same time, the analysis of

global software structure, interactions between system components, and evaluation of architectural decisions often remain difficult tasks requiring substantial manual effort from developers. Another important issue is the limited ability of existing tools to present analysis results in a form convenient for human perception. In large and complex software systems, textual reports or isolated metric values are often insufficient for building a comprehensive understanding of software architecture and localizing problematic areas of source code.

The dissertation includes an analysis of modern approaches to software architecture evaluation, including static source code analysis methods, software quality metrics, architectural anti-pattern detection techniques, and software architecture visualization methods. It is shown that many existing approaches analyze individual software characteristics in isolation and insufficiently account for the global dependency structure between software components. It has been established that traditional metrics of coupling, cohesion, and complexity are useful for evaluating individual fragments of source code but do not provide sufficient clarity for analyzing complex architectural relationships.

An approach to representing software architecture in the form of a directed weighted graph is proposed. Within the developed model, graph vertices correspond to structural elements of the software system, including data types, fields, and methods, while edges represent different types of dependencies between them. The model considers both declarative relationships, such as ownership or inheritance, and behavioral interactions, including method invocations, type usage, and field access. The use of weighted edges enables representation of interaction intensity between software components and supports further structural analysis of software architecture.

It is demonstrated that the graph model makes it possible to transform the textual representation of source code into a formalized structure suitable for algorithmic processing, analysis, and visualization. Unlike simplified dependency models, the proposed approach considers the direction of interactions between software components and enables architecture analysis at different levels of detail. The proposed model is sufficiently generic to be adapted to different programming languages, while for the Swift programming language it can be directly aligned with actual language constructs.

The dissertation proposes methods for automated software architecture analysis based on investigation of graph structure. In particular, approaches for detecting potential architectural flaws related to high coupling, low cohesion, and cyclic dependencies are developed. The analysis is performed without requiring any additional information about the software system and relies exclusively on the structure of the constructed graph model.

The research demonstrates that the spatial arrangement of graph nodes and the characteristics of local dependency structures can be used as indicators of potential software architecture problems. In particular, excessive concentration of dependencies, isolated groups of components, or abnormally large numbers of interactions between specific elements may indicate violations of software design principles. The proposed approach makes it possible to localize problematic areas of source code and simplify subsequent analysis and refactoring processes.

Special attention is devoted to the problem of software architecture visualization. The dissertation examines the use of spatial graph representation as a means of improving clarity and reducing cognitive load during the analysis of complex dependencies between software components. A force-directed graph layout algorithm is used for constructing the visual representation, modeling graph vertices as a system of interacting particles. Such an approach enables automatic generation of aesthetically acceptable graph configurations suitable for further analysis by users.

The dissertation also investigates the possibility of applying machine learning methods to optimize the graph visualization process. In particular, the use of graph neural networks is considered as an auxiliary tool for improving the initial placement of graph nodes and reducing the computational complexity of classical force-directed algorithms. The proposed combined approach improves visualization performance while preserving readability and aesthetic quality.

It is shown that machine learning methods are not considered a complete replacement for classical visualization algorithms but rather an auxiliary optimization instrument. Combining graph neural networks with force-directed algorithms makes it

possible to account for structural graph characteristics and provides adaptability of the approach for different types of software systems.

The practical result of the dissertation is the development of the A.D.A.R. (Architecture Displayer in Augmented Reality) software system intended for automated generation, analysis, and visualization of software architecture. The developed system performs graph model construction for software modules, supports analysis of architectural dependencies, and enables visualization of software structure in three-dimensional space. The use of spatial visualization and augmented reality technologies simplifies perception of complex relationships between software components at an intuitive level.

The proposed approach can be applied to software quality analysis, architectural flaw detection, maintenance support, refactoring, and investigation of large-scale software project structures. In addition to practical use in software development processes, the results of the research can also be used in education for courses related to software architecture, static code analysis, graph theory, and data visualization methods.

As a result of the conducted research, a comprehensive approach to software architecture analysis has been developed that combines graph-based representation of software structure, automated dependency analysis, visualization methods, and machine learning elements. The proposed approach provides more efficient and intuitive analysis of software systems and may serve as a foundation for further development of automated software analysis tools.

The dissertation consists of five chapters.

The first chapter presents an analytical review of software quality control approaches, software architecture analysis methods, and software system visualization techniques. The analysis reveals that none of the existing tools simultaneously provides static source code analysis, a weighted dependency graph, adaptive anomaly detection, and three-dimensional spatial visualization, which motivates the research problem.

The second chapter develops a graph model of software module architecture as a directed weighted graph with three node categories — data types (dt), properties (p), and methods (m) — and six edge classes covering both declarative and behavioral

dependencies between components. An automated model construction method for Swift based on the SymbolKit compiler symbol graph is described, and the model's applicability to detecting specific design problems is demonstrated.

The third chapter proposes an approach to automated architectural anomaly detection based on statistical analysis of dependency graph characteristics. The detector identifies deviations relative to the typical values within a specific system, ensuring adaptability without hard-coded thresholds. The approach is validated on a real Swift application, with detected anomalies confirmed through source code analysis.

The fourth chapter investigates graph visualization quality metrics and establishes their limitations from a cognitive perception perspective. A graph neural network is proposed as a heuristic for generating initial node placement in force-directed algorithms. On a dataset of 44 Swift projects, complete stabilization of results is confirmed along with layout construction time reduction of up to 58% for medium-sized graphs. The integration of the graph model into an augmented reality environment as a spatial visualization tool is examined.

The fifth chapter describes the A.D.A.R. software system implemented as a set of Swift Package Manager modules with 95% test coverage. Details of developer toolchain integration through SPM plugins and the AR viewer application with multi-user session support and cognitive load reduction mechanisms are presented.

The research was conducted within the scientific topics of the Department of Computer Science of the National University of Kyiv-Mohyla Academy: "Models and Methods of Machine Learning in Applied Problems" (UDC: 004.85, 004.49; 004.056.57, State Registration No. 0123U102430) and "Analysis of Large Data Volumes in Real Time" (UDC: 004.4:061.68, 004.67.775, State Registration No. 0118U000647).

Keywords: software architecture, software, dependency graph, static analysis, anti-pattern detection, graph convolutional network, augmented reality, automated architecture visualization, neural networks, artificial intelligence, computational complexity, graph diameter, abstract syntax tree, cognitive load, force-directed layout.

СПИСОК ПУБЛІКАЦІЙ ЗДОБУВАЧА

Наукові праці, в яких опубліковані основні наукові результати дисертації:

1. Франків, О. (2022). Використання доповненої реальності для візуалізації архітектур програмних модулів. Наукові записки НаУКМА. Комп'ютерні науки, 5, 26—30. https://doi.org/10.18523/2617-3808.2022.5.26-30
2. Франків, О., & Глибовець, М. (2024). Автоматизована візуалізація компонентів архітектури програми для мови Swift. Кібернетикаа та системний аналіз, 190—198. https://doi.org/10.34229/KCA2522-9664.24.6.16
3. Франків, О. (2025). Машинне навчання в покращенні візуалізації просторової моделі архітектури програми. Вісник Київського національного університету імені Тараса Шевченка. Фізико-математичні науки, 80(1), 164—173. https://doi.org/10.17721/1812-5409.2025/1.22
4. Франків, О. О. (2025). Автоматизоване виявлення вад архітектури програмного модуля з використанням графової моделі візуалізації. Наукові Записки НаУКМА. Комп'ютерні Науки, 8, 167—173. https://doi.org/10.18523/2617-3808.2025.8.167-173

Наукові праці, які засвідчують апробацію матеріалів дисертації:

1. Франків, О. О. (2024). Візуалізація архітектур програмних модулів для виявлення вад проєктування. У В. Є. Снитюк (Ред.), Когнітивні дослідження: результати, виклики та перспективи: матеріали Першої всеукраїнської наукової конференції (24 травня 2024 р., Київ, Україна) (с. 321–324). Видавництво «Каравела». ISBN 978-960-801-899-0.
2. Франків, О. О. (2024). Машинне навчання для візуалізації моделей архітектур програм на мобільних пристроях. У М. М. Глибовець & Т. В. Панченко (Ред.), Теоретичні та прикладні аспекти побудови програмних систем: праці 15 міжнародної науково-практичної конференції (23–24 грудня 2024 р., Київ, Україна) (с. 54–56). Видавництво НаУКМА.

3. Франків, О. О. (2025). Вдосконалення візуалізації моделей архітектур програм для зменшення когнітивного навантаження користувача. У М. М. Глибовець & Т. В. Панченко (Ред.), Теоретичні та прикладні аспекти побудови програмних систем: праці 16 міжнародної науково-практичної конференції (24–25 листопада 2025 р., Київ, Україна) (с. 58–60). Видавництво НаУКМА.

Наукові праці, які додатково відображають наукові результати дисертації:

1. Суліменко, А. А., Франків, О. О., & Нагнибіда, А. А. (2025). Програмний комплекс Stability Assurance Tool: Еволюція та розвиток для автоматизованої оцінки стабільності та зрозумілості коду Swift. Наукові записки НаУКМА. Комп'ютерні науки, 8, 232—237. https://doi.org/10.18523/2617-3808.2025.8.232-237

ЗМІСТ

ВСТУП

Обґрунтування вибору теми дослідження. Сучасні програмні системи характеризуються високим рівнем складності, що зумовлено зростанням кількості компонентів, їх взаємозв'язків та еволюційним характером розвитку програмного забезпечення. У таких умовах архітектура програмної системи відіграє ключову роль у забезпеченні її якості, зокрема підтримуваності, масштабованості та надійності.

Аналіз архітектури програмного забезпечення традиційно здійснюється із використанням формальних метрик, статичних методів аналізу та експертної оцінки. Однак зазначені підходи мають ряд обмежень, пов'язаних із недостатньою здатністю відображати структурні особливості системи у цілісному вигляді, а також складністю інтерпретації отриманих результатів.

Зокрема, використання метрик дозволяє оцінювати окремі характеристики програмних компонентів, проте не забезпечує повного уявлення про їх взаємодію у межах архітектури. У свою чергу, традиційні засоби візуалізації, переважно орієнтовані на двовимірне подання графових структур, часто не забезпечують достатнього рівня наочності при аналізі складних систем, що призводить до перевантаження візуального простору та ускладнює сприйняття інформації.

Незважаючи на наявність широкого спектра підходів до аналізу архітектури програмного забезпечення, на практиці зберігається протиріччя між зростаючою складністю програмних систем та можливістю їх ефективного аналізу.

З одного боку, формальні методи, зокрема метрики та статичний аналіз, дозволяють отримувати кількісні оцінки окремих характеристик архітектури. З іншого боку, такі оцінки є фрагментарними і не забезпечують цілісного уявлення про структуру системи та взаємодію її компонентів.

Водночас візуалізація графових моделей архітектури дозволяє представити систему як єдину структуру, однак існуючі підходи до побудови layout не враховують у повній мірі особливості сприйняття інформації користувачем. Як було

показано у попередніх дослідженнях, оптимізація формальних метрик якості візуалізації не гарантує покращення її сприйняття користувачем [1].

Таким чином, виникає розрив між формалізованими методами аналізу архітектури та засобами її візуального подання, що ускладнює виявлення архітектурних антипатернів і прийняття рішень щодо вдосконалення структури програмного забезпечення.

Це обумовлює необхідність розробки підходів, що поєднують формалізований аналіз архітектури з когнітивно орієнтованими засобами візуалізації та забезпечують ефективне виявлення архітектурних антипатернів.

Метою роботи є підвищення ефективності аналізу архітектури програмних модулів шляхом розробки методів і моделей автоматизованого виявлення архітектурних антипатернів на основі графового представлення та використання когнітивно орієнтованих засобів візуалізації у середовищі доповненої реальності.

Для досягнення поставленої мети у роботі вирішені такі **завдання**:

1. Проведено аналіз існуючих підходів до дослідження архітектури програмного забезпечення, зокрема методів формального аналізу та візуалізації, і визначити їх основні обмеження.

2. Розроблено графову модель представлення архітектури програмних модулів, що дозволяє формалізувати структуру залежностей між компонентами системи.

3. Розроблено метод автоматизованого формування графової моделі на основі аналізу вихідного коду програмної системи.

4. Розроблено підхід до виявлення архітектурних антипатернів на основі аналізу структурних характеристик графа залежностей та їх статистичної оцінки.

5. Досліджено метрики якості візуалізації графів та визначити їх обмеження у контексті аналізу архітектури програмного забезпечення.

6. Розроблено підхід до візуалізації графової моделі архітектури із використанням силових алгоритмів з урахуванням особливостей сприйняття інформації користувачем.

7. Розроблено метод використання машинного навчання як евристики для підвищення ефективності та стабільності роботи алгоритмів візуалізації.

8. Розроблено та досліджено підхід до візуалізації архітектури програмного забезпечення у середовищі доповненої реальності з метою зменшення когнітивного навантаження та покращення інтерпретованості результатів.

Формально дослідження може бути уточнено наступним чином. Нехай задано програмний модуль M, вихідний код якого доступний у вигляді набору файлів. Необхідно побудувати функцію аналізу A: M → (G, Λ, $\boldsymbol{l}$), де G = (V, E, W) — орієнтований зважений граф залежностей: V — множина вершин (типи даних, методи, властивості), E⊆V×V — орієнтовані ребра (залежності), W: E → (0; 1] — ваги інтенсивності взаємодій; Λ — множина архітектурних аномалій, виявлених у G на основі статистичного аналізу характеристик вершин та ребер; $\boldsymbol{l}$: G → $\mathbb{R}^3$ — функція візуалізації, що відображає кожну вершину у координати тривимірного простору таким чином, щоб просторова конфігурація графа відображала структуру залежностей. Функція A має бути побудована виключно на основі вихідного коду M, без використання додаткової документації або ручних анотацій [2].

Об’єктом дослідження є архітектура програмного забезпечення як система взаємопов’язаних програмних модулів та процеси її аналізу в ході життєвого циклу програмних систем.

Предметом дослідження є методи, моделі та алгоритми автоматизованого аналізу архітектури програмних модулів і способи її візуалізації в середовищі доповненої реальності, спрямовані на виявлення архітектурних антипатернів.

Методи дослідження. У процесі дослідження використано методи теорії графів, які застосовуються для побудови формалізованої моделі архітектури програмного забезпечення та аналізу структури залежностей між її компонентами. Графове представлення дозволяє розглядати архітектуру як єдину систему взаємопов’язаних елементів.

Для автоматизованого формування графової моделі використано методи статичного аналізу програмного коду, що забезпечують отримання інформації про залежності між компонентами системи без виконання програми.

З метою виявлення архітектурних антипатернів застосовано статистичні методи аналізу даних, які дозволяють визначати відхилення характеристик елементів графа від типових значень та формалізувати критерії аномальності.

Побудова візуального представлення графа здійснюється із використанням методів оптимізації, зокрема силових алгоритмів, що забезпечують формування просторової конфігурації, яка відображає структуру взаємозв'язків між компонентами.

Для підвищення ефективності та стабільності роботи алгоритмів візуалізації використано методи машинного навчання, які застосовуються як евристика для формування початкового розміщення вершин графа та покращення збіжності силових алгоритмів.

Реалізація візуалізації архітектури програмного забезпечення у просторі здійснюється із використанням методів тривимірної комп'ютерної графіки та доповненої реальності, що дозволяє враховувати особливості сприйняття інформації користувачем.

Оцінювання ефективності запропонованого підходу проводиться на основі аналізу кількісних показників та результатів практичного застосування.

Наукова новизна отриманих результатів полягає у наступному:

Вперше:

– запропоновано підхід до автоматизованого аналізу архітектури програмних модулів, що поєднує графове представлення структури системи, статистичні методи виявлення архітектурних антипатернів та когнітивно орієнтовану візуалізацію у середовищі доповненої реальності;

– запропоновано методологію використання методів машинного навчання як евристики для формування початкового розміщення вершин графа архітектури, що підвищує стабільність та ефективність роботи силових алгоритмів візуалізації;

Удосконалено:

– метод автоматизованого формування графової моделі архітектури програмного забезпечення на основі статичного аналізу вихідного коду за рахунок розширення набору врахованих типів залежностей між компонентами;

– підхід до виявлення архітектурних антипатернів, який, на відміну від існуючих, базується на аналізі відносних відхилень характеристик елементів графа, що забезпечує адаптивність до конкретної програмної системи;

Набули подальшого розвитку:

– методи візуалізації графових моделей архітектури програмного забезпечення шляхом врахування когнітивних аспектів сприйняття інформації користувачем;

– підходи до поєднання формалізованого аналізу архітектури програмного забезпечення з інтерактивними засобами візуалізації у тривимірному середовищі.

Особистий внесок здобувача. Усі основні результати дисертаційного дослідження, що виносяться на захист, одержані автором особисто. У публікаціях особистий внесок здобувача полягає в такому: аналіз існуючих підходів до візуалізації архітектур програмних систем, обґрунтування доцільності застосування доповненої реальності, розробка концепції та реалізація підходу до AR-відображення архітектурних залежностей, підготовка тексту (стаття опублікована одноосібно) [осн. 1]; розробка методу автоматизованої побудови графової моделі залежностей Swift-проєктів на основі графа символів компілятора, реалізація програмного прототипу, проведення експериментальних досліджень та підготовка тексту; постановку наукової проблеми здійснено у співавторстві з науковим керівником [осн. 2]; дослідження метрик якості візуалізації графів та їх обмежень, обґрунтування доцільності використання графової нейронної мережі (GCN) як евристики для стабілізації силових алгоритмів розміщення вузлів, верифікація підходу та підготовка тексту (стаття опублікована одноосібно) [осн. 3]; розробка формального підходу до виявлення архітектурних антипатернів на основі статистичного аналізу характеристик графа залежностей, обґрунтування порогових критеріїв, проведення верифікаційних експериментів та підготовка тексту (стаття опублікована одноосібно) [осн. 4]. У публікаціях [апр. 1–3], що засвідчують апробацію матеріалів дисертації, ідея, постановка задачі, реалізація та підготовка тексту та доповідей виконані здобувачем особисто. У публікації, яка додатково відображає наукові результати дисертації [дод. 1], здобувачем особисто запропоновано ідею, визначено вектор дослідження та виконано проєктування

програмного комплексу; реалізацію та частину аналізу результатів виконано співавторами.

Зв’язок роботи з науковими програмами, планами, темами, грантами. Дисертаційне дослідження виконано в рамках наукових тем кафедри інформатики Національного університету «Києво-Могилянська академія»: «Моделі та методи машинного навчання в прикладних задачах» (УДК: 004.85, 004.49; 004.056.57, № держреєстрації 0123U102430) та «Аналіз великих об'ємів даних в реальному режимі часу» (УДК: 004.4:061.68, 004.67.775, № держреєстрації 0118U000647). Результати впровадження підтверджено відповідними актами.

Практичне значення отриманих результатів полягає у можливості застосування запропонованого підходу для автоматизованого аналізу архітектури програмних систем та виявлення архітектурних антипатернів на різних етапах їх розробки та супроводження.

Розроблені методи і моделі можуть бути використані при проведенні архітектурних рев’ю, рефакторингу програмного забезпечення, а також у процесах контролю якості програмних систем, що дозволяє підвищити обґрунтованість прийняття рішень щодо зміни структури програмних модулів.

Запропонований підхід до візуалізації архітектури у середовищі доповненої реальності забезпечує покращення сприйняття складних структур та створює передумови для зменшення когнітивного навантаження, що є важливим при аналізі великих програмних систем.

Результати дослідження можуть бути використані при розробці програмних інструментів підтримки архітектурного аналізу, а також у навчальному процесі підготовки фахівців у галузі програмної інженерії.

Результати дисертаційного дослідження впроваджено у виробничу діяльність ТОВ «Девелопекс» (м. Київ, ЄДРПОУ 32529040) під час аналізу та аудиту архітектури програмних модулів реальних iOS-проєктів компанії: програмний комплекс A.D.A.R. використано для автоматизованого аналізу вихідного коду, побудови графової моделі архітектури та виявлення компонентів з надмірною зв'язністю, низькою згуртованістю та циклічними залежностями, що дало змогу

суттєво прискорити процес аудиту та виявляти потенційно проблемні ділянки коду значно швидше порівняно з ручним аналізом (акт про дослідне впровадження від 10 квітня 2026 р., наведено в Додатку Б). Окремо результати дисертаційного дослідження впроваджуються в освітньому процесі Національного університету «Києво-Могилянська академія» під час підготовки фахівців за освітньо-науковими програмами «Інженерія програмного забезпечення» та «Комп'ютерні науки» факультету інформатики під час викладання дисциплін сертифікатної програми “Програмування для iOS” (акт про впровадження від 22 червня 2026 р., наведено в Додатку В).

Апробація матеріалів дисертації. Основні результати дисертаційної роботи доповідалися та обговорювалися на наукових конференціях і семінарах, зокрема:

- Першій всеукраїнській науковій конференції «Когнітивні дослідження: результати, виклики та перспективи», м. Київ, 2024;
- 15 Міжнародній науково-практичній конференції «Теоретичні та прикладні аспекти побудови програмних систем» (TAAPSD 2024), м. Київ, 2024;
- 16 Міжнародній науково-практичній конференції «Теоретичні та прикладні аспекти побудови програмних систем» (TAAPSD 2025), м. Київ, 2025.

Основні положення та результати дисертаційної роботи відображено у наукових публікаціях автора.

Структура та обсяг дисертації. Дисертаційна робота складається зі вступу, п'яти розділів, висновків, списку використаних джерел та п’яти додатків.

У першому розділі проведено аналіз існуючих підходів до дослідження архітектури програмного забезпечення, зокрема методів формального аналізу та візуалізації, та визначено їх основні обмеження.

У другому розділі розроблено графову модель представлення архітектури програмних модулів та описано метод її автоматизованого формування на основі аналізу вихідного коду.

У третьому розділі запропоновано підхід до виявлення архітектурних антипатернів на основі аналізу структурних характеристик графа залежностей.

У четвертому розділі розглянуто підходи до візуалізації графової моделі архітектури, досліджено метрики якості візуалізації, запропоновано використання методів машинного навчання для підвищення ефективності алгоритмів та обґрунтовано використання доповненої реальності для зменшення когнітивного навантаження.

У п'ятому розділі описано програмний комплекс A.D.A.R.: архітектуру набору модулів Swift Package Manager, механізми інтеграції в середовище розробки через SPM-плагіни, а також деталі реалізації застосунку-переглядача з підтримкою багатокористувацьких сесій доповненої реальності та засобами зменшення когнітивного навантаження.

Додаток А містить список публікацій здобувача. Додатоки Б та В містять акти про впровадження результатів дисертаційної роботи. Додаток Г містить приклади індикації антипатернів у вихідному коді в середовищі розробки Xcode. Додаток Д містить знімки екрану програми-переглядача A.D.A.R., що ілюструють основні режими роботи застосунку.

РОЗДІЛ 1

АНАЛІЗ АРХІТЕКТУРИ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

1.1 Якість програмного модуля

Відповідно до класичного визначення, програмне забезпечення — це комп'ютерні програми, процедури та, можливо, пов'язана документація і дані, що стосуються роботи комп'ютерної системи [3]. Спираючись на це визначення важливо підкреслити складність та багатокомпонентність цього поняття. Сучасне програмне забезпечення, зважаючи на високий рівень складності та обсяг завдань, які воно вирішує, часто є доволі складною системою взаємопов'язаних компонентів.

Поняття програмного модуля може мати кілька значень залежно від контексту. З одного боку програмний модуль — це програмна одиниця, яка є відокремленою та ідентифікованою з погляду компіляції, поєднання з іншими одиницями та завантаження. Програмний модуль можна також визначити як "набір файлів вихідного коду під керуванням системи контролю версій, якими можна керувати разом як єдиним цілим", або ж як "сукупність як даних, так і процедур, що з ними працюють". Найбільш універсальним визначенням цього поняття можна вважати таке: програмний модуль — це "логічно відокремлена частина програми" [3]. Для того, щоб якомога менше залежати від конкретних технічних деталей побудови ПЗ, таких як спосіб зберігання вихідного коду у файлах, компіляції, завантаження, роботи з даними чи наявності систем контролю версій, ми надалі розглядаємо програмний модуль як певну атомарну в технічному плані і логічно незалежну частину програмного забезпечення. Варто підмітити, що багато простих програм можуть мати один модуль.

Якість програмного забезпечення — це здатність програмного продукту задовольняти заявлені та неявні потреби під час використання за визначених умов [3]. Під цим поняттям також можна розуміти ступінь, до якого програмне забезпечення задовольняє вищезгадані потреби чи вимоги. Так само можна

розуміти і поняття якості програмного модуля як частини ПЗ. Хоча спираючись на ці визначення логічно припустити, що якість ПЗ як ступінь відповідності потребам складає виключно загальна якість всіх модулів, що його складають, не менш важливим аспектом є правильність їх поєднання. Слід розуміти, що якість всіх модулів окремо є необхідною, але не достатньою умовою якості ПЗ. Зважаючи на нетривіальність і високу залежність від деталей реалізації проблеми якості поєднання модулів, ми фокусуємось на аналізі якості саме програмних модулів окремо.

У галузі розробки програмного забезпечення ключовим завданням є не лише досягнення заданого рівня якості, але й забезпечення його стабільності протягом усього життєвого циклу програмного продукту. З огляду на складність сучасних програмних систем, у яких поєднуються численні взаємозалежні компоненти, забезпечення якості неможливе виключно шляхом фінальної перевірки результату [4]. Міжнародні стандарти з інженерії програмного забезпечення розглядають контроль якості як систематичну діяльність, інтегровану в процеси розробки, спрямовану на запобігання появі дефектів, своєчасне їх виявлення та мінімізацію негативного впливу на властивості програмного продукту. У практиці розробки такі заходи реалізуються не у вигляді одного універсального механізму, а як сукупність взаємнодоповнюючих підходів до контролю якості, кожен з яких фокусується на окремих аспектах програмного забезпечення або процесу його створення [4], [5], [6].

Процесно-орієнтований підхід до контролю якості програмного забезпечення сфокусовано на організації та дотриманні самого процесу розробки, а не лише на кінцевому результаті. Якість тут розглядається як наслідок систематичного виконання визначених процедур, правил і практик протягом усього життєвого циклу програмного забезпечення. До таких практик належать встановлення та дотримання стандартів і рекомендацій щодо написання коду, регулярна перевірка коду (code review), аудит процесів розробки, формалізація критеріїв готовності програмного продукту (definition of done), а також використання інструментів статичного аналізу. Основною метою процесно-орієнтованого підходу є

запобігання появі дефектів шляхом забезпечення керованості та повторюваності процесів розробки. Вважається, що стабільний і формалізований процес дозволяє зменшити ймовірність систематичних помилок, покращити передбачуваність результатів і забезпечити відтворювану якість програмних модулів незалежно від конкретних виконавців.

Продукт-орієнтований підхід до контролю якості програмного забезпечення сфокусовано на аналізі характеристик уже реалізованого програмного продукту або окремих його модулів. Якість оцінюється шляхом безпосередньої перевірки поведінки програмного модуля відносно заданих вимог, що реалізується за допомогою тестування на різних рівнях, зокрема модульному (unit), інтеграційному, системному та приймальному. Окрім перевірки функціональної коректності, значна увага приділяється нефункціональним властивостям програмного модуля, таким як продуктивність, надійність, безпека та зручність використання. Продукт-орієнтований підхід дозволяє безпосередньо виявляти дефекти реалізації, оцінювати відповідність програмного модуля вимогам і фіксувати фактичний рівень якості на певному етапі життєвого циклу. Водночас цей підхід, як правило, не дає змоги встановити причини виникнення дефектів у процесі розробки, а тому є найбільш ефективним у поєднанні з процесно-орієнтованими та запобіжними підходами до забезпечення якості.

Запобіжний підхід до контролю якості програмного забезпечення, відомий також як Shift-Left, ґрунтується на перенесенні заходів із забезпечення якості на якомога ранні етапи життєвого циклу розробки. Основна ідея цього підходу полягає в тому, що виявлення та усунення дефектів на етапах формування вимог, проєктування та архітектурного аналізу є значно ефективнішим і менш затратним, ніж їх виправлення після реалізації або під час експлуатації. У межах запобіжного підходу застосовуються такі практики, як розробка через тестування (Test-Driven Development), поведінково-орієнтована розробка (Behavior-Driven Development), прототипування, формалізація вимог у поєднанні з тест-кейсами, проведення архітектурних перевірок та моделювання загроз. Запобіжний підхід дозволяє зменшити кількість дефектів, що переходять на пізніші стадії розробки, підвищити

узгодженість між вимогами та реалізацією, а також покращити загальну керованість якості програмних модулів, особливо в складних і довготривалих програмних системах.

Безперервний підхід до контролю якості програмного забезпечення передбачає постійне оцінювання якості програмних модулів у процесі їх розробки та супроводу. Цей підхід тісно пов'язаний із практиками безперервної інтеграції та доставки (CI/CD) і базується на автоматизованому виконанні перевірок якості при кожній зміні програмного коду — зокрема автоматизовані збірки, виконання тестів, статичний аналіз, перевірка відповідності пороговим значенням метрик якості (quality gates), а також регулярні інтеграційні або нічні збірки. Безперервний контроль якості забезпечує швидкий зворотний зв'язок для розробників, дозволяє оперативно виявляти регресії та деградацію якості програмних модулів, а також підтримує стабільний рівень якості в умовах частих змін і ітеративної розробки [7], [8], [9].

Підхід, заснований на метриках та вимірюванні, розглядає якість програмного забезпечення як характеристику, що може бути кількісно оцінена за допомогою формалізованих показників. Для аналізу якості використовуються метрики, зокрема щільність дефектів, середній час відновлення після відмов (MTTR), покриття тестами, цикломатична складність, інтенсивність змін коду та кількість дефектів, виявлених після релізу. Застосування метрик дозволяє порівнювати якість різних модулів, відстежувати динаміку змін у часі та приймати обґрунтовані рішення щодо подальшого розвитку програмного забезпечення. Водночас важливим аспектом цього підходу є усвідомлення обмежень окремих метрик, оскільки жоден числовий показник не може повністю описати якість програмного модуля без урахування контексту його використання [10], [11], [12], [13].

Архітектурно-орієнтований підхід до контролю якості програмного забезпечення зосереджується на аналізі структури програмних систем і взаємодії між їхніми модулями. Якість розглядається через призму архітектурних рішень, залежностей між компонентами та їх впливу на підтримуваність, масштабованість

і безпеку програмного забезпечення. Основними інструментами архітектурного контролю є архітектурні рев'ю, документування архітектурних рішень (ADR), аналіз залежностей, а також автоматизовані перевірки безпеки, зокрема статичний і динамічний аналіз вразливостей. Архітектурний підхід дозволяє виявляти системні проблеми, які не проявляються на рівні окремих тестів, але можуть суттєво впливати на довгострокову якість програмних модулів [14], [15], [16].

Користувацько-орієнтований підхід до контролю якості програмного забезпечення базується на оцінюванні властивостей програмного продукту з позиції кінцевого користувача. Якість програмних модулів визначається через зручність використання, надійність під час реальної експлуатації та відповідність очікуванням користувачів. Для цього застосовуються методи тестування зручності використання, бета-тестування, аналіз телеметрії та звітів про збої, а також експериментальні методи, зокрема A/B тестування. Користувацько-орієнтований підхід дозволяє виявляти проблеми, які не завжди проявляються під час формального тестування, але безпосередньо впливають на сприйняття якості програмного забезпечення та ефективність його використання.

Стандартизований підхід до контролю якості програмного забезпечення ґрунтується на застосуванні формальних моделей і міжнародних стандартів, які визначають вимоги до якості програмних продуктів і процесів їх розробки. Зокрема використовуються стандарти, що регламентують характеристики якості програмного забезпечення, життєвий цикл розробки та рівні зрілості процесів. Застосування стандартизованого підходу дозволяє забезпечити єдине трактування понять якості, зіставність результатів оцінювання та відповідність розробки нормативним вимогам, — насамперед для великих, розподілених або регульованих програмних систем.

Наведені підходи до контролю якості програмного забезпечення, як правило, застосовуються в поєднанні, адже не є взаємовиключними. Процесно-орієнтовані та запобіжні підходи спрямовані на попередження дефектів, продукт-орієнтований — на їх виявлення, а безперервний, архітектурний і користувацько-орієнтований

підходи забезпечують підтримку стабільної якості впродовж усього життєвого циклу програмного забезпечення.

Рис.1 Підходи до контролю якості ПЗ

Разом із тим ефективність застосування наведених підходів до контролю якості значною мірою залежить від рівня, на якому виникають потенційні проблеми програмного модуля. Якість програмного забезпечення не є одновимірною характеристикою і не може бути зведена лише до коректності реалізації окремих фрагментів коду. Дефекти можуть виникати на різних рівнях абстракції — від рівня вихідного коду до рівня архітектурної організації системи — і кожен із цих рівнів по-різному впливає на загальну якість програмного забезпечення, складність виявлення проблем та вартість їх усунення.

На рівні вихідного коду програмні модулі зазвичай містять дефекти, що мають локальний характер. До них належать синтаксичні та семантичні помилки, некоректна реалізація алгоритмів, порушення контрактів функцій, неправильна обробка помилкових або граничних випадків. Подібні проблеми, як правило, добре піддаються автоматизованому виявленню за допомогою статичного аналізу, модульного тестування та перевірки коду, а їх виправлення не потребує суттєвих змін у структурі програмного забезпечення. Хоча дефекти цього рівня можуть призводити до збоїв у роботі окремих функцій або модулів, їх вплив на загальну якість системи зазвичай є обмеженим, а вартість усунення — відносно низькою.

Більш складними для аналізу є проблеми, що виникають на рівні внутрішньої структури програмного модуля. Вони пов’язані не стільки з помилками реалізації, скільки з організацією коду, взаємодією між класами або компонентами та дотриманням принципів проєктування. Надмірна складність, високе зчеплення та низька зв’язність компонентів можуть не проявлятися у вигляді негайних та очевидних проблем, однак вони істотно ускладнюють супровід, тестування та розвиток програмного забезпечення. Такі дефекти часто накопичуються поступово й призводять до зростання технічного боргу, знижуючи здатність системи адаптуватися до змін вимог без суттєвих витрат.

На відміну від локальних і структурних проблем, архітектурні дефекти мають системний характер і проявляються на рівні взаємодії між програмними модулями. Вони пов’язані з вибором архітектурних рішень, визначенням меж відповідальності модулів, організацією залежностей та способами обміну даними. У багатьох випадках програмна система з архітектурними недоліками функціонує коректно з точки зору окремих сценаріїв використання, а наявні проблеми стають помітними лише зі зростанням обсягу коду, кількості модулів або складності вимог.

Складність виявлення архітектурних дефектів зумовлена їх неявною природою. На відміну від помилок коду, які можна зафіксувати під час виконання тестів або статичного аналізу, архітектурні проблеми часто проявляються опосередковано — у вигляді зниження продуктивності, зростання кількості дефектів при внесенні змін, ускладнення тестування або необхідності дублювання функціональності. У багатьох випадках такі симптоми сприймаються як неминучий наслідок зростання системи, що ускладнює своєчасну ідентифікацію першопричин проблем.

Архітектурні дефекти також характеризуються значно вищою ціною помилки порівняно з проблемами нижчих рівнів. Усунення архітектурних недоліків, як правило, потребує змін у кількох програмних модулях одночасно, перегляду інтерфейсів взаємодії, модифікації механізмів обміну даними та повторного виконання значної частини тестів. У багатьох випадках виправлення архітектурної помилки призводить до каскадних змін, що суттєво підвищує ризик появи нових

дефектів і збільшує витрати часу та ресурсів. Якщо такі проблеми виявляються на пізніх етапах життєвого циклу або після введення системи в експлуатацію, їх усунення може вимагати часткової або повної переробки архітектури, що істотно впливає на економічну доцільність подальшого розвитку програмного забезпечення.

Таким чином, дефекти на різних рівнях аналізу програмного модуля мають різний вплив на загальну якість програмного забезпечення. Локальні помилки коду є відносно простими у виявленні та виправленні, тоді як архітектурні проблеми характеризуються складністю ідентифікації, системним впливом і високою вартістю помилки. Це зумовлює необхідність приділяти особливу увагу аналізу архітектурних рішень і взаємодії програмних модулів, оскільки саме на цьому рівні формується довгострокова якість програмного забезпечення та його здатність до еволюції.

Враховуючи системний характер архітектурних дефектів, складність їх своєчасного виявлення та високу вартість усунення, аналіз саме архітектури програмного забезпечення набуває особливого значення в контексті забезпечення якості. На відміну від локальних помилок реалізації, архітектурні проблеми не завжди можуть бути виявлені за допомогою стандартних засобів тестування або статичного аналізу коду, що зумовлює необхідність застосування спеціалізованих методів і підходів до їхньої ідентифікації.

Саме тому подальший розгляд зосереджується на методах аналізу архітектури програмного забезпечення, які дозволяють оцінювати якість модульної структури, взаємодію компонентів та потенційні ризики, пов’язані з архітектурними рішеннями заздалегідь, до того, як їхні наслідки стануть критичними для функціонування та розвитку системи.

З огляду на специфіку задачі аналізу архітектури, найбільш релевантними для нашого дослідження є архітектурно-орієнтований та метрично-орієнтований підходи — саме вони складають основу запропонованого рішення.

1.2 Архітектура програмного модуля

На сьогодні існує низка підходів до аналізу архітектури програмного забезпечення, які відрізняються як за джерелами вхідних даних, так і за методами їх обробки. Зокрема, у дослідженнях розглядаються підходи, що базуються на аналізі вихідного коду, проєктної документації, історії змін програмної системи, а також її поведінки під час виконання. Кожен із зазначених підходів дозволяє отримати часткове уявлення про структуру та властивості архітектури, однак має обмеження, пов’язані з повнотою, точністю або інтерпретованістю результатів.

Аналіз на основі структури вихідного коду передбачає побудову моделі архітектури програмного модуля на основі безпосереднього аналізу його вихідного коду. Архітектура при цьому представляється у вигляді графа, в якому вузли відповідають структурним елементам програми, таким як класи, методи та властивості, а ребра — зв’язкам між ними, зокрема викликам, використанню або відношенням належності [3], [17], [18], [19], [20], [21]. Такий спосіб представлення дозволяє формалізувати структуру програмного модуля та перейти від текстового опису коду до математично визначеної моделі.

Побудова такої моделі здійснюється шляхом статичного синтаксичного та семантичного аналізу коду, наприклад, із використанням абстрактного синтаксичного дерева або графа залежностей. У процесі аналізу виділяються структурні елементи програми та визначаються зв’язки між ними на основі правил мови програмування. Отримане представлення дозволяє застосовувати формальні методи аналізу, включаючи обчислення метрик, дослідження структурних характеристик графа та виявлення потенційних аномалій. Зокрема, можуть використовуватися такі характеристики, як степінь вузлів, щільність графа, наявність компонент зв’язності, циклів та інших структурних особливостей, що прямо або опосередковано відображають якість архітектурних рішень.

Перевагою підходу є те, що він ґрунтується на вихідному коді як безпосередньому та повному джерелі інформації про систему. Це дозволяє забезпечити високу точність аналізу та повне покриття програмного модуля без необхідності залучення додаткових джерел інформації про нього. Крім того, такий

аналіз може бути повністю автоматизований, що робить його придатним для використання у великих програмних системах. Важливою перевагою є також незалежність від якості супровідної документації, яка на практиці часто є неповною або застарілою. Формалізоване графове представлення відкриває можливість застосування ефективних алгоритмів аналізу, а також спрощує інтеграцію з іншими підходами, зокрема методами машинного навчання.

Підхід, однак, має низку обмежень. Зокрема, він не враховує наміри розробників, що можуть бути відображені лише у документації, а також не дозволяє аналізувати поведінку системи під час виконання. Це означає, що отримана модель відображає лише потенційні, а не фактичні сценарії взаємодії компонентів. Окрім цього, значним недоліком є те, що отримані графові моделі часто містять значну кількість низькорівневих зв'язків, що ускладнює їх інтерпретацію та вимагає додаткових засобів узагальнення або візуального представлення. У великих проєктах це може призводити до надмірної складності моделі та втрати наочності.

Результати аналізу чутливі до особливостей мови програмування та стилю реалізації. Наприклад, використання певних патернів або мовних конструкцій може штучно збільшувати кількість зв'язків у графі без фактичного ускладнення архітектури. Також статичний аналіз не завжди дозволяє коректно обробляти динамічні механізми, такі як рефлексія, динамічне завантаження або використання функцій вищого порядку, що може призводити до неповноти або викривлення моделі.

Крім того, отримана графова модель не завжди безпосередньо відображає логічні межі модулів або відповідальностей у системі, що ускладнює інтерпретацію результатів аналізу без залучення додаткового контексту. У зв'язку з цим ефективне використання підходу часто потребує застосування додаткових методів обробки, таких як агрегування елементів, фільтрація зв'язків або візуалізація, що дозволяє підвищити інформативність отриманої моделі.

Таким чином, аналіз на основі структури вихідного коду є фундаментальним підходом до дослідження архітектури програмного модуля, який забезпечує високу

точність результатів, проте потребує доповнення іншими методами для врахування динамічних аспектів системи та підвищення інтерпретованості отриманих даних.

Окремий клас підходів до аналізу архітектури програмного забезпечення базується на *дослідженні історії змін програмного ко*ду, зокрема даних систем контролю версій [22], [23]. На відміну від статичного аналізу структури, у даному випадку архітектура розглядається в контексті її еволюції, що дозволяє виявляти приховані залежності між компонентами системи. Такий підхід дозволяє перейти від аналізу фіксованого стану системи до дослідження процесу її розвитку, що є важливим для розуміння довгострокових архітектурних тенденцій.

Метод ґрунтується на аналізі спільних змін елементів коду у процесі розвитку програмного забезпечення. Якщо певні компоненти системи регулярно змінюються разом у межах одних і тих самих комітів, це може свідчити про наявність між ними тісного зв’язку, навіть у випадках, коли такий зв’язок не є явно вираженим у структурі коду. На основі цих даних формується граф спільних змін, що відображає так звану темпоральну зв’язність компонентів. Додатково можуть враховуватись такі характеристики, як частота змін, розмір комітів, кількість авторів та часові інтервали між змінами, що дозволяє більш детально описати динаміку розвитку системи.

Перевагою такого підходу є можливість виявлення неявних залежностей, що формуються в процесі розвитку системи та можуть не бути представленими у статичній структурі коду. Це особливо важливо для великих проєктів, у яких архітектура еволюціонує поступово і не завжди відповідає початковим проєктним рішенням. Крім того, аналіз історії змін дозволяє досліджувати процеси деградації архітектури, зокрема поступове зростання зв’язності, накопичення технічного боргу або виникнення нестійких залежностей між модулями. Такий підхід також дає змогу виявляти області коду, що є найбільш чутливими до змін, а отже можуть становити підвищений ризик для стабільності системи.

Важливо, що дані систем контролю версій доступні для більшості сучасних проєктів без залучення додаткових інструментів, що робить підхід придатним для

ретроспективного аналізу наявних систем і дослідження еволюції відкритих проєктів.

Разом з тим, результати аналізу значною мірою залежать від якості історії змін. Наявність великих або нерегулярних комітів, рефакторингів без чіткої семантики змін, а також недостатньо інформативні повідомлення можуть призводити до появи шуму в даних і ускладнювати інтерпретацію результатів. Зокрема, одночасні зміни великої кількості файлів можуть створювати хибні зв'язки між компонентами, які фактично не є логічно пов'язаними.

Окрім цього, підхід не дозволяє безпосередньо визначити тип залежності між компонентами, а лише вказує на факт їх спільної еволюції. Це означає, що для коректної інтерпретації результатів необхідне додаткове зіставлення з іншими джерелами інформації, зокрема структурою коду або документацією. До того ж, історія змін відображає лише реалізовані сценарії розвитку системи і не гарантує повного покриття всіх можливих взаємодій між компонентами.

Практичною проблемою є також складність обробки та агрегації великих обсягів даних у довготривалих проєктах. Це може вимагати застосування спеціалізованих методів фільтрації та узагальнення, що ускладнює практичне використання підходу.

Отже, аналіз на основі історії змін програмного забезпечення дозволяє отримати цінну інформацію про еволюцію архітектури та виявити приховані залежності між компонентами, однак потребує обережної інтерпретації результатів і, як правило, ефективно застосовується у поєднанні з іншими підходами до аналізу.

Використання проєктної документації як основного джерела інформації є ще одним підходом до аналізу архітектури програмного забезпечення. На відміну від аналізу вихідного коду або історії змін, у цьому випадку архітектура розглядається через призму її початкового проєктування, що дозволяє відобразити наміри розробників та концептуальну структуру системи. Такий підхід орієнтований не стільки на фактичну реалізацію, скільки на логіку побудови системи, закладену на етапі її розробки [24], [25], [26].

Підхід полягає у виділенні структурних елементів та зв'язків між ними з текстових артефактів, таких як технічні описи, специфікації, README-файли або інші супровідні документи. Для цього застосовуються методи обробки природної мови, що дозволяють ідентифікувати сутності, їх властивості та взаємозв’язки, після чого формується відповідна модель архітектури. У більш складних випадках можуть використовуватись підходи, що враховують контекст, семантику термінів та їх взаємне розташування в тексті, що дозволяє підвищити точність відновлення архітектурних зв’язків.

Перевагою такого підходу є можливість відобразити архітектуру на високому рівні абстракції з урахуванням початкових проєктних рішень. Це особливо важливо у випадках, коли фактична реалізація відрізняється від задекларованої архітектури, або коли необхідно оцінити відповідність реалізації проєктним вимогам. Крім того, використання документації дозволяє врахувати ті аспекти системи, які не можуть бути безпосередньо виведені з коду, зокрема нефункціональні вимоги, обмеження або принципи побудови архітектури. Такий підхід також може бути корисним на ранніх етапах розробки, коли програмна система ще не має повної реалізації.

Проте проєктна документація часто є неповною або застарілою, що може призводити до некоректного відображення актуального стану системи. У реальних проєктах документація нерідко відстає від коду, що знижує її надійність як джерела інформації для аналізу. Підхід також не забезпечує повного покриття системи, оскільки зазвичай не всі її аспекти відображаються у документації, особливо на рівні деталей реалізації.

Додатковою проблемою є неоднозначність природної мови, що ускладнює автоматичну обробку текстових даних. Методи обробки природної мови не гарантують повної точності при виділенні сутностей і зв’язків, що є джерелом невизначеності та створює додаткові труднощі при побудові достовірної моделі архітектури. Помилки на етапі інтерпретації тексту можуть призводити до втрати важливих зв’язків або, навпаки, до появи хибних залежностей.

Ефективність підходу значною мірою залежить від якості та структури самої документації. Неформалізовані або слабо структуровані описи значно

ускладнюють автоматичний аналіз, тоді як формалізовані документи (наприклад, архітектурні описи або специфікації) зустрічаються значно рідше. Варто також мати на увазі, що документація може містити лише узагальнені описи, які не відображають реальних взаємодій між компонентами на рівні реалізації.

З іншого боку, нещодавні напрацювання в цьому напрямку демонструють значний потенціал у використанні сучасних методів обробки природної мови, зокрема великих мовних моделей, для підвищення точності виділення структурних елементів та зв'язків. Це відкриває перспективи для більш ефективного поєднання аналізу документації з іншими підходами, зокрема аналізом вихідного коду.

Відповідно, аналіз архітектури на основі проєктної документації дозволяє отримати уявлення про концептуальну модель системи та наміри її розробників, однак його ефективність обмежується якістю документації та складністю автоматичної інтерпретації текстових даних, що зумовлює доцільність його використання у поєднанні з іншими методами аналізу.

Окрему групу підходів до аналізу архітектури становлять методи, що базуються на *динамічному аналізі програми*. У цьому випадку архітектура розглядається як сукупність фактичних взаємодій між компонентами, що відбуваються у процесі виконання програми, а не лише як її статичне представлення. Такий підхід дозволяє перейти від потенційної структури системи до аналізу її реальної поведінки в конкретних умовах експлуатації [20], [27], [28], [29].

Основою цього підходу є збір та аналіз динамічних даних, зокрема журналів подій, трас виконання та метрик взаємодії між компонентами. Збір таких даних може здійснюватися шляхом інструментування коду, використання спеціалізованих засобів моніторингу або вбудованих механізмів трасування. На підставі отриманих даних формується модель, що відображає реальні сценарії взаємодії елементів системи, включаючи частоту викликів, послідовність операцій та інтенсивність обміну даними. У результаті може бути побудований динамічний граф взаємодій, який відображає фактичні шляхи виконання програми.

Перевагою підходу є можливість отримати уявлення про фактичну поведінку системи, що дозволяє виявляти проблеми, пов’язані з виконанням, зокрема вузькі місця, надмірну зв’язність або неефективні сценарії взаємодії компонентів. Такий аналіз є особливо актуальним для систем із складною динамікою, зокрема розподілених або мікросервісних архітектур, де статичний аналіз не завжди дозволяє коректно оцінити характер взаємодії між компонентами. Крім того, динамічний аналіз дозволяє враховувати вплив середовища виконання, зокрема навантаження, конфігурацію системи та поведінку користувачів, що робить результати більш наближеними до реальних умов експлуатації.

Крім того, динамічний аналіз дозволяє виявляти проблеми продуктивності та масштабованості, які не проявляються у статичній структурі коду. Зокрема, аналіз частоти викликів та інтенсивності взаємодій дозволяє визначити критичні ділянки системи, що можуть потребувати оптимізації або перегляду архітектурних рішень.

З іншого боку, застосування даного підходу передбачає необхідність організації збору динамічних даних, що ускладнює його використання та може впливати на продуктивність системи. Інструментування коду або використання механізмів трасування часто супроводжується додатковими витратами ресурсів, що може бути критичним для систем із жорсткими вимогами до швидкодії.

Крім того, отримані результати залежать від обмеженої і часто заданої вручну множини сценаріїв виконання, що були враховані під час збору даних, і не гарантують повного покриття всіх можливих варіантів поведінки. Це означає, що деякі взаємодії між компонентами можуть залишитися невиявленими, якщо відповідні сценарії не були відтворені під час аналізу. Також динамічні дані можуть містити значний обсяг шуму, зокрема викликаного службовими операціями або низькорівневими механізмами виконання, що ускладнює їх подальшу обробку та інтерпретацію.

Результати можуть також суттєво змінюватися залежно від умов виконання, що ускладнює їх узагальнення та порівняння між версіями. Це ускладнює порівняння різних версій програмного забезпечення або використання отриманих моделей для довгострокового аналізу архітектури.

Таким чином, динамічний аналіз дозволяє отримати найбільш наближене до реальності уявлення про взаємодію компонентів програмного модуля, проте його ефективність обмежується складністю збору даних, залежністю від сценаріїв виконання та необхідністю додаткової обробки отриманої інформації, що зумовлює доцільність його використання у поєднанні з іншими підходами до аналізу архітектури.

Окремий напрям аналізу архітектури програмного забезпечення пов’язаний із *виявленням архітектурних антипатернів та проявів технічного боргу*. На відміну від попередньо розглянутих підходів, які зосереджуються на побудові моделі або описі структури системи, у цьому випадку основна увага приділяється безпосередньому виявленню ознак неякісних архітектурних рішень. Такий підхід орієнтований на практичні аспекти оцінювання якості, зокрема на ідентифікацію проблем, що можуть негативно впливати на підтримуваність, розширюваність та надійність програмного забезпечення [30], [31], [32], [33], [34].

Зазвичай такі підходи базуються на використанні набору правил або евристик, що дозволяють ідентифікувати характерні патерни порушень (антипатерни), зокрема надмірну зв’язність компонентів, низьку згуртованість, наявність циклічних залежностей або невиправдано складну структуру окремих модулів. Виявлення подібних ознак може здійснюватися як на основі статичного аналізу коду, так і з використанням додаткових метрик або моделей. У більш формалізованих підходах застосовуються кількісні критерії, що дозволяють оцінити ступінь відхилення певних характеристик системи від очікуваних значень, тоді як у простіших випадках використовуються фіксовані правила виявлення порушень.

Перевагою такого підходу є його практична спрямованість, оскільки результати аналізу безпосередньо вказують на потенційно проблемні місця в архітектурі та можуть бути використані для планування рефакторингу. Це дозволяє скоротити витрати на пошук дефектів і підвищити ефективність процесу підтримки програмного забезпечення. Крім того, подібні підходи часто добре інтегруються у

процеси безперервної інтеграції та автоматизованого контролю якості, що дозволяє виявляти проблеми на ранніх етапах розробки.

Виявлені антипатерни зазвичай мають чітке практичне трактування та безпосередньо пов'язані з конкретними фрагментами коду чи архітектурними рішеннями — це спрощує роботу розробникам і архітекторам, які не мають потреби аналізувати складні формальні моделі.

Разом з тим, використання евристичних правил обмежує універсальність підходу, оскільки визначення того, що вважати архітектурною вадою, значною мірою залежить від контексту конкретної системи. Архітектурні рішення, які в одних умовах можуть вважатися небажаними, в інших можуть бути обґрунтованими або навіть необхідними. Це ускладнює формалізацію універсальних критеріїв якості та потребує гнучкого налаштування правил аналізу.

Крім того, подібні методи можуть генерувати хибні спрацювання, що ускладнює інтерпретацію результатів та вимагає додаткового аналізу з боку розробника. У деяких випадках це може призводити до зниження довіри до результатів автоматизованого аналізу. Більшість таких підходів зосереджені на окремих локальних ознаках і не завжди дозволяють отримати цілісне уявлення про архітектуру системи.

Подібні методи здебільшого фіксують уже наявні проблеми, не пояснюючи причин їх виникнення або передбачити подальший розвиток архітектури. Це зменшує їх ефективність у задачах довгострокового аналізу та планування еволюції системи.

Підходи до виявлення архітектурних антипатернів та технічного боргу є важливим інструментом практичного аналізу якості програмного забезпечення, однак їх ефективність обмежується залежністю від евристичних правил, складністю узагальнення результатів та необхідністю поєднання з іншими методами для отримання повної картини архітектури системи.

Розглянуті підходи до аналізу архітектури програмного забезпечення демонструють, що кожен із них дозволяє отримати певний аспект уявлення про структуру та властивості системи. Зокрема, аналіз на основі вихідного коду

забезпечує точність та повноту представлення, підходи, що базуються на історії змін, дозволяють врахувати еволюційні залежності між компонентами, використання документації дає змогу відобразити початкові проєктні рішення, а динамічний аналіз дозволяє дослідити фактичну поведінку системи під час виконання. Методи виявлення архітектурних антипатернів, у свою чергу, забезпечують практичний інструментарій для ідентифікації потенційно проблемних місць.

Відповідно, можна стверджувати, що основною перевагою існуючих підходів є їхня здатність надавати формалізовану або інтерпретовану інформацію про різні аспекти архітектури програмного модуля. Вони дозволяють здійснювати як кількісну оцінку структурних характеристик, так і виявляти окремі ознаки неякісних архітектурних рішень, що є важливим для забезпечення якості програмного забезпечення.

Водночас кожен із підходів має суттєві обмеження. Зокрема, аналіз на основі вихідного коду не враховує динамічні аспекти функціонування системи та початкові наміри проєктування, підходи на основі історії змін залежать від якості репозиторію та можуть містити значний обсяг шуму, аналіз документації обмежується її повнотою та актуальністю, а динамічний аналіз не забезпечує повного покриття всіх можливих сценаріїв виконання. Методи виявлення архітектурних антипатернів, у свою чергу, залежать від евристичних правил і не завжди дозволяють отримати узагальнене уявлення про систему.

Спільною вадою більшості підходів є те, що результати аналізу — числові показники або великі структурні моделі — потребують суттєвої додаткової обробки, перш ніж стати придатними для практичних рішень.

Жоден із розглянутих підходів не забезпечує одночасно повного покриття системи, точності аналізу, врахування динамічних аспектів і зручності інтерпретації результатів.

У зв’язку з цим виникає необхідність у розробці підходу, який дозволяє поєднати переваги існуючих методів аналізу та мінімізувати їх обмеження, зокрема

шляхом формалізації представлення архітектури, застосування кількісних характеристик та забезпечення інтуїтивного сприйняття результатів аналізу.

1.3 Візуалізація компонентів програмних модулів

Візуалізація структури програмного забезпечення є важливим інструментом аналізу, що дозволяє представити складні структурні та поведінкові характеристики програмного модуля у формі, придатній для сприйняття людиною. На відміну від формальних моделей або числових метрик, візуальні представлення забезпечують можливість швидкого виявлення закономірностей, аномалій та взаємозв’язків між компонентами системи.

Особливу роль візуалізація відіграє у задачах дослідження архітектури, де необхідно одночасно враховувати велику кількість елементів та зв’язків між ними. У таких умовах текстові або табличні представлення є недостатніми, що зумовлює необхідність використання спеціалізованих підходів до візуалізації.

Сучасні підходи до візуалізації програмного забезпечення відрізняються за способом представлення даних, рівнем абстракції та типом інформації, що відображається. Класично в літературі [35], [36], [37] описують кілька основних класів таких підходів, кожен з яких має власні особливості, переваги та обмеження.

Графові підходи до візуалізації є одним із найбільш поширених способів представлення структури програмного забезпечення. Як і у випадку з аналізом архітектури на основі графової моделі, в межах цього підходу програмний модуль представляється у вигляді графа, де вузли відповідають структурним елементам системи, а ребра відображають зв’язки між ними [38], [39], [40].

Основною ідеєю в цьому випадку є відображення структури програмного забезпечення у формалізованому вигляді, що дозволяє поєднати візуальне представлення з можливістю застосування алгоритмічних методів аналізу. Побудований граф може бути використаний як для безпосереднього відображення залежностей, так і як основа для подальшої обробки, зокрема агрегації, кластеризації або обчислення метрик.

Перевагою графових візуалізацій є їх універсальність та природна відповідність структурі програмного забезпечення. Більшість архітектурних моделей можуть бути безпосередньо відображені у вигляді графів, що робить цей підхід зручним для інтеграції з методами аналізу, розглянутими раніше. Крім того, графові візуалізації дозволяють відображати як ієрархічні, так і неієрархічні зв’язки між компонентами, що є важливим для аналізу реальних програмних систем.

Алгоритми автоматичного розміщення вузлів, зокрема force-directed підходи, дозволяють виявляти структурні особливості графа — кластери та щільно пов'язані компоненти. Це сприяє виявленню потенційних архітектурних проблем.

Головним обмеженням графових візуалізацій є слабка масштабованість. Зі збільшенням кількості вузлів та ребер граф швидко перетворюється на перевантажене представлення, в якому окремі елементи та зв'язки стають складними для сприйняття. Це призводить до ефекту так званого “hairball”, коли структура графа втрачає наочність і практичну цінність.

Значна кількість перетинів ребер підвищує когнітивне навантаження на користувача та ускладнює інтерпретацію. Навіть при використанні алгоритмів оптимізації розміщення, повністю уникнути цієї проблеми у великих системах, як правило, неможливо.

Якість візуалізації також суттєво залежить від обраного алгоритму розміщення. Різні алгоритми можуть давати суттєво відмінні результати, що впливає на інтерпретацію структури системи. При цьому відсутність єдиного критерію “якісного” розміщення ускладнює порівняння різних візуалізацій.

Графові візуалізації зазвичай відображають структуру лише на низькому рівні деталізації, що ускладнює їх використання для аналізу архітектури на високому рівні. Для підвищення інформативності часто необхідно застосовувати додаткові методи узагальнення, такі як групування вузлів або фільтрація зв’язків, що ускладнює побудову та інтерпретацію моделі.

Отже, графові підходи до візуалізації забезпечують формалізоване та універсальне представлення структури програмного забезпечення, однак їх ефективність обмежується проблемами масштабованості, перевантаження

візуального представлення та залежністю від методів розміщення елементів, що зумовлює необхідність використання додаткових засобів обробки та узагальнення даних.

Матричні підходи до візуалізації базуються на представленні взаємозв’язків між компонентами програмного забезпечення у вигляді матриці, в якій рядки та стовпці відповідають елементам системи, а значення у комірках відображають наявність або інтенсивність зв’язку між ними. Такий спосіб представлення часто використовується для відображення графових структур у більш компактній та формалізованій формі.

Матричний підхід замінює явне відображення зв'язків (як у графах) на табличне представлення, що дозволяє уникнути проблем, пов’язаних із візуальним перевантаженням. У найпростішому випадку використовується бінарна матриця суміжності, однак у більш складних варіантах значення можуть відображати частоту взаємодії, силу залежності або інші кількісні характеристики.

Перевагою матричних візуалізацій є їх висока масштабованість. На відміну від графових представлень, збільшення кількості елементів не призводить до експоненційного зростання складності візуалізації у вигляді перетинів або накладання зв’язків. Це дозволяє ефективно працювати з великими програмними системами, де графові підходи стають непридатними для безпосереднього використання.

Впорядкування рядків і стовпців матриці дозволяє виявляти структурні закономірності — кластери та групи тісно пов'язаних компонентів. У результаті можуть бути виявлені логічні підсистеми або аномалії у структурі залежностей. Також матричні представлення добре поєднуються з кількісними методами аналізу, оскільки безпосередньо відображають числові характеристики зв’язків.

Головним недоліком матричних візуалізацій є їх відносно низька наглядність для користувача. На відміну від графових представлень, де зв’язки між компонентами можна безпосередньо спостерігати, у матричному вигляді вони представлені опосередковано, що ускладнює їх швидке сприйняття.

Інтерпретація матриці залежить від порядку розташування елементів. Без відповідного впорядкування матриця може виглядати як випадковий набір значень, що не дозволяє зробити змістовні висновки. Це зумовлює необхідність застосування додаткових алгоритмів обробки, тобто ускладнює використання підходу.

Складно також відстежувати конкретні шляхи взаємодії між компонентами. Якщо у графовій візуалізації такі шляхи можуть бути простежені безпосередньо, то у матричному представленні це потребує додаткових обчислень і не має наочного відображення.

Також матричні підходи менш придатні для відображення ієрархічної структури системи, оскільки не забезпечують природного способу представлення вкладеності компонентів. У результаті, їх застосування часто обмежується задачами аналізу залежностей або еволюції системи, а не повноцінного відображення архітектури.

Отже, матричні підходи до візуалізації забезпечують компактне та масштабоване представлення взаємозв'язків між компонентами програмного забезпечення, однак їх ефективність обмежується складністю інтерпретації та необхідністю додаткової обробки для виявлення структурних закономірностей.

Ієрархічні підходи до візуалізації базуються на представленні програмного забезпечення у вигляді деревоподібної структури, що відображає вкладеність його компонентів. У межах такого підходу вузли дерева відповідають елементам системи, таким як пакети, модулі, класи або методи, а ребра визначають відношення належності між ними.

Основною ідеєю цього підходу є відображення структурної організації програмного забезпечення відповідно до його логічної або фізичної декомпозиції. Таке представлення дозволяє наочно продемонструвати розподіл системи на підсистеми та рівні абстракції, що є важливим для розуміння загальної архітектури.

Поширеними реалізаціями підходу є класичні деревоподібні діаграми, а також більш компактні форми, зокрема treemap-візуалізації, у яких вкладеність компонентів відображається через ієрархічне розбиття простору. У таких представленнях додаткові характеристики, наприклад розмір або складність

компонентів, можуть бути відображені через площу або колір відповідних елементів.

Перевагою ієрархічних візуалізацій є їх відносна простота та інтуїтивність. Вони добре відповідають способу організації більшості програмних систем, що зазвичай мають ієрархічну структуру. Це дозволяє користувачу швидко орієнтуватися у системі та переходити між різними рівнями деталізації.

Ще одною перевагою є ефективне використання простору, особливо у випадку treemap-візуалізацій. Це дозволяє відображати великі системи у компактному вигляді. Також вони добре підходять для візуалізації метрик, наприклад розміру коду або складності, шляхом відображення відповідних значень у вигляді візуальних атрибутів.

Головним обмеженням ієрархічних підходів є те, що вони відображають лише відношення вкладеності та не дозволяють повноцінно представити довільні залежності між компонентами. У реальних програмних системах значна частина взаємодій має неієрархічний характер, що робить такі візуалізації неповними з точки зору аналізу архітектури та мало придатними для практичного застосування.

У складних системах із великою кількістю рівнів вкладеності такі візуалізації можуть втрачати наочність, оскільки користувачеві складно одночасно охопити як загальну структуру, так і деталі на нижчих рівнях. Це ускладнює аналіз взаємодій між віддаленими компонентами.

Treemap-подібні візуалізації, попри компактність, не забезпечують жодного очевидного способу відображення зв'язків між елементами. У результаті користувач отримує інформацію про структуру та розподіл компонентів, але не про характер їх взаємодії.

Інтерпретація таких візуалізацій може бути ускладнена у випадках, коли структура системи не є строго ієрархічною або містить перехресні залежності між компонентами.

Відповідно, ієрархічні підходи до візуалізації забезпечують просте та інтуїтивне представлення структури програмного забезпечення, однак їх можливості

обмежуються відображенням лише відношень вкладеності, що знижує їх ефективність для повноцінного аналізу архітектури складних систем.

Метафоричні підходи [41] до візуалізації програмного забезпечення базуються на використанні аналогій з об’єктами реального світу для представлення структури та властивостей програмних систем. На відміну від формальних графових або матричних моделей, у даному випадку візуалізація орієнтована на інтуїтивне сприйняття інформації за рахунок використання знайомих користувачеві образів.

Компоненти програмного забезпечення відображаються у вигляді об'єктів певної метафоричної системи, а їх властивостей — через візуальні характеристики цих об’єктів, такі як розмір, форма, колір або розташування. Це дозволяє поєднати структурну інформацію з додатковими атрибутами, зокрема метриками, у межах єдиного візуального представлення.

Одним із найбільш відомих прикладів такого підходу є метафора міста (CodeCity) [42], у якій програмна система представляється як віртуальне місто. У межах цієї метафори пакети або модулі відображаються як райони міста, класи — як будівлі, а методи — як внутрішні елементи цих будівель. Висота будівель може відповідати кількості рядків коду або складності класу, а площа — кількості методів або інших характеристик. Просторове розташування елементів визначається їх структурними зв’язками або ієрархією.

Метафора міста вирізняється наочністю: користувач одразу бачить надмірно великі або складні компоненти та нерівномірний розподіл складності. Підхід підтримує багаторівневий аналіз — від загального огляду до деталей окремих компонентів.

Перевагою метафоричних підходів є також можливість інтеграції різних типів інформації у межах одного представлення. Окрім структурних зв’язків, можуть відображатися метрики, динамічні характеристики або результати аналізу, що підвищує інформативність візуалізації. Також використання просторових метафор дозволяє ефективніше використовувати тривимірний простір порівняно з традиційними графовими підходами.

Головним недоліком є відсутність строгого формального зв'язку між елементами метафори та властивостями програмного забезпечення. Відображення характеристик системи через візуальні параметри є певною мірою умовним і може бути інтерпретоване неоднозначно.

Ефективність підходу залежить від правильного вибору метафори та параметрів відображення. Невдале відображення метрик або некоректне масштабування може призводити до викривлення сприйняття структури системи. У складних випадках це може ускладнювати, а не спрощувати аналіз.

Додатковим обмеженням є складність точного аналізу взаємозв’язків між компонентами. На відміну від графових підходів, де зв’язки явно представлені у вигляді ребер, у метафоричних візуалізаціях вони часто відображаються опосередковано або взагалі не відображаються, що обмежує можливості їх використання для автоматизації аналізу.

Окрім цього, використання тривимірного простору, характерне для багатьох метафоричних підходів, може створювати додаткове когнітивне навантаження. Проблеми перекриття об’єктів, необхідність навігації у просторі та складність оцінки відстаней можуть ускладнювати сприйняття інформації, особливо у великих системах.

Ефективність метафоричних підходів до візуалізації, зокрема метафори міста, обмежується складністю формалізації, неоднозначністю інтерпретації та обмеженими можливостями для точного аналізу структурних зв'язків.

Метрико-орієнтовані підходи до візуалізації базуються на відображенні кількісних характеристик програмного забезпечення у графічній формі. На відміну від підходів, що безпосередньо відображають структуру системи, у даному випадку основна увага приділяється представленню значень метрик, які характеризують окремі компоненти або систему в цілому [43], [44], [45].

Підхід полягає у перетворенні числових показників, таких як розмір коду, складність, зв’язність або частота змін, у візуальні атрибути, зокрема колір, розмір, інтенсивність або положення елементів. Такі візуалізації можуть бути реалізовані у

вигляді теплових карт (heatmaps), діаграм розподілу або накладання метрик на існуючі структурні представлення, наприклад графи чи ієрархічні моделі.

Перевагою метрико-орієнтованих підходів є їх здатність швидко відображати кількісні характеристики системи та дозволяти виявляти аномалії, такі як надмірно складні або нестабільні компоненти. Використання візуальних атрибутів дозволяє користувачу швидко оцінити стан системи без необхідності аналізу числових значень у табличній формі.

Перевагою вважається й можливість простого поєднання з іншими підходами до візуалізації. Зокрема, метрики можуть бути інтегровані у графові, ієрархічні або метафоричні представлення, що підвищує їх інформативність та дозволяє одночасно аналізувати як структуру, так і її характеристики.

Метрико-орієнтовані візуалізації мають ряд обмежень. Наприклад, самі по собі метрики не відображають структуру системи, а лише її окремі характеристики. Це ускладнює інтерпретацію результатів без додаткового контексту, зокрема інформації про взаємозв'язки між компонентами.

Ефективність підходу значною мірою залежить від вибору метрик. Некоректно обрані або неправильно інтерпретовані метрики можуть призводити до хибних висновків щодо якості архітектури. Різні метрики можуть суперечити одна одній, що ускладнює формування узагальненого уявлення про систему. Складним є й масштабування при одночасному використанні великої кількості метрик, оскільки це призводить до перевантаження візуального представлення та зниження його наочності.

Отже, метрико-орієнтовані підходи забезпечують ефективний спосіб відображення кількісних характеристик програмного забезпечення, однак їх застосування потребує поєднання зі структурними візуалізаціями для отримання повного уявлення про архітектуру системи.

Підбиваючи підсумок, існуючі підходи до візуалізації характеризуються наявністю взаємодоповнюючих переваг та недоліків і не забезпечують одночасно масштабованість, інтуїтивність, формалізованість та можливість інтеграції з методами аналізу.

У зв'язку з цим виникає необхідність у розробці такого підходу до візуалізації компонентів програмних модулів, який усуватиме ці вади.

1.4 Застосування комбінованого підходу до аналізу та візуалізації архітектури програмного модуля

Проведений аналіз підходів до аналізу архітектури та її візуалізації свідчить про доцільність розгляду методів, у межах яких кількісні характеристики не існують ізольовано, а співвідносяться зі структурними елементами системи. Така інтеграція дозволяє розглядати значення метрик не як окремі показники, а як властивості компонентів у межах їх взаємодії, що підвищує інформативність аналізу.

У цьому контексті доцільним є використання таких способів візуалізації, які поєднують формальну основу з інтуїтивними засобами представлення. Це дозволяє зберегти точність відображення взаємозв’язків між компонентами та одночасно зменшити складність їх сприйняття.

Окрему роль відіграє вибір простору представлення. Використання двовимірних підходів часто призводить до перевантаження зображення при зростанні кількості елементів, тоді як залучення додаткового виміру відкриває можливості для більш ефективного розміщення компонентів та зменшення щільності їх відображення. Це створює передумови для підвищення читабельності складних структур без втрати інформації про їх взаємозв’язки.

Тому ми сформулювали вимоги до узагальненого способу представлення архітектури програмного модуля, який має:

А) зберігати формальний опис структури системи;

Б) забезпечувати можливість кількісної оцінки її характеристик;

В) підтримувати інтуїтивне сприйняття складних залежностей;

Г) бути придатним до масштабування при зростанні розміру системи.

Обґрунтування необхідності розробки власного підходу підтверджується інформацією таблиці 1.1. В ній наведено порівняльний аналіз існуючих засобів аналізу та візуалізації архітектури програмного забезпечення за такими критеріями: джерело даних (статичний або динамічний аналіз), наявність зваженого графа

залежностей, підтримка ваг інтенсивності взаємодій між елементами, рівень деталізації моделі, наявність автоматизованого адаптивного виявлення аномалій та підтримка тривимірної візуалізації.

Таблиця 1.1 — Порівняльний аналіз існуючих інструментів аналізу та візуалізації архітектури (✓ — підтримується; – — не підтримується; ○ — частково)

Інструмент	**Джерело даних**	**Зважений граф залежностей**	**Ваги інтенсивності**	**Рівень деталізації**	**Авт. виявлення аномалій (адаптивне)**	**3D-візуалізація**
SonarQube	Статика	– (метрики)	–	Клас / метод	– (абс. пороги)	–
CodeScene	Git-історія	○ (hotspot)	–	Файл / модуль	○ (еволюційне)	–
JQAssistant	Статика (JVM)	✓ (без ваг)	–	Клас / пакет	– (ручні запити)	–
Understand	Статика	✓ (без ваг)	–	Клас / функція	– (абс. пороги)	–
Structure101	Статика (JVM)	✓ (ієрарх.)	–	Пакет / модуль	–	–
ExplorViz	Runtime (динаміка)	✓ (без ваг)	○ (runtime)	Клас / метод	–	✓ (3D/VR)
A.D.A.R. (пропонований)	Статика	✓	✓ (інтенсивність)	Тип / метод / властивість	✓ (відносне, авт.)	✓ (3D AR)

Як видно з таблиці 1.1, жоден із розглянутих інструментів не задовольняє одночасно всі сформульовані вимоги. Зокрема, інструменти, що будують граф залежностей на основі статичного аналізу (JQAssistant, Understand, Structure101), не підтримують ваг інтенсивності взаємодій між елементами і не забезпечують автоматизованого адаптивного виявлення аномалій [46]. Інструменти з вагами взаємодій (ExplorViz [47]) базуються на динамічному аналізі і не виконують статичного аналізу вихідного коду. Інструменти з автоматичним виявленням відхилень (SonarQube, CodeScene [48]) використовують абсолютні порогові значення або еволюційні дані замість структури графа залежностей. Жоден із наявних інструментів не реалізує повний pipeline: статичний аналіз → зважений граф із вагами інтенсивності → адаптивне виявлення аномалій → тривимірна

просторова візуалізація. Саме поєднання цих чотирьох властивостей є ключовою відмінністю запропонованого нами підходу.

З іншого боку, поєднання різних підходів до аналізу та візуалізації очевидно супроводжується певними обмеженнями, що зумовлені як властивостями окремих методів, так і особливостями їх інтеграції.

Зокрема, використання структурних моделей, отриманих на основі аналізу коду, обмежує можливість врахування динамічних аспектів функціонування системи. Однак у задачах дослідження архітектури основний інтерес становлять саме стабільні структурні залежності, а також фактичний стан системи, а не інтенції, що зменшує критичність цього обмеження.

Іншим аспектом є залежність результатів від вибору характеристик, що використовуються для аналізу. Водночас орієнтація на відносні властивості структури, а не на фіксовані порогові значення, дозволяє зменшити вплив цього фактору та підвищити універсальність підходу.

Використання просторових способів представлення також пов’язане з додатковими вимогами до сприйняття інформації користувачем. Проте за умови наявності засобів взаємодії та можливості адаптації представлення ці обмеження можуть бути зведені до прийнятного рівня.

Крім того, поєднання формальних і інтуїтивних підходів потребує балансування між точністю та наочністю. Надмірна деталізація ускладнює сприйняття, тоді як надмірне узагальнення може призводити до втрати важливої інформації. Це зумовлює необхідність вибору такого способу представлення, який дозволяє змінювати рівень деталізації залежно від задачі аналізу.

Таким чином, хоча комбінування різних підходів супроводжується певними обмеженнями, вони не є визначальними для задачі аналізу архітектури та можуть бути враховані при побудові узагальненого підходу. Це створює підґрунтя для формулювання конкретного методу, що поєднує зазначені властивості у межах єдиної моделі.

1.5 Висновки

У цьому розділі розглянуто проблематику аналізу якості програмного забезпечення з акцентом на архітектурний рівень програмного модуля. Показано, що якість програмного забезпечення не зводиться лише до коректності реалізації окремих фрагментів коду, а визначається також структурною організацією програмних модулів та характером їх взаємодії. При цьому дефекти різних рівнів мають різний вплив на загальну якість системи: локальні помилки коду зазвичай є простішими у виявленні та усуненні, тоді як архітектурні проблеми мають системний характер, складніше піддаються ідентифікації та характеризуються значно вищою вартістю виправлення.

На підставі аналізу підходів до контролю якості програмного забезпечення встановлено, що архітектурно-орієнтований аналіз посідає особливе місце серед засобів забезпечення якості, оскільки дозволяє виявляти проблеми, які не проявляються безпосередньо на рівні тестів або локального статичного аналізу. Це обумовлює доцільність подальшого дослідження саме методів аналізу архітектури програмного модуля.

На основі проведеного аналізу встановлено, що жоден із розглянутих підходів окремо не забезпечує одночасно формалізованого представлення структури програмного модуля, можливості кількісного оцінювання її характеристик, наочності подання та достатньої придатності до масштабування. Разом з тим, виявлено, що окремі класи підходів характеризуються взаємодоповнюючими властивостями, поєднання яких потенційно дозволяє усунути частину зазначених обмежень.

У зв’язку з цим у межах даного дослідження доцільним є розгляд комбінованого підходу, який спирається на формалізоване графове представлення структури програмного модуля, використання кількісних характеристик для аналізу його властивостей, а також візуальне подання, придатне для сприйняття складних архітектурних залежностей. Саме це створює підґрунтя для формулювання конкретного методу аналізу та візуалізації архітектури програмного модуля у наступному розділі.

РОЗДІЛ 2

ВИКОРИСТАННЯ ГРАФОВОЇ МОДЕЛІ ДЛЯ АНАЛІЗУ АРХІТЕКТУРИ ПРОГРАМНОГО МОДУЛЯ

Як ми зазначали, графова модель є одним із найбільш придатних підходів для аналізу архітектури програмного модуля, оскільки дозволяє представити його у вигляді множини компонентів та зв’язків між ними у строгій і водночас універсальній формі.

Використання графа як моделі архітектури дає змогу перейти від текстового опису вихідного коду до формального задання, придатного як для алгоритмічної обробки, так і для подальшої візуалізації.

Практична реалізація запропонованого підходу здійснюється у вигляді програмного комплексу ADAR (Architecture Displayer in Augmented Reality), який поєднує засоби автоматизованого аналізу вихідного коду та інструменти інтерактивної візуалізації отриманої моделі. У цьому розділі розглянуто загальну структуру комплексу, формальну модель архітектури, а також підходи до її побудови та застосування [49], [50].

2.1 Загальна структура програмного комплексу ADAR

Програмний комплекс ADAR реалізує запропонований підхід до аналізу архітектури програмного забезпечення та поєднує засоби автоматизованого аналізу вихідного коду із засобами інтерактивної візуалізації отриманої моделі. Загальна структура комплексу та взаємодія його компонентів наведені на рис. 2.1.

Принциповою особливістю комплексу є те, що побудова моделі архітектури здійснюється виключно на основі вихідного коду програмного модуля без використання додаткових джерел інформації, таких як документація, конфігураційні файли чи ручні анотації. Це дозволяє застосовувати підхід до існуючих проєктів без попередньої підготовки та гарантує відповідність отриманої моделі фактичній реалізації системи.

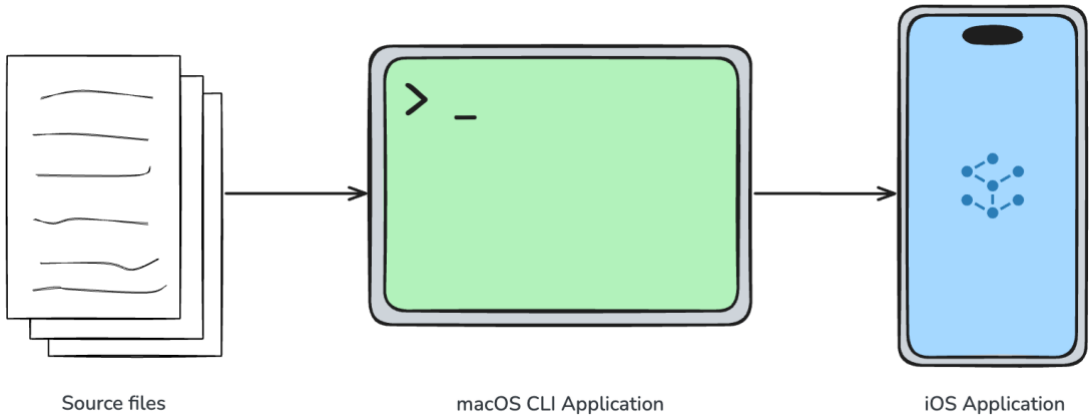


Рис. 2.1 Будова комплексу ADAR

Архітектурно ADAR складається з двох основних підсистем: програми-аналізатора, що виконується в середовищі macOS, та мобільного застосунку-переглядача для платформи iOS. Такий поділ дозволяє винести ресурсоємні задачі аналізу у середовище розробки, тоді як відображення та взаємодія з моделлю здійснюються на окремому пристрої.

Процес роботи комплексу можна умовно поділити на два основні етапи — аналіз та візуалізацію. На етапі аналізу програма-аналізатор отримує на вхід набір вихідних файлів програмного модуля мовою Swift та виконує їх обробку. Результатом цього етапу є побудова формалізованого представлення архітектури у вигляді орієнтованого зваженого графа, що відображає структуру компонентів та інтенсивність взаємодій між ними. Отримана модель серіалізується у файл спеціального формату (*.adar), який використовується як універсальне проміжне представлення.

На етапі відображення мобільний застосунок зчитує збережену модель та відображає її у тривимірному просторі з використанням технологій доповненої реальності. Окрім відображення, застосунок забезпечує інтерактивну взаємодію з моделлю, зокрема навігацію по графу, вибір окремих елементів та аналіз їхніх характеристик, що дозволяє використовувати його як інструмент дослідження архітектури (рис. 2.2).

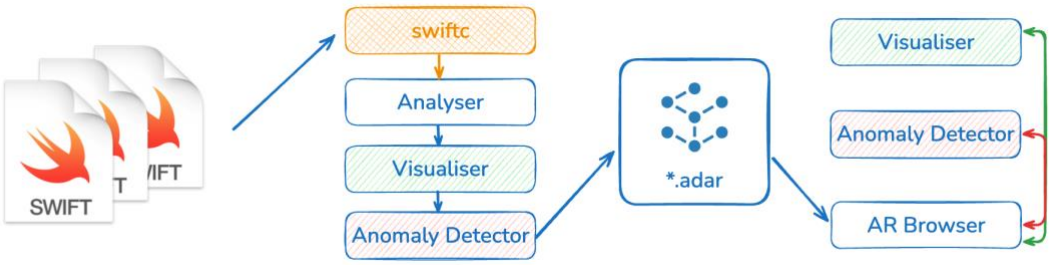


Рис. 2.2 Компоненти комплексу ADAR

Архітектура комплексу також передбачає можливість розширення функціональності шляхом підключення додаткових компонентів. Зокрема система доповнена модулем виявлення антипатернів та альтернативних способів візуалізації, які буде детально описано в наступних розділах. Це дозволяє розглядати ADAR не лише як інструмент відображення, але і як платформу для комплексного аналізу архітектурних властивостей програмного забезпечення.

2.2 Графова модель архітектури програмного модуля

У ході дослідження було запропоновано, а згодом суттєво розвинено графову модель архітектури програмного забезпечення. Для первинної реалізації було обрано мову Swift, що зумовлено її підтримкою об’єктно-орієнтованої парадигми, зручністю практичного використання, а також наявністю розвиненого інструментарію для аналізу структури вихідного коду.

На початковому етапі модель, орієнтована на задачі аналізу та візуалізації, була представлена у вигляді графа, в якому вузли відповідали класам, а ребра — зв’язкам між ними. У межах цієї моделі враховувався, зокрема, зв’язок типу property («є полем»), що відображає використання екземплярів одного класу як полів іншого. Крім того, було введено зв’язок staticFuncUsage, який описує виклики статичних методів одного класу з іншого. Таке представлення дозволяло частково врахувати не лише сам факт існування залежності, але й окремі аспекти її природи.

Подальший розвиток моделі був спрямований на підвищення її аналітичних можливостей, зокрема у контексті дослідження наявності, інтенсивності та типів взаємодій між компонентами системи. З цією метою було здійснено перехід від

представлення на рівні класів до більш детального опису, що охоплює їхні складові частини. У результаті модель набуває вигляду орієнтованого зваженого графа, в якому вузли відповідають окремим елементам архітектури програмного забезпечення, а ребра відображають різні види зв'язків між ними [50].

Таке представлення створює передумови для інтеграції кількісних характеристик архітектури, зокрема метрик, що використовуються для формалізованого оцінювання властивостей програмного модуля. У загальному випадку архітектура може бути задана як явно, наприклад у вигляді документації, так і неявно — через структуру реалізованого коду. У другому випадку доцільним є застосування метрик, що дозволяють оцінювати характеристики взаємодії між компонентами, зокрема рівень їх зв'язності або згуртованості.

Для об'єктно-орієнтованих систем одним із найбільш поширених підходів є використання набору метрик, запропонованого у [51], серед яких важливу роль відіграє метрика Coupling Between Objects (CBO). Вона визначає кількість класів, з якими взаємодіє аналізований клас, і може розглядатися як індикатор складності структури та потенційної нестійкості архітектури [52]. Водночас подібні метрики, як правило, забезпечують лише узагальнену числову оцінку та не дозволяють відтворити детальну структуру відповідних залежностей.

Дані, що використовуються для обчислення метрик, можуть бути інтерпретовані як основа для побудови більш детальної структурної моделі. У нас вони безпосередньо відображаються у вигляді зв'язків між компонентами, тоді як інтенсивність цих зв'язків кодується за допомогою ваг ребер. Таким чином, ваги можна розглядати як узагальнене представлення метрик зв'язності, інтегроване у структуру графа. Детальніший розгляд метрик та їх застосування буде наведено у наступних розділах.

Орієнтованість графа, покладеного в основу моделі, дозволяє явно фіксувати напрям залежності між компонентами, що дає змогу однозначно визначити, який елемент системи використовує інший. Це істотно спрощує встановлення відповідності між елементами моделі та їх деклараціями у вихідному коді, роблячи задачу навігації по структурі системи практично тривіальною. Використання

зважених ребер доповнює це представлення, оскільки дозволяє не лише зафіксувати наявність зв’язку, але й кількісно охарактеризувати його інтенсивність.

Тому для візуалізації моделі застосовується силовий алгоритм розміщення, заснований на фізичній аналогії, ваги ребер безпосередньо впливають на просторову конфігурацію графа. Зокрема, більш інтенсивні зв’язки призводять до зменшення відстані між відповідними вузлами у тривимірному просторі, тоді як слабші взаємодії відображаються більшою віддаленістю. У результаті формується інтуїтивно зрозуміле просторове представлення, у якому тісно пов’язані компоненти групуються між собою, утворюючи кластери, що відображають внутрішню структуру взаємодій у системі.

Ми виділяємо три основні типи вузлів, що відповідають різним категоріям елементів програмного забезпечення: dt (data type) — типи даних у загальному випадку, p (property) — поля відповідних типів, а також m (method) — методи.

Слід зазначити, що ці типи вузлів мають узагальнений характер і не прив’язані до конкретної мови програмування. Вони можуть інтерпретуватися по-різному залежно від особливостей аналізованої мови. Зокрема, у випадку Swift вузли типу dt відповідають таким конструкціям, як класи, структури та переліки, вузли типу p — властивостям, а вузли типу m — методам екземпляра і типу, індексаторам, операторам, ініціалізаторам та деініціалізаторам. Така узагальненість дозволяє використовувати модель для аналізу різних мов програмування без суттєвої зміни її структури.

Ребра графа відображають різні види зв’язків між елементами системи. У найпростішому випадку наявність у одному типі поля іншого типу може бути зведена до прямого зв’язку між відповідними типами. Однак таке представлення є надмірно спрощеним, оскільки не враховує можливості існування кількох полів одного типу, а також не дозволяє коректно відобразити ситуації із двосторонніми або складеними залежностями. З цієї причини у моделі вводиться окремий тип вузлів для представлення полів, що дозволяє деталізувати структуру взаємодій.

Ми використовуємо два взаємопов'язані типи ребер. Ребро типу isPropertyOf з'єднує вузол p із вузлом dt, що відповідає типу, якому належить відповідне поле.

Водночас ребро типу isOfType з'єднує вузол p із вузлом dt, що визначає тип самого поля. Така схема дозволяє більш точно відобразити природу зв'язків і уникнути втрати інформації при їх агрегації.

Окрему складність становить відображення механізму наслідування у межах візуальної моделі. З одного боку, очевидним підходом є транзитивне розповсюдження зв’язків батьківського класу на дочірні, що дозволяє зберегти повноту інформації про доступні взаємодії. Однак у такому випадку сама наявність ієрархії може призводити до формування штучних кластерів високої зв’язності, що спотворює реальну картину залежностей. З іншого боку, повне ігнорування наслідування призводить до втрати важливої інформації про структуру системи. У зв’язку з цим доцільним є компромісний підхід, який полягає у явному представленні відношення наслідування за допомогою окремого ребра типу inherits, що має відносно високу вагу порівняно з іншими декларативними зв’язками. Відповідно, для досягнення нашої мети у вагах таких ребер доцільно використовувати фіксовані значення, що не залежать від частоти використання.

Окрім декларативних зв’язків, у моделі передбачено також відображення взаємодій, що виникають у процесі виконання методів. Для опису таких залежностей ми вводимо поняття «власного» типу, якому належить метод, та «сторонніх» типів, що використовуються в його реалізації. Базовим є зв’язок methodOf, який встановлює відповідність між вузлом методу та вузлом типу, до якого він належить.

У межах методу можуть використовуватися поля власного типу, що формує додаткові зв’язки у графі. Такі взаємодії відображаються за допомогою ребер типу methodUsesProperty, що з’єднують вузли m та p. Незалежно від кількості звернень до відповідного поля у межах одного методу, зв’язок фіксується як один, що дозволяє уникнути надмірного ускладнення структури графа. Водночас інтенсивність використання враховується через вагу ребра, яка відповідає кількості звернень до поля у межах методу. Це дозволяє поєднати структурну та кількісну інформацію в межах єдиної моделі.

Створення екземплярів стороннього типу в межах методу формує окремий вид залежності між методом і відповідним типом, що відображається у моделі ребром типу methodUsesInit. Такий зв’язок фіксує факт використання ініціалізатора стороннього типу в тілі методу, а його інтенсивність визначається кількістю викликів відповідного конструктора. Тому модель дозволяє не лише встановити наявність залежності, але й оцінити ступінь залучення стороннього типу у реалізацію методу.

Разом з тим можливі ситуації, коли метод не створює екземплярів стороннього типу, однак активно взаємодіє з ним через використання його статичних властивостей. У такому випадку формується зв'язок іншого типу, який позначається ребром methodUsesTypePropertyOrMethod. Це дозволяє врахувати більш широкий спектр взаємодій між компонентами, наприклад доступ до функціональності типу.

Окремо слід враховувати використання сторонніх типів у сигнатурі методу, що також є важливим аспектом архітектурних залежностей. Зокрема, якщо тип використовується серед параметрів методу, це відображається за допомогою ребра типу methodUsesTypeInParams, тоді як використання типу як типу повернення позначається ребром methodUsesTypeAsReturnType. Такі зв’язки дозволяють врахувати залежності, що не проявляються безпосередньо у тілі методу, але визначають його інтерфейс і, відповідно, впливають на взаємодію з іншими компонентами системи.

Для забезпечення коректного порівняння інтенсивності різних зв’язків у моделі використовується нормалізація ваги ребер. Зокрема, фактична вага ребра обчислюється як функція від кількості звернень до відповідного компонента із застосуванням сигмоїдного перетворення:

$$w_n = \sigma\left(\frac{call_{c_n}}{a} \times 12 - 6\right),$$

де w_n — вага ребра, що з’єднує поточний вузол із вузлом, який відповідає компоненту c_n, $call_{c_n}$ — кількість звернень до цього компонента, a — параметр, що задає характерну шкалу значень, а σ — сигмоїдна функція. Використання такого перетворення дозволяє обмежити значення ваг у інтервалі (0;1) з урахуванням того,

що $call_{c_n} \in N$ та зменшити вплив екстремально великих значень, що є типовими для реальних програмних систем.

Застосування нормалізації також забезпечує узгодженість представлення зв'язків різної природи та дозволяє використовувати отримані ваги як універсальну характеристику інтенсивності взаємодії між компонентами. Вона є особливо важливою у контексті подальшого використання моделі для візуалізації, де ваги безпосередньо впливають на просторове розміщення елементів.

Наша модель передбачає й можливість розширення шляхом додавання нових типів ребер для більш детального представлення залежностей. Така гнучкість забезпечує адаптивність підходу до специфіки різних мов програмування та задач аналізу, а також створює передумови для подальшого розвитку моделі без необхідності її принципової перебудови.

2.3 Автоматизована побудова графової моделі для мови Swift

Під час побудови будь-якої моделі архітектури програмного модуля на основі вихідного коду виникає низка проблем, пов'язаних із еволюцією сучасних мов програмування. Зокрема, нові версії однієї й тієї самої мови можуть використовувати різні синтаксичні специфікації або змінювати окремі правила інтерпретації конструкцій. Унаслідок цього рішення, що безпосередньо ґрунтуються на аналізі тексту вихідного коду, є доволі крихкими, оскільки можуть частково або повністю втрачати коректність при переході до нових версій мови [50].

Дещо стійкішим підходом є аналіз не самого тексту коду, а не понять, похідних від нього, наприклад абстрактних синтаксичних дерев. У цьому випадку ми можемо абстрагуватися від частини неістотних синтаксичних відмінностей, однак повністю усунути залежність від змін специфікації мови він не може. Крім того, у випадку самостійного розбору абстрактного синтаксичного дерева розробник системи бере на себе відповідальність за підтримку механізмів інтерпретації конструкцій мови, коректне розпізнавання символів та узгодження результатів аналізу з фактичною семантикою коду.

Важливим є й визначення саме тих частин вихідного коду, які реально використовуються у проєкті. Сам факт наявності файлу з програмним кодом у директорії модуля не гарантує, що його вміст бере участь у процесі компіляції або формуванні кінцевого програмного продукту. Хоча додаткові витрати ресурсів на аналіз невикористовуваного коду вже самі по собі є небажаними, серйознішою проблемою є потрапляння зайвих компонентів та зв'язків до кінцевої моделі архітектури.

Не менш важливим є також коректне врахування областей імен. Залежно від контексту одна і та сама назва може відповідати різним сутностям мови. У разі прямої роботи з вихідним кодом або навіть із сирими синтаксичними структурами розробник аналізатора змушений самостійно забезпечувати правильне трактування, що суттєво підвищує складність реалізації та ризик помилок.

Під час розроблення нового аналізатора для нашої системи зазначені ризики було враховано, у зв’язку з чим було прийнято рішення використовувати граф символів (symbol graph) для мови Swift, описаний у бібліотеці SymbolKit, як основу для автоматизованої побудови моделі візуалізації.

Граф символів цієї бібліотеки являє собою структуру, що моделює орієнтований граф, вершинами якого є мовні символи, тобто декларації, а ребрами — визначені види зв'язків між ними. Сукупність символів представлена у вигляді словника symbols, де ключем є унікальний ідентифікатор символу, а значенням — структура, що містить інформацію про його декларацію (вид, рівень доступу, конкретне розташування у вихідному коді, тощо). Відомості про зв’язки між символами подано у вигляді масиву структур relationships, які містять ідентифікатори символів, що беруть участь у зв’язку, а також інформацію про тип цього зв’язку.

Граф символів може бути згенерований компілятором Swift за умови використання відповідних параметрів компіляції. Це дозволяє значною мірою абстрагуватися від особливостей конкретного виду модуля та його призначення, а також забезпечує узгодженість аналізу з актуальною специфікацією мови. Використання графа символів SymbolKit дає змогу відсіяти код, який фактично не

використовується, та гарантувати унікальність типів завдяки наявності унікальних ідентифікаторів, що вже враховують нюанси роботи областей імен.

Водночас слід підкреслити, що граф символів не є повністю тотожним графовій моделі архітектури, запропонованій у нашому дослідженні. Symbol graph відображає декларації та зв'язки між ними на рівні символів мови, тоді як розроблювана тут модель описує типи даних і їх складові саме в контексті архітектури програмного модуля. Отже, граф символів виступає не кінцевим представленням, а основою для побудови більш спеціалізованої моделі.

Аналізатор виконує побудову моделі архітектури у три етапи. На першому етапі за допомогою відповідних команд виконується компіляція модуля з генерацією графа символів. Далі символи, що становлять інтерес у контексті запропонованої моделі, відображаються у вершини орієнтованого зваженого графа відповідно до заздалегідь визначених правил класифікації.

Інформація про використання типів у сигнатурі методів отримується безпосередньо з метаданих відповідного символу SymbolKit. Наступним етапом є наповнення моделі зв'язками на основі масиву relationships, що може бути виконане за один прохід.

Отримання зв'язків типу methodOf у цьому випадку є фактично тривіальним, оскільки граф символів містить зв'язки типу memberOf та optionalMemberOf. Зв'язок такого типу від символу, який у межах розробленої моделі розглядається як метод, до символу, який інтерпретується як тип, безпосередньо відповідає зв'язку methodOf. Використання SymbolKit додатково спрощує побудову цих зв'язків, оскільки дозволяє коректно визначати належність типу навіть для методів, задекларованих у розширеннях.

Аналогічно зв'язки memberOf та optionalMemberOf між символами, що відповідають вузлам типів p та dt, інтерпретуються як наявність зв'язку propertyOf між ними. Зв'язок isOfType, у свою чергу, може бути побудований для символів типів property і typeProperty на основі їхніх метаданих. Інформацію про наявність наслідування між типами також отримуємо із зв'язків типу inherits між відповідними символами.

Оскільки SymbolKit працює на рівні декларацій, для аналізу звернень до полів, методів і типів із тіл методів та функцій необхідно виконати додатковий аналіз абстрактного синтаксичного дерева. Цей третій етап реалізується за допомогою бібліотеки SwiftSyntax із використанням шаблону visitor.

Відповідність між видами символів SymbolKit та вузлами запропонованої візуальної моделі наведено в табл. 2.1.

Таблиця 2.1 – Відповідність видів символів SymbolKit вузлам візуальної моделі

Тип вузла	Назва типу вузла	Вид символа в термінах SymbolKit
dt	Data type	class, struct, enum
m	Method	method, typeMethod, subscript, typeSubscript, operator, init, deinit
p	Property	property, typeProperty

У результаті роботи аналізатора отримуємо зважений орієнтований граф, який після розміщення у тривимірному просторі з використанням силового алгоритму може бути відображений за допомогою спеціалізованого інструмента, як показано на рис. 2.3.

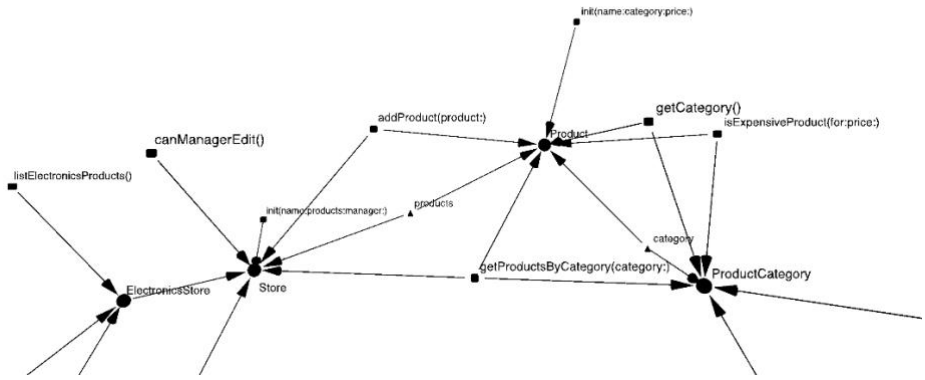


Рис. 2.3 Приклад згенерованої графової моделі архітектури

Наголосимо, що підсистема аналізатора є мовно-специфічною: використання SymbolKit і SwiftSyntax зумовлено особливостями екосистеми Swift і не може бути

безпосередньо застосовано для інших мов. Водночас запропонована графова модель (типи вузлів dt/m/p, система ребер і ваг), алгоритм виявлення аномалій та підсистема візуалізації є мовно-незалежними. Портування підходу на іншу мову програмування потребує лише реалізації відповідного аналізатора, який виконує аналогічне відображення мовних конструкцій у вузли та ребра запропонованого графа. Наприклад, для мови Java вузлам dt відповідатимуть класи та інтерфейси, вузлам m — методи, вузлам p — поля; як джерело даних може використовуватись JavaParser або аналогічний інструмент статичного аналізу.

2.4 Приклад застосування графової моделі

Використання запропонованої моделі для візуалізації зв’язків між типами та методами дає змогу не лише наочно представити структуру взаємодій, але й глибше аналізувати причини формування конкретних значень метрик, пов’язаних зі зв’язністю компонентів. Зокрема, регулярне обчислення метрики зв’язності між об’єктами (Coupling Between Objects, CBO) у межах програмного модуля може використовуватися як індикатор. У разі зростання максимального або середнього значення CBO виникає потреба у більш детальному аналізі, який доцільно здійснювати з використанням візуальної моделі — для безпосередньої локалізації джерел підвищеної зв'язності.

Водночас сама метрика CBO, як і інші показники відповідного набору метрик, надає лише узагальнену оцінку на рівні типів і не дозволяє визначити структуру залежностей на рівні окремих методів. у цьому контексті запропонована нами графова модель може розглядатися як засіб деталізації таких оцінок, оскільки дозволяє безпосередньо пов’язати значення метрик із конкретними елементами реалізації.

Завдяки цьому модель може бути використана не лише для візуалізації, але й для аналізу архітектури з метою виявлення характерних ознак поганого проєктування (code smells) [53], [54]. Зокрема, аномальна кількість або структура зв’язків окремих вузлів може свідчити про потенційні порушення принципу єдиної відповідальності (Single Responsibility Principle), що було продемонстровано у [55].

Значна інтенсивність взаємодії методу зі стороннім типом може свідчити про наявність таких антипатернів, як «заздрісна функція» (Feature Envy) або «недоречна близькість» (Inappropriate Intimacy) [56], [57]. Характерною ознакою у цьому випадку є надмірне використання методів і полів іншого типу в межах реалізації певного методу, зокрема допоміжних або службових.

У нашій моделі така ситуація відображається у вигляді ребра з відносно великою вагою, що з'єднує вузол типу m із вузлом типу dt, який не пов'язаний із цим методом ребром типу methodOf. Можливість автоматизованого візуального виявлення подібних аномалій залежить від масштабу системи та вираженості проблеми, однак відповідні характеристики можуть бути також предметом формалізованого аналізу. Вони відкривають можливість створення інструментів автоматизованого пошуку таких ознак неякісного коду.

Розглянемо застосування запропонованої нами моделі візуалізації на прикладі конкретного фрагмента вихідного коду (лістинг 1). У наведеному прикладі структура Task містить як статичні властивості, так і поля екземпляра, тоді як конструктор із повним набором параметрів генерується компілятором автоматично. Клас TaskManager, у свою чергу, містить поле tasks, що є масивом структур Task, а також два методи: addTask(title:description:) та createNewTask(). Конструктор без параметрів також формується автоматично.

Очевидною є значна кількість звернень до типу Task у межах реалізації методу createNewTask(). Водночас залежність цього методу від внутрішніх компонентів класу TaskManager є суттєво меншою, що дає підстави говорити про наявність у даному випадку антипатерни типу «заздрісна функція». Слід також зауважити, що використання полів defaultTitle та defaultDescription може інтерпретуватися і як прояв «недоречної близькості».

```
struct Task {

    static let defaultTitle = "Default title"
    static let defaultDescription = "Default description"

    var title: String
    var description: String
    var isCompleted: Bool = false
}

class TaskManager {

    private var tasks: [Task] = []

    func addTask(title: String, description: String) {
        let task = Task(title: title, description: description)
        self.tasks.append(task)
```

Лістинг 1. Фрагмент вихідного коду мовою Swift

У результаті автоматизованої побудови візуального представлення архітектури отримано тривимірну модель, яку наведено на рис. 2.4. Вузли типу dt позначено колами, вузли типу m — квадратами, а вузли типу p — трикутниками. Різні типи зв’язків відображаються за допомогою різних типів стрілок та градацій сірого кольору.

Для підвищення наочності у випадках, коли між двома вузлами існує кілька зв’язків різної природи, вони агрегуються в одне ребро з накопиченням ваги та збереженням інформації про всі типи зв’язків у вигляді метаданих. У розглянутому прикладі ребро reduced_connection, що з’єднує вузол m (метод createNewTask()) із вузлом dt (структура Task), є результатом згортання зв’язків типів methodUsesTypePropertyOrMethod (вага 2), methodUsesInit (вага 1) та methodUsesTypeAsReturnType (вага 1).

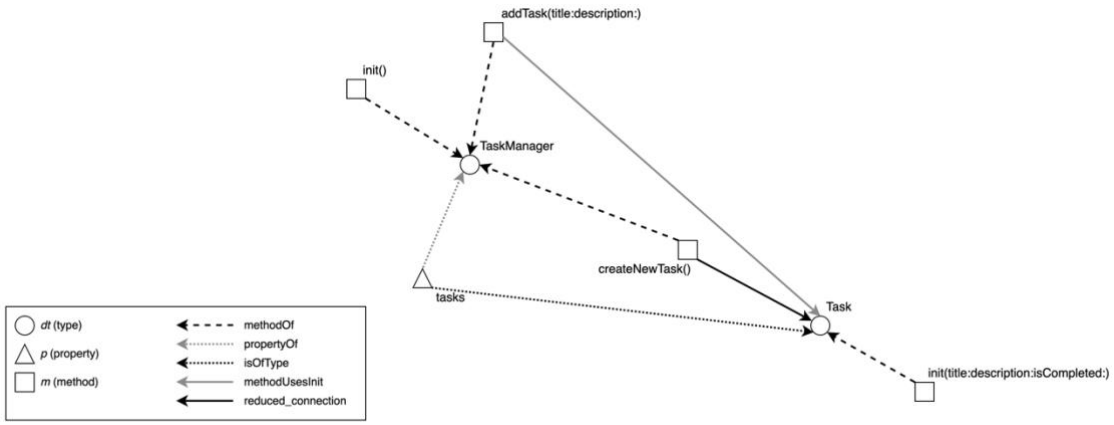


Рис. 2.4 Деталізація відображення арзітектури програми графічними засобами

Оскільки для побудови візуального представлення використовується силовий алгоритм розміщення, вага ребра безпосередньо впливає на просторове розташування вузлів. Зокрема, більша вага зв’язку призводить до зменшення відстані між відповідними елементами у тривимірному просторі, тоді як слабші зв’язки, навпаки, відображаються більшою віддаленістю.

У результаті вузол m (метод createNewTask()) розташовується значно ближче до вузла dt (структура Task), ніж до вузла dt (клас TaskManager), якому цей метод належить. Така просторова конфігурація є наочною ознакою наявності антипатерна типу «заздрісна функція» у візуальному представленні моделі.

Варто зазначити, що подібну проблему складно виявити, спираючись виключно на значення метрики CBO, оскільки вона є інформативною переважно на рівні класів і не враховує структуру залежностей на рівні окремих методів.

2.5 Висновки

Графова модель архітектури — центральний інструмент, запропонований у цій роботі для формалізації та аналізу залежностей між компонентами програмного модуля. Показано, що графове представлення дозволяє перейти від текстового опису вихідного коду до формалізованої структури, придатної як для алгоритмічної обробки, так і для подальшої візуалізації. На відміну від більш грубих представлень, запропонована модель враховує не лише наявність зв’язків між

компонентами, але й напрям та інтенсивність залежностей — завдяки орієнтованим зваженим ребрам.

У межах розробленої моделі виділено три типи вузлів, що відповідають типам даних, полям та методам, а також визначено систему зв’язків, яка дозволяє відображати як декларативні, так і поведінкові залежності між елементами програмного забезпечення. Показано, що таке представлення є достатньо загальним для адаптації до різних мов програмування, а для мови Swift може бути безпосередньо узгоджене з реальними мовними конструкціями.

Обґрунтовано доцільність використання ваг ребер як узагальненого представлення інтенсивності взаємодій між компонентами. Зазначено, що таке подання створює природний зв’язок між графовою моделлю та кількісними характеристиками архітектури, зокрема метриками зв’язності. При цьому ваги ребер можуть бути безпосередньо використані не лише у формальному аналізі, але й у просторовому розміщенні вузлів під час візуалізації, що поєднує структурну точність із наочністю подання.

Окрему увагу приділено автоматизованій побудові моделі для мови Swift. Показано, що прямий аналіз тексту вихідного коду є нестійким до еволюції мови та пов’язаний із низкою технічних труднощів, зокрема коректним врахуванням областей імен та відсіканням невикористовуваного коду. У зв’язку з цим обґрунтовано використання графа символів SymbolKit як проміжного представлення, що дозволяє підвищити стійкість аналізу, спростити побудову зв’язків і забезпечити коректне відображення фактично використовуваних елементів програмного модуля.

На прикладі конкретного фрагмента коду продемонстровано, що запропонована модель придатна не лише для загального представлення архітектури, але й для виявлення конкретних проблем проєктування.

Запропонована орієнтована зважена графова модель з трьома типами вузлів (dt, m, p) і шістьма класами ребер придатна для автоматизованої побудови на основі графа символів Swift-компілятора та безпосередньо використовується як вхід для аналізу антипатернів і просторової візуалізації.

РОЗДІЛ 3

ПОШУК АНТИПАТЕРНІВ В АРХІТЕКТУРІ ПРОГРАМНОГО МОДУЛЯ

3.1 Загальні підходи та принципи виявлення аномалій в ООП

Архітектура програмного забезпечення визначає структуру системи, її основні компоненти та взаємозв'язки між ними, а також принципи організації і розвитку цих компонентів у часі. Від якості архітектурних рішень безпосередньо залежать такі характеристики системи, як модифікованість, розширюваність, зрозумілість і супроводжуваність. У процесі еволюції програмного забезпечення ці властивості можуть погіршуватися внаслідок появи відхилень у структурі або поведінці системи, які прийнято називати архітектурними аномаліями.

Під архітектурними аномаліями розуміються структурні або поведінкові відхилення в архітектурі програмної системи, що призводять до зниження її якісних характеристик. У літературі такі явища також описуються термінами architectural smells, design erosion або architecture degradation symptoms [30], [31], [58]. Важливою особливістю є те, що ці відхилення не завжди проявляються як помилки у традиційному розумінні, а радше як небажані структурні властивості, які ускладнюють подальший розвиток системи.

На відміну від code smells, що характеризують локальні проблеми на рівні окремих елементів коду (методів, класів), архітектурні аномалії проявляються на рівні взаємодії компонентів і охоплюють більші фрагменти системи. Це можуть бути як окремі підсистеми, так і вся система в цілому. Дослідження показують, що такі аномалії мають суттєво більший вплив на довгострокову еволюцію програмного забезпечення, оскільки вони визначають глобальну структуру залежностей між компонентами [31], [59].

Архітектурні аномалії тісно пов'язані з процесом деградації архітектури (architecture erosion), який полягає у поступовому розходженні між початково запроєктованою архітектурою та фактичною реалізацією системи. Така деградація є типовим наслідком довготривалого розвитку програмного забезпечення, особливо

в умовах частих змін вимог, відсутності системного контролю архітектури та накопичення технічного боргу. З часом це призводить до появи складних і важко інтерпретованих залежностей, що ускладнюють аналіз системи.

З формальної точки зору, архітектурну аномалію доцільно розглядати як відхилення від бажаних властивостей архітектури. До таких властивостей, зокрема, належать низька зв’язаність (low coupling), висока згуртованість (high cohesion) та розділення відповідальностей (separation of concerns). Відхилення від цих принципів призводить до появи надлишкових або нестабільних залежностей, зниження модульності та ускладнення повторного використання компонентів. Відповідно, аномалії можуть розглядатися як індикатори порушення базових принципів проєктування.

Окремо слід підкреслити, що архітектурні аномалії не є бінарним явищем (наявність або відсутність), а мають кількісний характер. Тобто ступінь їх прояву може варіюватися залежно від інтенсивності зв’язків, розміру компонентів або їх ролі у системі. Це ускладнює їх формальне визначення та потребує використання кількісних підходів, зокрема метрик, для оцінювання рівня відхилення від бажаного стану.

Частина архітектурних аномалій має усталені форми та описується у вигляді антипатернів, які представляють собою типові, повторювані помилки проєктування. Водночас не всі аномалії можуть бути однозначно віднесені до відомих антипатернів, що обумовлює необхідність більш загального підходу до їх виявлення та аналізу.

З огляду на це архітектурні аномалії є ключовим індикатором якості програмної архітектури та важливим об’єктом дослідження. Їх виявлення та аналіз створюють основу для подальшого застосування формалізованих методів оцінювання, зокрема метрик, а також для автоматизації процесу аналізу архітектури програмного забезпечення.

Виявлення архітектурних аномалій неможливе без визначення базових критеріїв, відносно яких оцінюється якість програмної архітектури. У контексті об’єктно-орієнтованого програмування такими критеріями виступають

фундаментальні принципи проєктування, що визначають бажані властивості структури системи та взаємодії її компонентів.

Одним із найбільш поширених узагальнень цих принципів є набір SOLID, запропонований Robert C. Martin [60], [61]. Хоча ці принципи формулюються переважно на рівні окремих класів та інтерфейсів, їх вплив поширюється на архітектуру системи в цілому, оскільки саме вони визначають характер зв’язків між компонентами.

Принцип єдиної відповідальності (Single Responsibility Principle) передбачає, що кожен компонент повинен відповідати лише за одну функціональну область. Порушення цього принципу призводить до появи складних елементів, які акумулюють різнорідну логіку, що ускладнює їх аналіз та повторне використання. У свою чергу, принцип відкритості/закритості (Open/Closed Principle) визначає, що система має бути відкритою для розширення, але закритою для модифікації. Його недотримання проявляється у необхідності змінювати існуючі компоненти при додаванні нової функціональності, що створює додаткові залежності та підвищує ризик появи помилок.

Принцип підстановки Барбари Лісков (Liskov Substitution Principle) вимагає, щоб підкласи могли використовуватися замість базових класів без порушення коректності роботи системи. Його порушення призводить до неузгодженості поведінки компонентів та ускладнює побудову стабільних абстракцій. Принцип розділення інтерфейсів (Interface Segregation Principle) орієнтує на створення вузьких, спеціалізованих інтерфейсів замість універсальних. У разі його порушення формуються надлишкові залежності, коли компоненти змушені використовувати функціональність, яка їм фактично не потрібна.

Окремо слід відзначити принцип інверсії залежностей (Dependency Inversion Principle), який передбачає, що залежності мають спрямовуватися від абстракцій, а не від конкретних реалізацій. Порушення цього принципу є однією з ключових причин жорсткої зв’язаності компонентів, що ускладнює зміну або заміну окремих частин системи без впливу на інші.

Поряд із принципами SOLID важливу роль відіграють більш загальні принципи архітектурного проєктування, зокрема низька зв’язаність (low coupling), висока згуртованість (high cohesion) та розділення відповідальностей (separation of concerns). Вони визначають бажану форму графа залежностей між компонентами системи: обмежену кількість зв’язків, їх локалізацію в межах окремих підсистем та відсутність надлишкових або циклічних залежностей.

З точки зору аналізу архітектури зазначені принципи можуть бути інтерпретовані як еталонний стан системи, відносно якого визначаються аномалії. Відповідно, архітектурна аномалія розглядається як відхилення від одного або декількох принципів проєктування. Наприклад, надмірна кількість зв’язків між компонентами свідчить про порушення принципу низької зв’язаності, тоді як концентрація великої кількості залежностей в одному елементі може вказувати на порушення принципу єдиної відповідальності.

Важливо підкреслити, що більшість зазначених принципів мають якісний характер і не можуть бути безпосередньо виміряні. Це обумовлює необхідність переходу до формалізованих підходів, які дозволяють кількісно оцінювати ступінь їх дотримання. У цьому контексті метрики програмної архітектури виступають засобом трансформації абстрактних принципів у числові показники, що робить можливим автоматизований аналіз.

Зокрема, принципи об’єктно-орієнтованого проєктування формують концептуальну основу для виявлення архітектурних аномалій, визначаючи критерії оцінювання якості системи. Їх подальша формалізація у вигляді метрик забезпечує можливість переходу до кількісного аналізу.

Архітектурні аномалії можуть проявлятися у різних формах залежно від рівня абстракції, на якому розглядається програмна система, а також від характеру порушень у її структурі або поведінці. Тому доцільно розглядати їх у рамках узагальненої класифікації, що дозволяє систематизувати підходи до їх аналізу та виявлення.

Одним із найбільш поширених підходів є поділ архітектурних аномалій на структурні, поведінкові та еволюційні. Така класифікація дозволяє врахувати як статичні характеристики системи, так і динаміку її розвитку.

Структурні аномалії пов’язані з організацією залежностей між компонентами системи. Вони проявляються у вигляді надлишкових або неефективних зв’язків, що ускладнюють модульність та зрозумілість архітектури. До таких аномалій належать, зокрема, циклічні залежності між модулями, надмірна централізація залежностей в окремих компонентах, а також висока щільність зв’язків між підсистемами. Подібні відхилення безпосередньо впливають на зв’язаність системи та ускладнюють її подальшу модифікацію.

Поведінкові аномалії характеризують особливості взаємодії компонентів у процесі виконання системи. Вони можуть проявлятися у вигляді неефективних або неочевидних сценаріїв взаємодії, надмірної залежності між логічно незалежними частинами системи або порушення очікуваної ролі окремих компонентів. Такі аномалії часто важче виявити на основі лише статичного аналізу, однак вони також можуть бути частково інтерпретовані через структуру залежностей.

Еволюційні аномалії пов’язані з процесом розвитку програмного забезпечення та відображають зміни у його архітектурі з часом. До них відносяться явища архітектурної деградації та дрейфу, що виникають у результаті поступового накопичення змін, які не узгоджуються з початковими проєктними рішеннями. Такі аномалії зазвичай не є результатом одноразового порушення, а формуються поступово, що ускладнює їх своєчасне виявлення.

Важливо відзначити, що наведені класи аномалій не є взаємовиключними. На практиці одна і та ж проблема може проявлятися одночасно у кількох аспектах. Наприклад, циклічні залежності між модулями є структурною аномалією, але водночас можуть впливати на поведінку системи та бути наслідком її еволюції. Це свідчить про необхідність комплексного підходу до аналізу архітектури.

З точки зору формалізації, найбільш придатними для автоматизованого аналізу є структурні аномалії, оскільки вони безпосередньо відображаються у вигляді графа залежностей між компонентами системи. Це дозволяє застосовувати методи

теорії графів та кількісні метрики для їх виявлення. Водночас поведінкові та еволюційні аномалії часто потребують додаткових джерел інформації або складніших моделей аналізу.

Зрозуміло, що основну увагу ми приділили структурним аномаліям, які можуть бути формалізовані у вигляді графової моделі та проаналізовані із використанням метрик.

3.2 Метрики оцінювання архітектури програмного забезпечення

У більшості випадків метрики програмного забезпечення виступають як засіб формалізації властивостей архітектури. Під метрикою розуміється функція, що відображає характеристики програмної системи або її компонентів у числові значення. Таке відображення дозволяє порівнювати різні частини системи між собою, відстежувати зміни у часі та виявляти відхилення від очікуваних характеристик. Формально метрика може бути представлена як відображення:

$$f: S \rightarrow \mathbb{R},$$

де S – множина елементів програмної системи або їх характеристик.

Використання метрик має декілька ключових переваг. По-перше, вони забезпечують об’єктивність оцінювання, оскільки базуються на формалізованих правилах, а не на суб’єктивних судженнях розробників. По-друге, метрики дозволяють здійснювати повторюваний аналіз, що є важливим для контролю якості в процесі еволюції програмного забезпечення. По-третє, вони створюють основу для автоматизації аналізу, оскільки можуть бути обчислені без безпосереднього втручання людини.

Водночас важливо підкреслити, що метрики не є безпосереднім показником якості програмного забезпечення. Вони відображають лише окремі аспекти архітектури та повинні інтерпретуватися у контексті конкретної системи. Наприклад, велике значення певної метрики може свідчити як про наявність проблеми, так і про особливості доменної області або архітектурного рішення. Саме

тому, метрики слід розглядати не як абсолютні критерії, а як індикатори, що сигналізують про потенційні відхилення.

З позиції аналізу архітектури метрики дозволяють перейти від абстрактних принципів проєктування до їх кількісного представлення. Наприклад, принцип низької зв'язаності може бути інтерпретований через кількість залежностей між компонентами, а принцип високої згуртованості — через ступінь взаємозв'язку між елементами всередині компонента. Такий перехід є необхідним для подальшого автоматизованого аналізу, оскільки дозволяє застосовувати математичні та статистичні методи.

Разом із тим, використання метрик пов'язане з рядом обмежень. Зокрема, більшість метрик оцінюють властивості окремих компонентів і не враховують глобальну структуру системи. Їхні значення можуть суттєво залежати від масштабу системи та стилю проєктування. Це ускладнює інтерпретацію результатів та обмежує можливості використання метрик як єдиного інструменту аналізу [62], [63], [64].

Відповідно, метрики програмного забезпечення є інструментом формалізованого оцінювання архітектурних властивостей та виявлення потенційних аномалій. Водночас їх застосування потребує обережної інтерпретації та доповнення іншими підходами, що враховують структурні особливості системи.

Одним із найбільш поширених підходів до кількісного оцінювання архітектури є використання метрик об'єктно-орієнтованого дизайну, зокрема набору, запропонованого Чідамбером і Кемерером (Chidamber & Kemerer) [51]. Ці метрики отримали широке застосування завдяки своїй простоті, інтерпретованості та можливості автоматизованого обчислення на основі вихідного коду [65], [66].

Метрика WMC (Weighted Methods per Class) характеризує складність класу через кількість і, за потреби, складність його методів. У загальному випадку вона визначається як сума складностей усіх методів класу:

$$\mathrm{WMC} = \Sigma\, c_i,$$

де c_i — складність і-го методу. Як правило, у ролі c_i може використовуватися цикломатична складність або інший показник, що відображає внутрішню структуру методу.

У найпростішому випадку, коли всі методи вважаються рівнозначними та їх складність приймається рівною одиниці ($c_i = 1$), метрика зводиться до підрахунку кількості методів у класі:

$$\mathrm{WMC} = n,$$

де n — кількість методів.

Інтуїтивно це відповідає обсягу функціональності, інкапсульованої в одному компоненті. Високі значення WMC можуть свідчити про надмірну концентрацію логіки, що ускладнює тестування, супроводження та повторне використання. Наприклад, клас, який реалізує одночасно бізнес-логіку, обробку даних і взаємодію з користувачем, як правило, має високе значення WMC і є потенційним кандидатом на декомпозицію.

Метрика RFC (Response For a Class) оцінює кількість методів, які можуть бути викликані у відповідь на звернення до об’єкта класу. Вона включає як власні методи класу, так і методи інших класів, що викликаються з нього. У спрощеному вигляді RFC може бути визначена як:

$$\mathrm{RFC} = |M| + |R|,$$

де M — множина методів, визначених у самому класі, а R — множина унікальних методів інших класів, що викликаються безпосередньо з методів цього класу.

У більш загальному випадку, з урахуванням транзитивних викликів, використовується розширене визначення:

$$\mathrm{RFC}' = |M| + |R'|,$$

де R' — множина всіх унікальних методів, що можуть бути досягнуті через ланцюг викликів, починаючи з методів класу.

RFC відображає ступінь взаємодії компонента з оточенням і слугує показником зв'язаності. Високі значення RFC свідчать про складні сценарії виконання та наявність великої кількості залежностей, що підвищує ризик виникнення каскадних

змін: модифікація одного методу може вплинути на значну кількість інших компонентів.

Метрика NOC (Number of Children) визначає кількість безпосередніх підкласів для заданого класу. Вона характеризує положення компонента в ієрархії наслідування та може використовуватися для оцінки ступеня повторного використання. З одного боку, велике значення NOC може свідчити про ефективне використання механізмів наслідування. З іншого боку, це також може вказувати на надмірну залежність від базового класу, що робить його критичним елементом системи: зміни в такому класі можуть впливати на значну кількість підкласів.

Метрика LOC (Lines of Code), хоча і не є специфічною для об’єктно-орієнтованого підходу, широко використовується як базовий показник розміру компонента. Вона дозволяє оцінити масштаб реалізації та використовується як допоміжний фактор при інтерпретації інших метрик. Наприклад, однакові значення WMC для двох класів можуть мати різний зміст залежно від їх розміру: у невеликому класі це може свідчити про перевантаження, тоді як у великому — бути нормальним явищем.

Важливо підкреслити, що наведені метрики не повинні розглядатися ізольовано. Кожна з них відображає лише окремий аспект архітектури, і лише їх спільний аналіз дозволяє отримати більш повне уявлення про структуру системи. Наприклад, поєднання високих значень WMC та RFC може свідчити про наявність складного і сильно зв’язаного компонента, що є характерною ознакою архітектурної аномалії.

Тобто метрики об'єктно-орієнтованого дизайну створюють основу для кількісного аналізу архітектури, дозволяючи оцінювати складність, зв’язаність та структуру компонентів системи. Водночас їх ефективне використання потребує подальшого розгляду питань інтерпретації та взаємозв’язку між різними показниками.

Поряд із базовими метриками об’єктно-орієнтованого дизайну важливе значення для аналізу архітектури мають характеристики згуртованості (cohesion) та зв’язаності (coupling), які визначають якість структури програмної системи. Ці властивості безпосередньо пов’язані з принципами проєктування, розглянутими у

попередньому підрозділі, і виступають одними з ключових індикаторів наявності архітектурних аномалій.

Згуртованість характеризує ступінь взаємозв'язку між елементами всередині одного компонента. Висока згуртованість означає, що всі частини компонента спрямовані на виконання спільної функції, тоді як низька згуртованість свідчить про наявність декількох незалежних функціональних областей у межах одного класу або модуля. Наприклад, якщо методи класу використовують різні набори даних і не мають спільної логічної основи, це може вказувати на порушення принципу єдиної відповідальності.

Для кількісного оцінювання згуртованості використовується метрика LCOM (Lack of Cohesion in Methods) [26], [51], [67], [68]. Її ідея полягає у визначенні ступеня зв'язку між методами класу через спільно використовувані поля. Зокрема, розглядаються пари методів, що мають спільні змінні стану, та пари, які не мають таких зв'язків.

Формально метрика LCOM може бути визначена як:

$$\mathrm{LCOM} = \max(\mathrm{P} - \mathrm{Q}, 0),$$

де P — кількість пар методів, що не мають спільних полів, Q — кількість пар методів, що мають принаймні одне спільне поле.

Якщо значна частина методів не пов'язана між собою через спільні дані (тобто $P > Q$), це свідчить про низьку згуртованість. Таким чином, високе значення LCOM вказує на слабкий внутрішній зв'язок між елементами компонента і може розглядатися як ознака архітектурної аномалії.

На відміну від згуртованості, зв'язаність характеризує ступінь залежності компонента від інших частин системи. Висока зв'язаність означає, що компонент активно взаємодіє з великою кількістю інших компонентів, що ускладнює його зміну та повторне використання. Наприклад, клас, який викликає значну кількість методів інших класів або залежить від їх внутрішньої структури, є сильно зв'язаним.

Зв'язаність може оцінюватися за допомогою різних метрик, зокрема RFC або CBO (Coupling Between Objects). Метрика CBO визначається як кількість

унікальних класів, з якими пов’язаний даний клас, і може бути представлена у вигляді:

$$\mathrm{CBO} = |C|,$$

де C — множина класів, від яких безпосередньо залежить даний клас.

Метрика RFC, у свою чергу, враховує не лише кількість залежностей, але й потенційний обсяг викликів, що виникають у процесі виконання. Разом ці показники дозволяють оцінити, наскільки компонент інтегрований у систему та наскільки його поведінка залежить від інших елементів.

Високі значення метрик зв’язаності свідчать про складну мережу залежностей, що може бути ознакою порушення принципу інверсії залежностей або розділення відповідальностей.

Важливо, що згуртованість і зв’язаність слід розглядати у взаємозв’язку. Ідеальна архітектура передбачає високу згуртованість компонентів при одночасно низькій зв’язаності між ними. У такому випадку кожен компонент виконує чітко визначену функцію і взаємодіє з іншими компонентами через обмежений набір залежностей. Порушення цього балансу, наприклад, у вигляді одночасно низької згуртованості та високої зв’язаності, є характерною ознакою архітектурних аномалій.

Разом із тим, як і у випадку інших метрик, значення показників згуртованості та зв’язаності не можуть бути інтерпретовані ізольовано. Наприклад, висока зв’язаність може бути виправданою у центральних компонентах системи, які виконують координаційну роль. Аналогічно, низька згуртованість може бути наслідком специфіки доменної області, а не помилкою проєктування. Це підкреслює необхідність врахування контексту при аналізі метрик.

Метрики згуртованості та зв'язаності доповнюють базові показники об'єктно-орієнтованого дизайну і дають змогу глибше аналізувати структуру системи. Вони відіграють ключову роль у виявленні архітектурних аномалій, проте їх ефективне застосування потребує подальшого розгляду питань інтерпретації та визначення критеріїв оцінювання.

Незважаючи на формалізований характер метрик, їх використання для аналізу архітектури пов’язане з низкою складнощів, основною з яких є інтерпретація отриманих значень. На відміну від строгих математичних критеріїв, більшість метрик програмного забезпечення не мають однозначного трактування, і їх значення повинні розглядатися у контексті конкретної системи.

Інтерпретація метрик ускладнюється також залежністю їх значень від масштабу системи. У великих програмних проєктах значення метрик, як правило, є вищими, ніж у невеликих, що не обов’язково свідчить про гіршу якість архітектури. Наприклад, клас у великій системі може мати більше залежностей лише через більшу кількість доступних компонентів, що робить пряме порівняння метрик між різними системами некоректним.

На практиці для аналізу програмного забезпечення часто використовуються порогові значення метрик, які визначають допустимі межі їх зміни. Такі пороги дозволяють спростити інтерпретацію результатів і автоматизувати процес виявлення потенційних проблем. Проте визначення універсальних порогових значень є складною задачею, оскільки вони значною мірою залежать від контексту застосування, доменної області та архітектурного стилю системи.

Наприклад, значення WMC, яке може вважатися прийнятним для невеликого сервісного класу, може бути недостатнім для компонента, що реалізує складну бізнес-логіку. Аналогічно, високий рівень зв’язаності може бути допустимим для центральних компонентів системи, які виконують координаційну функцію. Відповідно, використання фіксованих порогів без урахування контексту може призводити як до хибнопозитивних, так і до хибнонегативних результатів.

Доцільнішим є аналіз відносних значень метрик у межах конкретної системи. Замість використання фіксованих порогів доцільно розглядати відхилення значень метрик від середніх або типових для даної системи. Такий підхід дозволяє виявляти компоненти, що суттєво відрізняються від інших, і, відповідно, можуть бути джерелом архітектурних аномалій.

Разом із тим навіть відносний аналіз метрик не вирішує всіх проблем, пов’язаних з інтерпретацією. Зокрема, метрики, як правило, оцінюють властивості

окремих компонентів і не враховують глобальну структуру системи. Вони не відображають просторове розташування елементів у структурі залежностей і не дозволяють безпосередньо виявляти складні структурні патерни, такі як циклічні залежності або групи взаємопов’язаних компонентів.

Отже, метрики програмного забезпечення є ефективним інструментом для кількісного оцінювання окремих характеристик архітектури, проте їх використання має обмеження, пов’язані з інтерпретацією та відсутністю урахування глобальної структури системи. Це обумовлює необхідність доповнення метрик іншими підходами, зокрема використанням структурних моделей, які дозволяють аналізувати архітектуру на рівні взаємозв’язків між компонентами.

З огляду на обмеження метрик, пов’язані з відсутністю урахування глобальної структури системи, доцільним є використання підходів, що дозволяють аналізувати взаємозв’язки між компонентами. У попередньому розділі було показано, що така структура може бути формалізована у вигляді графової моделі, де програмна система представлена як множина елементів та залежностей між ними.

Тому метрики можуть бути інтерпретовані як локальні характеристики елементів графа, тоді як сама графова модель дозволяє аналізувати їх взаємне розташування та структурні властивості. Це відкриває можливість поєднання кількісного підходу, заснованого на метриках, із аналізом глобальної структури системи.

Зокрема, характеристики, що визначаються метриками, можуть бути співвіднесені з властивостями графа. Наприклад, кількість залежностей компонента відповідає степеню відповідної вершини, тоді як згуртованість може бути інтерпретована через щільність зв’язків у відповідному підграфі. Така оцінка дозволяє перейти від аналізу окремих значень до дослідження структурних патернів.

Використання графової моделі не замінює метрики, а доповнює їх, дозволяючи враховувати як локальні, так і глобальні характеристики архітектури. Це створює основу для застосування методів автоматизованого виявлення архітектурних аномалій, що враховують як числові показники, так і структуру системи.

3.3 Автоматизоване виявлення архітектурних аномалій на основі графової моделі

На основі графової моделі архітектури програмної системи стає можливим автоматизувати процес виявлення архітектурних аномалій. Ми розуміємо аномалії як відхилення у структурі графа, що відображає взаємозв'язки між компонентами системи.

Виявлення таких відхилень базується на аналізі характеристик вершин та підграфів, що відповідають окремим елементам архітектури. Зокрема, розглядаються показники зв'язаності, щільності зв'язків, а також структурні особливості, такі як наявність циклів або аномально пов'язаних груп компонентів. Подібні характеристики дозволяють інтерпретувати архітектурні аномалії як структурні патерни, що можуть бути формально визначені та автоматично виявлені.

Для виявлення таких відхилень ми запропонували використати спеціалізований компонент — детектор архітектурних аномалій, який здійснює аналіз графового представлення системи. Детектор виконує обробку структури залежностей між компонентами та застосовує набір критеріїв для виявлення потенційних відхилень від типових характеристик.

Ключовою особливістю запропонованої методології є поєднання аналізу структурних властивостей графа із використанням кількісних показників, отриманих на попередньому етапі. Це дозволяє враховувати як локальні характеристики компонентів, так і їх роль у загальній структурі системи.

Виявлення архітектурних аномалій зводиться до аналізу графа залежностей з метою пошуку елементів, характеристики яких суттєво відрізняються від типових для даної системи або порушують очікувані структурні властивості.

Побудова графа передбачає виділення ключових типів елементів, що мають значення для аналізу архітектури. До таких елементів належать, зокрема, класи, структури, методи та інші програмні компоненти, які можуть виступати як джерелами, так і отримувачами залежностей. Рівень деталізації моделі визначається поставленими задачами: у даній роботі основна увага приділялася

взаємозв’язкам між компонентами, що дозволило абстрагуватися від низькорівневих деталей реалізації.

Залежності між елементами формуються на основі аналізу різних типів взаємодій у коді. До них належать виклики методів, доступ до полів, використання типів у сигнатурах, а також відношення наслідування та реалізації інтерфейсів. Кожен такий зв’язок інтерпретується як ребро графа, що відображає напрямок залежності між компонентами.

У результаті формується орієнтований граф залежностей, який відображає структуру програмної системи на рівні взаємодії її елементів. Важливо, що така модель будується виключно на основі вихідного коду і не потребує додаткових артефактів, таких як документація або вручну створені архітектурні діаграми. Це забезпечує її актуальність та можливість застосування до реальних програмних систем без попередньої підготовки.

Отримане графове представлення є основою для подальшого аналізу, зокрема для виявлення структурних аномалій. Залежно від обраного рівня абстракції, одна і та ж система може бути представлена у вигляді різних графів, що відображають різні аспекти її архітектури. Ми використовуємо представлення, орієнтоване на аналіз залежностей між компонентами, що є найбільш релевантним для задачі виявлення архітектурних аномалій.

Аналіз графового представлення програмної системи дозволяє виявляти архітектурні аномалії як характерні відхилення у структурі залежностей між компонентами. На відміну від підходів, що базуються лише на значеннях метрик, у даному випадку аномалії визначаються через особливості розташування елементів у графі та характер їх взаємозв’язків.

Однією з найбільш очевидних ознак аномалій є наявність вершин із аномально високим ступенем. Такий показник відповідає компонентам, які мають велику кількість залежностей із іншими елементами системи. З практичної точки зору це можуть бути класи або модулі, що виконують роль централізованих вузлів взаємодії. Хоча в окремих випадках така роль є виправданою, надмірна концентрація залежностей часто свідчить про порушення принципу низької

зв’язаності та може призводити до ускладнення супроводження системи. Подібні елементи нерідко відповідають відомим архітектурним антипатернам, таким як “God Object”. Наприклад, у досліджуваних програмних системах подібні компоненти проявлялися як вузли графа, що мали значно більшу кількість вхідних і вихідних зв’язків порівняно з іншими елементами, фактично виконуючи роль централізованого координатора взаємодії.

Іншим типом структурних аномалій є слабка згуртованість компонентів, що проявляється у вигляді розріджених підграфів або наявності декількох слабо пов’язаних груп елементів усередині одного компонента. Така структура може свідчити про те, що компонент поєднує в собі декілька незалежних функціональних областей. У термінах метрик це відповідає високим значенням LCOM, однак у графовій моделі така аномалія проявляється через структуру внутрішніх зв’язків. У практичних прикладах це відповідало компонентам, у яких групи методів формували ізольовані підмножини зв’язків, що не мали спільної логічної основи.

Важливим класом аномалій є циклічні залежності між компонентами. У графовому представленні вони відповідають наявності циклів або сильно зв’язаних компонент (strongly connected components). Наявність таких структур ускладнює розуміння архітектури, створює труднощі при тестуванні та модифікації системи, а також обмежує можливості повторного використання окремих компонентів. Циклічні залежності часто виникають у процесі еволюції системи та є одним із ключових індикаторів деградації архітектури. У досліджуваних випадках такі залежності проявлялися як замкнені ланцюги взаємодії між кількома модулями, що унеможливлювало їх незалежне використання або заміну.

Окрім окремих вузлів і циклів, аномалії можуть проявлятися на рівні груп компонентів. Зокрема, щільно пов’язані підграфи можуть свідчити про формування неявних модулів або підсистем, які не відповідають початковому архітектурному розподілу. Такі структури можуть бути як позитивним явищем (наприклад, природне групування компонентів), так і ознакою порушення принципу розділення відповідальностей, якщо зв’язки виходять за межі логічних модулів. У практичних сценаріях подібні кластери відповідали групам компонентів, що мали високу

щільність взаємодії, але при цьому були розподілені між різними частинами системи без чіткої модульної структури.

Архітектурні аномалії у графовій моделі проявляються у вигляді характерних структурних патернів, що відображають відхилення від бажаних властивостей системи. Їх виявлення базується на аналізі як окремих характеристик вершин, так і глобальних властивостей графа — зокрема його діаметра, що визначається як довжина найдовшого найкоротшого шляху між вершинами і характеризує максимальну глибину ланцюжків залежностей у системі, — це дає комплекснішу картину стану архітектури, ніж аналіз лише окремих метрик.

Подальше формалізоване виявлення архітектурних аномалій може бути здійснене на основі статистичного аналізу характеристик графа залежностей. На відміну від підходів, що використовують фіксовані порогові значення метрик, у даному випадку аномалії визначаються як відхилення від типових значень, характерних для конкретної програмної системи.

У межах цього підходу кожному елементу графа ставиться у відповідність набір характеристик, зокрема ступінь вершини, вага ребра, коефіцієнт згуртованості або інші похідні показники. Частина цих характеристик може бути отримана безпосередньо зі структури графа, наприклад степінь вершини, тоді як інші потребують додаткового обчислення. Зокрема, для виявлення низької згуртованості важливим є не лише сам факт наявності зв’язку між елементом і класом, а й вага цього зв’язку, що відображає інтенсивність взаємодії.

Для виявлення ознак низької згуртованості між компонентами класу було вдосконалено підхід до побудови зваженої графової моделі. У початковій версії моделі A.D.A.R. ребра, що відображали зв’язок типу «є методом» або «є властивістю» певного класу, мали фіксовану вагу, що дорівнювала 1 в області значень [0;1]. Такий підхід дозволяв відобразити факт структурної належності елемента до класу, однак не давав змоги оцінити реальний рівень його згуртованості з цим класом.

Для усунення цього обмеження було запропоновано обчислювати вагу таких ребер на основі фактичного використання відповідного елемента іншими

компонентами класу. Оскільки методи та властивості мають різну природу взаємодії, формули для них визначаються окремо.

Для властивості p, що належить класу c, вага ребра між вузлом властивості та вузлом класу визначається так:

$$w(p,c) = \sigma(12 * \frac{m_{p,c}}{Mc} - 6),$$

де $w(p,c)$ – вага ребра між вузлом властивості p та класу c; $m_{p,c}$ – к-сть методів класу c, що використовують властивість p; Mc – загальна к-сть методів класу c, σ — сигмоїдна функція, що нормалізує значення в межах інтервалу [0;1].

Таке визначення відображає припущення, що властивість є більш згуртованою зі своїм класом тоді, коли вона використовується більшою частиною методів цього класу. Якщо ж властивість майже не використовується методами класу, вага відповідного ребра буде низькою, що може свідчити про слабку структурну належність цієї властивості до класу.

Для методу μ, що належить класу c, вага ребра між вузлом методу та вузлом класу визначається за формулою:

$$w(\mu,c) = (\frac{m_{\mu,c}+p_{\mu,c}}{M_c+P_c}),$$

де $w(\mu,c)$ – вага ребра між вузлом методу μ та класу c; $m_{\mu,c}$ – к-сть методів класу c, що використовує метод μ; M_c – загальна к-сть методів класу c; $p_{\mu,c}$ – к-сть властивостей класу c, що використовує метод μ; P_c – загальна к-сть властивостей класу c.

На відміну від властивості, метод може бути пов’язаний зі своїм класом як через взаємодію з іншими методами, так і через використання властивостей класу. Тому у формулі враховуються обидва типи зв’язків. Чим більше метод взаємодіє з іншими елементами свого класу, тим вищою є вага відповідного ребра і тим сильнішою є його згуртованість із класом.

Отримані значення ваг ребер входять до загального набору характеристик графової моделі та надалі аналізуються статистично. Зокрема, ребра з аномально малою вагою можуть бути інтерпретовані як ознака низької згуртованості, оскільки

вони вказують на слабкий зв’язок елемента з класом, якому він структурно належить.

Для подальшого аналізу кожної з таких характеристик використовується єдиний статистичний критерій. Нехай x_i — значення певної характеристики для і-го елемента графа, тоді середнє значення визначається як:

$$\mu = \frac{1}{N} \Sigma x_i,$$

де N — кількість елементів, для яких обчислюється характеристика.

Стандартне відхилення σ обчислюється як:

$$\sigma = \sqrt{\frac{1}{N}\Sigma (x_i - \mu)^2}.$$

На основі цих величин вводиться критерій аномальності значення. Елемент графа вважається аномальним, якщо виконується умова:

$$|x - \mu| > k \cdot \sigma,$$

де x — значення характеристики для конкретного елемента ($x = x_i$), k — пороговий коефіцієнт, що визначає чутливість виявлення (у нашому випадку за замовчуванням $k = 2$).

Вибір значення $k = 2$ ґрунтується на емпіричному правилі двох стандартних відхилень, що широко застосовується у статистичному аналізі для визначення значущих відхилень від норми. При нормальному розподілі характеристик приблизно 95,4 % елементів знаходяться у межах ±2σ від середнього значення, тому елементи за межами цього інтервалу можна вважати статистично аномальними. У контексті аналізу характеристик графа, розподіл яких на практиці є правоскошеним, значення $k = 2$ забезпечує прийнятний баланс між чутливістю виявлення та кількістю хибнопозитивних спрацьовувань: при $k = 1,5$ кількість позначених аномальними елементів суттєво зростає, ускладнюючи інтерпретацію, тоді як при $k = 2,5$ частина реальних архітектурних проблем може залишатись непоміченою. Значення k є конфігурованим параметром і може бути скориговане залежно від задачі аналізу [69].

Такий підхід дозволяє визначати аномалії як суттєві відхилення від типових значень у межах конкретної системи. Зокрема, вузли з аномально великим степенем

визначаються як компоненти, для яких значення кількості зв’язків значно перевищує середнє, тоді як ребра з аномально малою вагою можуть свідчити про низьку згуртованість.

Важливою особливістю цього підходу є його адаптивність: критерій аномальності формується на основі самої системи, а не зовнішніх еталонів. Це дозволяє уникнути проблеми нерелевантності фіксованих порогових значень, що є характерною для класичних метрик програмного забезпечення.

Реалізація описаного підходу здійснюється у вигляді спеціалізованого компонента — детектора архітектурних аномалій, який виконує послідовну обробку графового представлення програмної системи. Робота детектора базується на поєднанні структурного аналізу графа та статистичної оцінки характеристик його елементів, що дозволяє виявляти відхилення як на локальному, так і на глобальному рівні.

Процес виявлення аномалій має поетапний характер і може бути представлений у вигляді узагальненого конвеєра обробки. На першому етапі здійснюється побудова графа залежностей на основі вихідного коду, як було описано раніше. Результатом цього етапу є орієнтований граф.

На наступному етапі для кожного елемента графа обчислюється набір характеристик, що використовуються для подальшого аналізу. До таких характеристик належать, зокрема, кількість вхідних та вихідних зв’язків (ступінь вершини), ваги ребер, а також похідні показники, що можуть відображати інтенсивність взаємодії між компонентами. Обчислення цих величин дозволяє сформувати числове представлення структури системи, яке є основою для застосування статистичних методів.

Після цього для кожної характеристики виконується статистичний аналіз, спрямований на визначення типових значень і виявлення відхилень. Як було показано вище, для цього обчислюються середнє значення та стандартне відхилення, на основі яких формується критерій аномальності. У результаті кожному елементу графа може бути поставлена у відповідність оцінка його відхилення від середнього значення для системи.

На основі отриманих оцінок детектор виконує ідентифікацію потенційних аномалій. При цьому враховуються як окремі характеристики елементів, так і їх поєднання. Наприклад, компонент може бути визнаний аномальним не лише через високий ступінь зв’язаності, але й у поєднанні з іншими ознаками, такими як низька згуртованість або нетипова структура зв’язків.

Окрім аналізу окремих елементів, детектор також враховує глобальні властивості графа. Зокрема, виконується пошук циклічних залежностей, що відповідають сильно зв’язаним компонентам, а також виявлення щільно пов’язаних підграфів. Для цього можуть використовуватися алгоритми пошуку компонент зв’язності, аналізу досяжності та інші методи теорії графів. Подібні структури розглядаються як потенційні індикатори архітектурних аномалій.

Важливою особливістю роботи детектора є відсутність жорстко заданих правил, що визначають аномальність [70]. Натомість рішення приймається на основі відносних відхилень характеристик у межах конкретної системи. Це дозволяє адаптувати підхід до різних програмних проєктів, незалежно від їх масштабу або архітектурного стилю.

Результатом роботи детектора є множина елементів графа, що ідентифікуються як потенційні архітектурні аномалії. Кожен із таких елементів може бути співвіднесений із відповідним фрагментом вихідного коду, що дозволяє одразу локалізувати проблему у вихідному коді.

Наш детектор архітектурних аномалій виступає як інструмент, що інтегрує результати аналізу метрик і структури графа, забезпечуючи автоматизоване виявлення відхилень у програмній архітектурі. Його використання дозволяє перейти від описового аналізу до формалізованого процесу оцінювання якості архітектури.

У межах запропонованого підходу архітектурні аномалії інтерпретуються не як абстрактні порушення принципів проєктування, а як конкретні типи відхилень, що можуть бути формалізовані на основі характеристик графа залежностей. Така формалізація дозволяє не лише виявляти аномалії, але й класифікувати їх за типами, що спрощує подальший аналіз і інтерпретацію результатів.

Одним із базових типів аномалій є вузли з аномально високим рівнем зв’язаності. Формально такі елементи визначаються як вершини графа, для яких значення степеня (кількість вхідних і вихідних зв’язків) суттєво перевищує середнє значення для системи відповідно до статистичного критерію, наведеного вище. Подібні компоненти виступають як точки концентрації залежностей і можуть відповідати централізованим елементам архітектури. У випадку, якщо така централізація не обумовлена архітектурними вимогами, це може свідчити про наявність антипатернів, пов’язаних із надмірною зв’язаністю.

Іншим типом аномалій є елементи з аномально низькою згуртованістю. У графовому представленні це проявляється як наявність слабко пов’язаних або ізольованих підмножин зв’язків, що відповідають одному компоненту. Формально така аномалія може бути пов’язана з характеристиками, що відображають інтенсивність взаємодії між елементами, зокрема вагою ребер або щільністю локального підграфа. Значення цих характеристик, що суттєво відрізняються від типових, можуть вказувати на порушення внутрішньої цілісності компонента.

Окрему категорію становлять циклічні залежності, які у графовій моделі відповідають наявності сильно зв’язаних компонент. На відміну від попередніх типів аномалій, вони визначаються не через окремі значення характеристик, а через структуру графа. Виявлення таких аномалій здійснюється шляхом пошуку підграфів, у яких кожна вершина досяжна з будь-якої іншої. Подібні структури є індикаторами взаємозалежності компонентів і часто свідчать про порушення принципів модульності.

Крім того, детектор дозволяє виявляти аномалії, пов’язані з нетиповими характеристиками зв’язків між компонентами. Зокрема, ребра з аномально низькою або високою вагою можуть свідчити про слабку або надмірну інтенсивність взаємодії між відповідними елементами. У першому випадку це може вказувати на випадкові або несуттєві залежності, тоді як у другому — на надмірну зв’язаність між компонентами.

Ще одним типом аномалій є формування щільно пов’язаних груп компонентів, які можуть не відповідати логічній структурі системи. У графовому представленні

такі групи проявляються як підграфи з високою щільністю зв’язків. Їх виявлення може свідчити як про природне групування компонентів, так і про порушення принципів декомпозиції, якщо ці зв’язки виходять за межі очікуваних модулів.

Важливо відзначити, що зазначені типи аномалій не є взаємовиключними. Один і той самий елемент може одночасно проявляти декілька ознак відхилення, що ускладнює їх однозначну класифікацію. Звідси детектор розглядає аномалії як множину характеристик, що відхиляються від типових значень, а не як жорстко визначені категорії.

Важливим етапом застосування запропонованого підходу є локалізація виявлених архітектурних аномалій у структурі програмної системи. На відміну від абстрактного аналізу графових характеристик, практичне використання результатів потребує їх співвіднесення з конкретними елементами вихідного коду, перетворюючи формальний звіт про відхилення на практично придатний для рефакторингу.

У процесі роботи детектора кожному елементу графа, для якого було зафіксовано аномальне відхилення, ставиться у відповідність набір характеристик, що стали підставою для його класифікації як аномального. Це можуть бути як окремі показники (наприклад, значення ступеня вершини), так і їх комбінації, що дозволяють більш точно описати природу виявленої проблеми. У результаті формується множина аномальних елементів із поясненням причин їх відхилення від типових значень.

Важливим аспектом є те, що одна і та ж аномалія може бути представлена на різних рівнях абстракції. Наприклад, вузол із високим рівнем зв’язаності може відповідати окремому класу, який виконує надмірну кількість функцій, або ж групі методів, що активно взаємодіють із великою кількістю інших компонентів. Тому результати аналізу повинні інтерпретуватися з урахуванням рівня деталізації моделі, що використовується.

Практичне застосування результатів аналізу передбачає використання отриманої інформації для вдосконалення архітектури програмної системи. Зокрема, виявлення компонентів із високим рівнем зв’язаності може слугувати підставою

для їх декомпозиції, тоді як ідентифікація слабко згуртованих елементів — для перегляду їх функціонального призначення. Аналогічно, виявлення циклічних залежностей може вказувати на необхідність реорганізації структури модулів з метою усунення взаємних залежностей.

Важливо підкреслити, що результати роботи детектора не повинні розглядатися як однозначні вказівки до рефакторингу, а радше як індикатори потенційних проблем, що потребують подальшого аналізу. У багатьох випадках виявлені аномалії можуть бути обумовлені специфікою предметної області або свідомими архітектурними рішеннями. Саме тому інтерпретація результатів повинна здійснюватися з урахуванням контексту конкретної системи.

Окрему увагу слід приділити можливості інтеграції запропонованого підходу у процес розробки програмного забезпечення. Завдяки автоматизованому характеру детектора його можна використовувати як інструмент регулярного аналізу архітектури, що дозволяє виявляти проблеми на ранніх етапах їх виникнення. Це особливо важливо в умовах довготривалого розвитку системи, коли накопичення архітектурних аномалій може призводити до поступової деградації її структури.

Результати аналізу можуть також використовуватись для візуалізації архітектури програмної системи, що полегшує їх сприйняття та інтерпретацію. Відображення аномальних елементів у графовому представленні дозволяє швидко ідентифікувати проблемні області та оцінити їх вплив на загальну структуру системи. Такий підхід є особливо ефективним у поєднанні з інтерактивними засобами візуалізації, що дозволяють досліджувати структуру системи на різних рівнях деталізації.

Локалізація та інтерпретація архітектурних аномалій є невід'ємною частиною запропонованого підходу, що забезпечує його практичну цінність і дозволяє не лише виявляти структурні відхилення, але й пов'язувати їх із конкретними елементами програмної системи., що створює основу для подальшого вдосконалення її архітектури.

Для ілюстрації запропонованого підходу розглянемо результати його застосування до реальної програмної системи, отримані у ході проведеного дослідження [71].

У якості тестового прикладу було використано застосунок-гра 2048, реалізований мовою Swift. Незважаючи на відносно невеликий розмір проєкту, його структура містить характерні приклади порушень архітектурних принципів, що дозволяє продемонструвати ефективність запропонованого підходу навіть у простих системах. У результаті автоматизованого аналізу було виявлено декілька типів аномалій, зокрема вузли з аномально великим степенем, елементи з низькою згуртованістю, а також області підвищеної щільності зв’язків.

Одним із найбільш показових прикладів є клас ViewController, який було ідентифіковано як вузол з аномально великим степенем. Відповідний фрагмент коду наведено нижче:

```
class ViewController: UIViewController {
        var manager: GameLogicManager!
        var score: Score!
        var highScore: HighScore!
        var renderer: GameBoardRenderer!
```

Лістинг 2. Властивості класу ViewController

У графовій моделі архітектури цей клас відповідає вершині, що має значну кількість як вхідних, так і вихідних зв’язків. Кількісний аналіз показує, що значення степеня цієї вершини суттєво перевищує середнє значення для множини аналогічних елементів, що задовольняє умову критерію аномальності. Це дозволяє класифікувати відповідний компонент як аномальний.

З точки зору архітектури така структура свідчить про концентрацію значної частини функціональності в одному класі. Компонент фактично виступає як централізований вузол взаємодії, що поєднує різні аспекти логіки застосунку — управління станом гри, обробку результатів та взаємодію з інтерфейсом. Подібна концентрація відповідальностей ускладнює модифікацію коду, знижує його модульність та підвищує ризик виникнення побічних ефектів при внесенні змін.

Схожий характер аномалії було виявлено і на рівні окремих методів. Зокрема, метод save(dimension: Int, tiles: [Tile]) також було класифіковано як вузол з аномально високим степенем:

```
func save(dimension: Int, tiles: [Tile]) {
        deleteAllTiles()
        for tile in tiles {
                let tileModel = TileModel(context:
ResistanceService.context)
                tileModel.positionX = Int16(tile.position.x)
                tileModel.positionY = Int16(tile.position.y)
                tileModel.tileValue = Int32(tile.value ?? 0)
                ResistanceService.saveContext()
        }
```

Лістинг 3. Порушення інкапсуляції в методі save(dimension: Int, tiles: [Tile])

У даному випадку велика кількість зв’язків формується внаслідок активної взаємодії методу з різними елементами системи, зокрема створенням нових об’єктів, доступом до їх властивостей та викликом допоміжних сервісів. З позиції графової моделі це призводить до збільшення кількості ребер, інцидентних відповідній вершині, що і зумовлює її класифікацію як аномальної.

Крім високої зв’язаності, у цьому прикладі також простежуються ознаки порушення інкапсуляції. Метод безпосередньо оперує внутрішніми деталями реалізації інших компонентів, що знижує рівень абстракції та ускладнює підтримку коду. Таким чином, виявлена аномалія має не лише формальний, але й змістовний характер, що підтверджує коректність використаного критерію.

Іншим важливим типом аномалій, виявлених у ході аналізу, є порушення згуртованості компонентів. У досліджуваному проекті було ідентифіковано ряд властивостей класу Tile, які демонструють аномально низький рівень зв’язку зі своїм класом. При цьому більш тісна взаємодія спостерігається з методами іншого компонента, зокрема методу isGameOver() класу GameLogicManager:

```
private func isGameOver() -> Bool {
        if tiles.filter({$0.value == nil}).count != 0 {
                return false
        }
        for tile in tiles {
                let v = tile.value!
                let neighbours = [tile.upTile, tile.rightTile, tile.bottomTile,
tile.leftTile]
                                        .filter { $0?.value == v }
```

Лістинг 4. Порушення інкапсуляції в методі isGameOver() -> Bool

У графовій моделі така ситуація проявляється як аномально мала вага ребер, що поєднують відповідні властивості з їхнім класом. Відповідно до використаного статистичного підходу такі значення ідентифікуються як відхилення від типових і класифікуються як аномальні. З архітектурної точки зору це свідчить про те, що логіка роботи з даними винесена за межі їх природного контексту, що є порушенням принципу інкапсуляції.

Окрім аналізу окремих вузлів, у ході дослідження було зафіксовано наявність структурних аномалій, пов’язаних із просторовими характеристиками графа. Зокрема, було виявлено області з підвищеною щільністю зв’язків, що проявляються у вигляді значного зближення або перетину ребер у візуалізації. Хоча такі ознаки не завжди однозначно вказують на конкретну помилку, вони корелюють із загальною складністю структури та можуть слугувати додатковим індикатором проблемних ділянок.

У цілому результати аналізу показали, що виявлені аномалії узгоджуються з реально існуючими проблемами в архітектурі програмного коду, що було підтверджено подальшим ручним аналізом. Це свідчить про те, що запропонований підхід дозволяє не лише формально виявляти відхилення, але й ефективно локалізовувати потенційно проблемні області системи.

Проведені експерименти підтвердили гіпотезу що, використання графової моделі у поєднанні зі статистичним аналізом характеристик дозволяє отримати інформативне та інтерпретоване представлення архітектурних аномалій, що має безпосередню практичну цінність у процесі аналізу та вдосконалення програмного забезпечення.

Перш за все, слід зазначити, що якість результатів безпосередньо залежить від повноти та точності побудованої графової моделі. Оскільки граф формується на основі аналізу вихідного коду, будь-які обмеження або неточності на цьому етапі можуть впливати на подальші результати. Зокрема, складні механізми мови програмування, такі як динамічне зв’язування, рефлексія або використання зовнішніх бібліотек, можуть бути не повністю враховані при побудові графа, що призводить до неповного відображення залежностей між компонентами.

Другим важливим обмеженням є залежність результатів від обраного рівня абстракції моделі. Представлення системи на рівні класів, методів або інших елементів може суттєво впливати на характер виявлених аномалій. Наприклад, аномалія, що проявляється на рівні методів, може бути непомітною на рівні класів, і навпаки. Результати аналізу повинні інтерпретуватися з урахуванням обраного рівня деталізації.

Ще одним обмеженням є використання статистичного підходу до визначення аномалій. Хоча критерій відхилення від середнього значення дозволяє адаптуватися до конкретної системи, він не гарантує однозначної інтерпретації результатів. Зокрема, елементи, що формально відповідають критерію аномальності, можуть бути обґрунтованими з точки зору архітектури, наприклад, якщо вони виконують роль центральних компонентів системи. У той же час деякі проблемні елементи можуть не бути виявленими, якщо їх характеристики не відрізняються достатньо від середніх значень.

Використання лише структурної інформації про систему обмежує можливості виявлення поведінкових аномалій. Граф залежностей відображає статичну структуру взаємодій між компонентами, однак не враховує особливості їх виконання, такі як частота викликів, динаміка використання або контекст

виконання. Частина типів проблем, пов’язані з продуктивністю або некоректною поведінкою системи, залишаються поза межами розглянутого підходу.

Окремо слід відзначити обмеження, пов'язані з інтерпретацією результатів. Враховуючи евристичний характер методології, кінцеве рішення щодо наявності та критичності аномалії потребує участі розробника або архітектора. Детектор надає інформацію про потенційні відхилення, однак не може автоматично визначити, чи є вони проблемними у конкретному контексті. Це обумовлює необхідність поєднання автоматизованого аналізу з експертною оцінкою.

Ефективність підходу також може залежати від розміру та складності програмної системи. У невеликих системах статистичні характеристики можуть бути нестабільними через обмежену кількість елементів, що ускладнює визначення типових значень. У великих системах, навпаки, може виникати проблема надмірної кількості виявлених аномалій, що потребує додаткових механізмів фільтрації та пріоритезації.

Нарешті, слід підкреслити, що запропонований підхід не є універсальним засобом оцінювання якості архітектури, а розглядається як один із інструментів аналізу. Його ефективність підвищується у поєднанні з іншими методами, такими як аналіз метрик, рев’ю коду або використання архітектурних моделей вищого рівня.

3.4 Висновки

Архітектурна аномалія — відхилення від бажаних властивостей системи (низька зв'язаність, висока згуртованість, чітке розділення відповідальностей) — може проявлятися на рівні окремих компонентів або структури системи загалом. Показано, що такі аномалії можуть проявлятися як на рівні окремих компонентів, так і на рівні структури системи в цілому.

Було розглянуто роль метрик об’єктно-орієнтованого дизайну у формалізації архітектурних властивостей. Зокрема, метрики WMC, RFC, NOC, CBO та LCOM дозволяють кількісно оцінювати складність, зв’язаність, ієрархічну структуру та згуртованість компонентів. Водночас встановлено, що використання лише метрик

має обмеження, оскільки їх значення потребують контекстної інтерпретації та не завжди враховують глобальну структуру системи.

Для подолання цих обмежень було запропоновано використовувати графову модель архітектури. У межах такого підходу аномалії розглядаються як структурні відхилення у графі залежностей. Було формалізовано ознаки таких відхилень, зокрема вузли з аномально високим степенем, ребра з аномально малою вагою, циклічні залежності та просторові аномалії у візуальному представленні графа.

Окрему увагу було приділено автоматизації виявлення аномалій із використанням детектора архітектурних аномалій. Запропонований підхід базується на обчисленні характеристик графа, визначенні типових значень у межах конкретної системи та виявленні елементів, що суттєво відхиляються від них. Це дозволяє уникнути жорстко заданих універсальних порогів і зробити аналіз адаптивним до особливостей конкретної програмної системи.

На прикладі реального Swift-застосунку було показано, що виявлені у графовій моделі аномалії можуть бути співвіднесені з конкретними фрагментами вихідного коду та підтверджені шляхом детального аналізу. Це демонструє практичну цінність запропонованого підходу для локалізації потенційно проблемних областей архітектури.

Підтверджено гіпотезу: поєднання метрик, графової моделі та статистичного аналізу формує основу для автоматизованого виявлення архітектурних аномалій. Детектор не замінює експертну оцінку, але суттєво звужує простір пошуку проблемних ділянок.

РОЗДІЛ 4

ВІЗУАЛІЗАЦІЯ ГРАФОВОЇ МОДЕЛІ АРХІТЕКТУРИ ПРОГРАМНОГО МОДУЛЯ

4.1 Проблема візуалізації графів архітектури програмного забезпечення

Візуалізація графових моделей архітектури програмного забезпечення є складною міждисциплінарною задачею, що знаходиться на перетині аналізу програмного коду, теорії графів та візуальної аналітики.

На відміну від класичних задач візуалізації графів, що розглядаються у загальному вигляді, графи програмної архітектури мають низку специфічних особливостей. Це означає, що граф не є однорідною структурою, і його візуалізація повинна враховувати семантичні відмінності між елементами.

Орієнтованість відображає напрямок залежностей між компонентами, що є критично важливим для розуміння структури системи, тоді як ваги ребер дозволяють враховувати інтенсивність взаємодії між елементами. Візуалізація повинна одночасно передавати кілька рівнів інформації: топологію графа, напрямок зв’язків, їх вагу та семантичний тип. Це суттєво ускладнює задачу побудови наочного представлення.

Додатковою складністю є масштаб графа. Навіть для відносно невеликих програмних систем кількість елементів може досягати сотень або тисяч, що призводить до швидкого зростання кількості зв’язків. У випадку повного представлення залежностей між компонентами кількість ребер може зростати квадратично відносно кількості вузлів, що створює суттєве навантаження як на алгоритми візуалізації, так і на сприйняття результату користувачем.

У цьому контексті виникає фундаментальна суперечність між двома основними вимогами до візуалізації: обчислювальною ефективністю та якістю розміщення. З одного боку, алгоритми повинні забезпечувати побудову візуалізації за прийнятний час, що є критично важливим у інтерактивних сценаріях використання. З іншого боку, отримане розміщення має бути достатньо інформативним і читабельним, щоб

користувач міг інтерпретувати структуру системи та виявляти потенційні проблеми.

Якість візуалізації визначається рядом критеріїв, які відображають як формальні, так і когнітивні аспекти сприйняття графа. До таких критеріїв належать мінімізація перетинів ребер, рівномірність розподілу вузлів у просторі, збереження локальної структури зв’язків, а також відсутність перекриття елементів. Водночас ці критерії не є незалежними: оптимізація одного з них може призводити до погіршення іншого. Наприклад, зменшення кількості перетинів часто супроводжується збільшенням довжини ребер, що ускладнює сприйняття структури, тоді як ущільнення вузлів може призводити до перекриття та втрати читабельності.

Найбільш поширеним класом алгоритмів для візуалізації графів є силові (force-directed) методи. Вони базуються на моделюванні графа як фізичної системи, у якій вершини відштовхуються одна від одної, а ребра діють як пружини, що прагнуть утримувати пов’язані елементи разом. Такий підхід дозволяє отримати розміщення, яке інтуїтивно відображає структуру зв’язків між компонентами, і широко використовується завдяки своїй універсальності.

Однак силові алгоритми мають низку суттєвих обмежень. По-перше, вони характеризуються високою обчислювальною складністю, оскільки на кожній ітерації необхідно обчислювати взаємодію між великою кількістю пар вузлів. Це обмежує їх застосування для великих графів або в інтерактивних системах. По-друге, результати таких алгоритмів значною мірою залежать від початкового розміщення вузлів і параметрів моделі, що може призводити до різних варіантів візуалізації для одного і того ж графа. По-третє, вони можуть застрягати у локальних мінімумах, що знижує якість отриманого розміщення.

Додатковою проблемою є відсутність стабільності та повторюваності результатів. У задачах аналізу архітектури важливо мати можливість порівнювати різні версії системи або результати різних запусків алгоритму. Якщо візуалізація змінюється навіть без суттєвих змін у структурі графа, це ускладнює інтерпретацію результатів і знижує довіру до інструменту.

Окремо слід розглянути проблему масштабування візуалізації. Зі зростанням розміру графа складність його обробки зростає не лише з точки зору обчислень, але й з точки зору сприйняття. Навіть якщо алгоритм здатен побудувати формально коректне розміщення, користувач може не бути здатним ефективно інтерпретувати результат через надмірну кількість елементів, високу щільність зв’язків та відсутність чіткої візуальної структури. Це явище можна розглядати як когнітивне перевантаження, що обмежує практичну корисність візуалізації.

Проблема візуалізації графів архітектури є комплексною — вона охоплює алгоритмічні та когнітивні аспекти одночасно.

Наявні підходи не забезпечують повного вирішення задачі візуалізації графових моделей архітектури, що зумовлює необхідність пошуку нових методів побудови та відображення layout. Основною проблемою залишається одночасне забезпечення високої якості розміщення, обчислювальної ефективності та стабільності результатів, що є складною багатокритеріальною задачею.

Одним із перспективних напрямків є використання методів машинного навчання, які дозволяють перейти від жорстко заданих евристик до адаптивних моделей, здатних враховувати особливості конкретного графа. Зокрема, моделі на основі графових нейронних мереж можуть використовуватися для прогнозування початкового розміщення вузлів, що наближене до оптимального. Це дозволяє суттєво зменшити кількість ітерацій, необхідних для збіжності класичних алгоритмів, а також підвищити стабільність отриманих результатів. Таким чином, машинне навчання виступає не як повна заміна існуючих методів, а як їх доповнення, що дозволяє підвищити ефективність процесу візуалізації.

Водночас не менш важливим є вдосконалення засобів відображення графів, орієнтованих на сприйняття користувачем. У цьому контексті перспективним є використання тривимірних середовищ, зокрема технологій доповненої реальності, які дозволяють природно розширити простір візуалізації та зменшити щільність розміщення елементів. Використання додаткового виміру дає змогу знизити кількість перекриттів, покращити розділення компонентів та забезпечити більш

гнучку взаємодію з графом, зокрема шляхом його обертання, масштабування та вибіркового дослідження окремих підструктур.

Важливу роль відіграють підходи, спрямовані на зменшення когнітивного навантаження користувача, що виникає при роботі з великими та складними графами. Це включає оптимізацію візуальних характеристик, таких як контраст, щільність розміщення, використання кольору та відображення міток, а також адаптацію візуалізації до задачі аналізу. У сукупності ці заходи дозволяють підвищити інтерпретованість графа та ефективність його використання як інструменту аналізу архітектури.

Подальший розвиток підходів до візуалізації потребує одночасного вдосконалення алгоритмів і врахування особливостей людського сприйняття. У наступних підрозділах розглядаються відповідні напрямки, зокрема застосування методів машинного навчання для побудови layout, використання доповненої реальності та підходи до зменшення когнітивного навантаження при аналізі складних структур.

4.2 Метрики якості візуалізації та їх обмеження

Задача візуалізації графа традиційно розглядається як задача відображення дискретної структури у неперервний простір із збереженням її топологічних та геометричних властивостей. Нехай граф задано як

$$\mathrm{G} = (\mathrm{V}, \mathrm{E}),$$

де V — множина вершин, а E — множина ребер.

Візуалізація такого графа визначається функцією розміщення:

$$\mathrm{L} : \mathrm{V} \to \mathbb{R}^d,$$

де d = 2 або d = 3. Функція L задає координати кожної вершини у просторі візуалізації та визначає просторову конфігурацію графа.

У більшості підходів задача побудови layout формалізується як задача оптимізації певної функції якості:

$$\min_L F(L),$$

де F(L) оцінює “якість” розміщення. Як правило, ця функція не є одновимірною і представляється як набір критеріїв:

$$F(L) = (f_1(L), f_2(L), \dots, f_k(L)),$$

де кожен компонент відповідає окремій метриці, наприклад кількості перетинів ребер, довжині зв’язків або їх кутовому розділенню.

Задача візуалізації — це багатокритеріальна оптимізація: вимоги до мінімізації перетинів, рівномірного розподілу і збереження симетрії часто суперечать одна одній. У загальному випадку не існує єдиного оптимального розміщення, і результат залежить від обраного набору критеріїв та їх відносної важливості.

У контексті візуалізації графів архітектури програмного забезпечення така формалізація має суттєві обмеження. По-перше, більшість критеріїв якості є геометричними і не враховують семантичний зміст графа, зокрема роль окремих компонентів системи або типи залежностей між ними. По-друге, вибір функції F(L) та вагових коефіцієнтів між її складовими є суб’єктивним і залежить від конкретної задачі аналізу.

При цьому навіть формально "оптимальне" розміщення не гарантує його ефективності з точки зору користувача. Візуалізація може задовольняти задані математичні критерії, але залишатися складною для сприйняття через велику кількість елементів або відсутність чіткої структурної організації.

Відповідно, хоча уточнення задачі візуалізації у вигляді оптимізаційної проблеми є зручною з теоретичної точки зору, вона не забезпечує повного вирішення задачі побудови інформативного та зручного для аналізу представлення графа. Це обумовлює необхідність детального розгляду окремих метрик якості та їх обмежень.

У задачах візуалізації графів для оцінювання якості розміщення традиційно використовується набір метрик, що відображають геометричні та структурні властивості отриманого layout. Ці метрики дозволяють формалізувати інтуїтивні уявлення про “якісну” візуалізацію та застосовуються як критерії оптимізації у відповідних алгоритмах.

Однією з базових метрик є кількість перетинів ребер (crossing number). Вона визначається як загальна кількість точок, у яких два ребра графа перетинаються у площині:

$$CN = |\{(e_i, e_j) \in E \times E \mid e_i \cap e_j \neq \emptyset\}|$$

Мінімізація цієї метрики спрямована на зменшення візуального шуму, оскільки перетини ускладнюють відстеження зв'язків між вершинами. Проте задача мінімізації кількості перетинів є NP-складною, що змушує використовувати наближені методи. Крім того, у великих графах повне усунення перетинів є практично неможливим, що обмежує застосовність цієї метрики як універсального критерію.

Іншою важливою метрикою є довжина ребер (edge length). Для заданого розміщення вона визначається як:

$$EL = \sum_{(u,v)\in E} \|L(u) - L(v)\|$$

де $L(u)$ та $L(v)$— координати відповідних вершина $\|\cdot\|$ — евклідова норма, що визначає відстань між ними у просторі візуалізації. Часто розглядається не лише сумарна довжина, але й її варіація:

$$\mathrm{Var(EL)} = \frac{1}{|\mathrm{E}|}\sum_{(\mathrm{u,v})\in \mathrm{E}}\left(\|L(u) - L(v)\| - \overline{EL}\right)^2,$$

Мінімізація варіації довжин ребер дозволяє отримати більш "рівномірний" граф, що покращує його візуальну узгодженість. Водночас надмірна оптимізація цієї метрики може призводити до штучного викривлення структури, зокрема до зближення вузлів, які не мають сильних зв'язків.

Ще однією метрикою є кутове розділення (angular resolution), яке визначається як мінімальний кут між будь-якими двома ребрами, що виходять з однієї вершини:

$$\mathrm{AR} = \min_{\mathrm{v}\in \mathrm{V}} \min_{(\mathrm{e_i},\mathrm{e_j})\in \delta(\mathrm{v})} \angle(e_i, e_j)$$

де $\delta(v)$ — множина ребер, інцидентних вершині v. Збільшення цього показника покращує локальну читабельність, оскільки дозволяє чітко розрізняти зв'язки, що виходять з одного вузла. Проте оптимізація кутового розділення часто призводить до збільшення простору, необхідного для розміщення графа.

Окрему групу становлять метрики, що оцінюють відповідність геометричної структури графа його топології. До них належить, зокрема, метрика stress, яка визначається як:

$$Stress = \sum_{i<j}\left(\left\|L(v_i) - L(v_j)\right\| - d_{ij}\right)^2,$$

де d_{ij} — найкоротша відстань між вершинами v_i та v_j у графі. Ця метрика прагне зберегти відносні відстані між вершинами, забезпечуючи відповідність між топологічною та геометричною структурами. Водночас її оптимізація може призводити до глобальних деформацій графа, що погіршують локальну читабельність.

Попри формальну визначеність, наведені метрики мають спільну властивість — вони оцінюють окремі аспекти структури графа ізольовано. Жодна з них не дає повного уявлення про якість візуалізації, а їх комбінування призводить до задачі багатокритеріальної оптимізації, у якій необхідно балансувати між взаємно суперечливими вимогами.

У контексті графів архітектури програмного забезпечення ці обмеження стають ще більш вираженими через наявність семантичного навантаження. На відміну від абстрактних графів, елементи програмної системи виконують різні ролі, а зв’язки між ними можуть мати різну природу: виклик методу, доступ до даних, наслідування або використання типу. Класичні геометричні метрики не враховують цих відмінностей і оцінюють layout переважно як просторову структуру. Через це один і той самий фрагмент графа може мати різну інтерпретацію з точки зору архітектури. Наприклад, вузол із великою кількістю зв’язків може бути як ознакою архітектурної аномалії, так і відображенням центрального компонента системи. У межах геометричних метрик ці випадки не розрізняються.

Саме тому конфлікт між метриками у графах архітектури має не лише геометричний, але й семантичний характер. Покращення формального розміщення може зменшити кількість перетинів або вирівняти довжини ребер, але водночас спотворити уявлення про роль окремих компонентів у системі. Отже, якість такої візуалізації повинна оцінюватися не лише за геометричними показниками, а й з

урахуванням того, наскільки коректно вона підтримує аналіз архітектурної структури.

Зокрема, компоненти з великою кількістю залежностей можуть виступати як центральні елементи системи, що ускладнює баланс між структурною коректністю та геометричною читабельністю.

У результаті побудова візуалізації неминуче пов’язана з вибором компромісу між різними критеріями якості. Такий компроміс не є універсальним і визначається конкретною задачею аналізу, а також очікуваннями користувача. Для одних сценаріїв більш важливою є геометрична акуратність, тоді як для інших — збереження структурних або семантичних зв’язків між компонентами.

Для одного і того ж графа може існувати декілька альтернативних варіантів візуалізації, кожен з яких є коректним з формальної точки зору, але відрізняється за рівнем зручності для інтерпретації. Відсутність єдиного критерію якості ускладнює як побудову оптимального layout, так і його об’єктивне оцінювання.

Візуалізація графа не зводиться до оптимізації фіксованого набору метрик. Вона має комплексний характер і потребує врахування не лише геометричних властивостей, але й структурних особливостей системи, а також контексту її використання. Це обумовлює необхідність розширення підходів до оцінювання якості візуалізації.

Незважаючи на формалізований характер метрик якості візуалізації, їх застосування не враховує особливостей сприйняття інформації користувачем. У контексті аналізу архітектури програмного забезпечення це є критичним обмеженням, оскільки кінцева мета візуалізації полягає не лише у відображенні структури графа, а у забезпеченні її ефективного розуміння.

Однією з основних проблем є перевантаження візуального простору (visual clutter) [72], [73], [74], що виникає при великій кількості вузлів і зв’язків. Навіть за відсутності значної кількості перетинів граф може залишатися складним для сприйняття через високу щільність елементів. У таких умовах користувач змушений витрачати додаткові зусилля для відокремлення релевантної інформації від фону, що знижує ефективність аналізу.

Іншим важливим фактором є перекриття елементів (occlusion), коли вузли або ребра частково або повністю перекривають одне одного. Це особливо характерно для двовимірних візуалізацій, де обмежений простір не дозволяє розмістити всі елементи без втрат видимості. Перекриття ускладнює відстеження зв’язків та може призводити до втрати інформації про структуру графа.

Додатковою проблемою є накладання підписів (label overlap), що виникає при відображенні текстової інформації, пов’язаної з вузлами. У випадку графів архітектури програмного забезпечення це має особливе значення, оскільки ідентифікація елементів є необхідною умовою їх інтерпретації. Накладання підписів або їх надмірне скорочення знижує інформативність візуалізації та ускладнює її використання.

Важливу роль відіграє також контрастність та візуальне розділення елементів. Недостатній контраст між вузлами, ребрами та фоном може призводити до втрати чіткості структури, тоді як надмірна візуальна різноманітність, зокрема використання великої кількості кольорів, може ускладнювати сприйняття через перевантаження уваги користувача.

Крім того, на ефективність сприйняття впливає наявність або відсутність візуальної ієрархії. У складних графах важливо мати можливість швидко ідентифікувати ключові елементи, групи компонентів або структурні патерни. У випадку, якщо всі елементи представлені однаково, користувач змушений самостійно шукати ці структури, що значно ускладнює аналіз.

Ефективність візуалізації визначається не геометричними характеристиками сама по собі, а відповідністю особливостям людського сприйняття. Метрики, що використовуються у класичних підходах, не враховують ці аспекти, що призводить до розриву між формальною якістю layout та його практичною корисністю. Це узгоджується з результатами сучасних досліджень у галузі візуалізації графів, які показують, що оптимізація за формальними метриками не завжди призводить до покращення сприйняття структури користувачем.

У зв’язку з цим виникає необхідність використання підходів, орієнтованих на зменшення когнітивного навантаження та підвищення інтерпретованості графових

представлень. До таких підходів належать як удосконалення алгоритмів розміщення, так і використання нових засобів візуалізації, зокрема тривимірних середовищ і доповненої реальності, що дозволяють більш ефективно організувати простір відображення.

4.3 Використання методів машинного навчання для прискорення візуалізації графа

Незважаючи на широке застосування класичних алгоритмів візуалізації графів, зокрема методів, заснованих на фізичних моделях, їх практичне використання у задачах аналізу архітектури програмного забезпечення пов'язане з рядом обмежень [75], [76], [77].

Такі алгоритми демонструють високу якість візуалізації з точки зору сприйняття, оскільки дозволяють формувати природні та інтуїтивно зрозумілі структури, у яких взаємопов'язані елементи розташовуються ближче один до одного. Це узгоджується з результатами досліджень у галузі візуалізації графів, які показують, що силові підходи забезпечують кращу інтерпретованість структури порівняно з багатьма альтернативними методами.

Водночас такі алгоритми мають суттєві недоліки. По-перше, їх обчислювальна складність у базовому варіанті становить $O(n^2)$, що обмежує можливість застосування до великих графів. По-друге, результат роботи алгоритму значною мірою залежить від початкового розміщення вершин, що призводить до нестабільності отриманих візуалізацій. По-третє, оптимізація функції енергії часто зупиняється у локальних мінімумах, що може погіршувати якість фінального layout.

Існуючі підходи до прискорення обчислень, зокрема використання алгоритму Barnes–Hut [78], дозволяють зменшити обчислювальну складність до $O(n \log n)$, однак не усувають залежність результату від початкового стану системи. У таких умовах навіть незначні відмінності у початковому розміщенні можуть призводити до суттєво різних конфігурацій графа [79].

З цього виникає необхідність пошуку підходів, які дозволяють зберегти переваги силових алгоритмів з точки зору якості візуалізації, одночасно зменшуючи їх обчислювальну складність та підвищуючи стабільність отриманих результатів.

У рамках нашого підходу силові алгоритми розглядаються як базовий підхід до побудови візуалізації графа. Такий вибір обумовлений тим, що, незважаючи на наявні обмеження, вони забезпечують високий рівень відповідності між структурою графа та його візуальним представленням, що є критично важливим у задачах аналізу архітектури програмного забезпечення.

Силові алгоритми дозволяють зберігати природні кластери та відображати структуру залежностей між компонентами системи. Це особливо важливо для графів архітектури, де просторове розташування вузлів має відображати логіку взаємодії між ними.

Альтернативні підходи, засновані на методах машинного навчання [80], [81], [82], можуть використовуватися для побудови векторних представлень графа або апроксимації його структури. Водночас такі підходи зазвичай орієнтовані на швидкодію, або ж на вирішення окремих задач, зокрема отримання embedding-представлень або прогнозування окремих характеристик графа, і не завжди забезпечують формування повноцінного layout, придатного для візуального аналізу.

Зокрема, методи на кшталт node2vec [83], [84] орієнтовані на отримання embedding-представлень, тоді як підходи, що безпосередньо генерують координати, можуть мати обмеження щодо масштабованості або не враховувати ваги зв’язків [85], [86], [87], [88], [89].

У зв’язку з цим доцільним є використання комбінованого підходу, у якому силовий алгоритм відповідає за формування фінальної конфігурації графа, а методи машинного навчання використовуються для покращення початкових умов його роботи.

З метою подолання обмежень класичних силових алгоритмів у даній роботі запропоновано підхід, що поєднує їх із методами машинного навчання. Основна ідея полягає не у заміні силового алгоритму, а у використанні моделі машинного

навчання як евристики, що дозволяє покращити початкове розміщення вершин графа [90], [91].

У класичних реалізаціях силових алгоритмів початкове розміщення вершин зазвичай визначається випадковим чином. Такий підхід призводить до значної варіативності результатів та впливає на швидкість збіжності алгоритму, оскільки початковий стан може знаходитися далеко від оптимальної конфігурації.

У запропонованому підході замість випадкового розміщення використовується функція, що апроксимується моделлю машинного навчання:

$$L_0 = ML(G),$$

де G — граф, а L_0 — початкове розміщення вершин, сформоване моделлю. Отримане розміщення використовується як початковий стан для подальшої оптимізації за допомогою силового алгоритму.

Фінальне розміщення графа визначається як результат роботи силового алгоритму над отриманим початковим станом:

$$L* = ForceDirected(L_0)$$

Модель машинного навчання не замінює процес оптимізації — вона формує якісніше початкове наближення, що дозволяє зменшити кількість необхідних ітерацій та підвищити стабільність результату.

Ефективність запропонованого підходу визначається співвідношенням між витратами на обчислення початкового розміщення за допомогою моделі машинного навчання та виграшем у швидкості роботи силового алгоритму.

Для того щоб використання ML-евристики було доцільним, повинні виконуватися дві ключові умови. По-перше, застосування моделі має призводити до зменшення кількості ітерацій силового алгоритму, необхідних для досягнення стабільного стану. По-друге, обчислювальна вартість отримання початкового розміщення повинна бути меншою за виграш, отриманий за рахунок скорочення кількості ітерацій.

Іншими словами, використання моделі машинного навчання є виправданим лише у випадку, якщо:

$$T_{ML} + T_{force}(L_0) < T_{force}(random),$$

де T_{ML} — час роботи моделі, T_{force} — час роботи силового алгоритму при відповідному початковому розміщенні.

Важливою особливістю такого підходу є те, що його ефективність залежить від розміру та структури графа. Для невеликих графів використання моделі може бути недоцільним через відносно високі накладні витрати, тоді як для графів середнього та великого розміру очікується суттєвий виграш у продуктивності.

Для реалізації запропонованого підходу було використано модель машинного навчання, призначену для апроксимації початкового розміщення вершин графа у тривимірному просторі. Враховуючи графову природу вхідних даних, доцільним є використання моделей, що безпосередньо працюють із графовими структурами.

Для навчання моделі було зібрано 1000 проєктів, написаних мовою Swift, з платформи GitHub за допомогою відкритого API.

Ми застосували графову нейронну мережу, побудовану на основі згорткових шарів (Graph Convolutional Network, GCN), яка дозволяє враховувати локальну структуру графа при формуванні представлень вершин. Як вхідні дані для моделі використовуються структурні характеристики графа, зокрема інформація про зв'язки між вершинами, а також додаткові ознаки, такі як ступінь вершини.

Результатом роботи моделі є відображення кожної вершини графа у вектор координат у тривимірному просторі, що інтерпретується як початкове розміщення для подальшої роботи силового алгоритму. Модель розв'язує задачу регресії, апроксимуючи функцію відображення графа у простір візуалізації.

Для оцінювання впливу складності моделі на якість та продуктивність було розглянуто три конфігурації нейронної мережі, що відрізняються кількістю шарів: дворівнева (L2), трирівнева (L3) та чотирирівнева (L4). Це дозволило дослідити компроміс між точністю апроксимації початкового розміщення та обчислювальною вартістю моделі.

Для оцінювання ефективності запропонованого підходу було проведено серію експериментів, спрямованих на аналіз впливу ML-евристики на швидкість та стабільність роботи силового алгоритму. Основна увага приділялася порівнянню

кількості ітерацій, необхідних для досягнення стабільного стану, а також загального часу побудови layout. Як базовий сценарій розглядався класичний силовий алгоритм із випадковим початковим розміщенням вершин (Default). Для класифікації тестових зразків використовувався розмір графа за кількістю вершин: малі — $N(V) < 50$, середні — $50 \leq N(V) \leq 300$, великі — $N(V) > 300$.

Результати експериментів для різних конфігурацій моделі наведено у таблиці 4.1.

Таблиця 4.1 Час виконання алгоритмів побудови графового представлення

Розмір графа	Модель	К-сть зразків	Середнє avg(I)	Середнє avg(t), с
Малі (<50)	Default	16	7483,27	0,30
Малі (<50)	L2	16	6397,25	0,46
Малі (<50)	L3	16	6735,81	0,54
Малі (<50)	L4	16	7470,50	0,64
Середні	Default	20	6635,42	1,25
Середні	L2	20	6519,65	1,59
Середні	L3	20	5920,05	1,36
Середні	L4	20	6726,75	2,00
Великі	Default	8	9256,29	110,69
Великі	L2	8	10000,00	112,09
Великі	L3	8	10000,00	114,16
Великі	L4	8	9594,88	113,91

Аналіз отриманих результатів показує, що для малих графів ML-евристика не забезпечує суттєвого скорочення кількості ітерацій: середнє значення avg(I) для моделей L2 та L3 залишається близьким до значення Default, а для L4 — навіть дещо зростає. Для середніх графів модель L3 демонструє найкращий результат серед усіх конфігурацій, забезпечуючи як зменшення середньої кількості ітерацій (5920 проти 6635 у Default), так і скорочення середнього часу виконання (1,36 с проти 1,25 с). Для великих графів більшість зразків досягає межі у 10 000 ітерацій як для Default, так і для ML-моделей, що обмежує можливості порівняння.

Важливою перевагою використання ML-евристики є повна стабілізація роботи алгоритму. На відміну від випадкового початкового розміщення, застосування

нейронної мережі забезпечує детерміновані результати: розмах між мінімальним та максимальним значеннями кількості ітерацій і часу виконання дорівнює нулю для всіх конфігурацій моделі. Відповідні дані наведено у таблиці 4.2.

Таблиця 4.2 Порівняння відхилень результатів роботи моделей

Модель	Середній розмах I	Середній розмах t, с	Макс. розмах I	Макс. розмах t, с
Default	3017,14	1,02	8416	16,55
L2	0,00	0,10	0	0,95
L3	0,00	0,15	0	2,17
L4	0,00	0,34	0	9,82

Як видно з наведених даних, варіативність результатів Default є суттєвою: середній розмах за кількістю ітерацій становить 3017 при максимальному значенні 8416, а розмах за часом сягає 16,55 с. Для всіх ML-конфігурацій розмах за ітераціями дорівнює нулю — кожен запуск дає однаковий результат, що робить виконання алгоритму цілком передбачуваним і дозволяє точно оцінювати очікуваний час завершення.

Відносні зміни показників кількості ітерацій та часу побудови layout відносно базового сценарію Default наведено у таблиці 4.3.

Таблиця 4.3 Порівняльна характеристика класичного та ML-орієнтованого підходів

Розмір графа	Модель	ΔI, %	Δt, %
Малі (<50)	L2	−16,40	+119,57
Малі (<50)	L3	−6,71	+164,69
Малі (<50)	L4	+1,16	+216,32
Середні	L2	−0,46	+29,37
Середні	L3	−9,32	+27,30
Середні	L4	+6,02	+53,06
Великі	L2	+10,61	+10,94
Великі	L3	+10,61	+11,92
Великі	L4	+4,70	+5,57

З наведених результатів видно, що для малих графів ML-евристика забезпечує помірне скорочення кількості ітерацій (від −6,71 % для L3 до −16,40 % для L2), проте загальний час виконання при цьому зростає — від +119 % (L2) до +216 % (L4). Це пояснюється тим, що для невеликих графів накладні витрати на обчислення нейронної мережі перевищують виграш від скорочення ітерацій силового алгоритму. Таким чином, застосування ML-евристики для малих графів є недоцільним.

Для середніх графів модель L3 забезпечує найкращий баланс між точністю апроксимації та обчислювальною вартістю: скорочення кількості ітерацій на 9,32 % при збільшенні середнього часу на 27,30 %. Водночас у 7 із 20 зразків цієї групи L3 забезпечує реальне прискорення — час виконання є меншим, ніж у Default, з максимальним скороченням до 58 % для окремих зразків. Умовою досягнення прискорення є достатньо велика кількість ітерацій Default-алгоритму: коли силовий алгоритм без евристики виконує значну кількість ітерацій, виграш від покращеного початкового розміщення перекриває вартість обчислення мережі. У решті зразків середнє збільшення часу є відносно невеликим і компенсується перевагою стабілізації.

Для великих графів переважна більшість зразків досягає ліміту у 10 000 ітерацій незалежно від евристики, що обмежує вплив початкового розміщення. Проте і в цій групі є зразки, де ML-евристика забезпечує прискорення: зокрема, для трьох зразків з восьми L2 або L4 виявляється швидшою за Default. Невелике збільшення середнього часу (+5–11 %) для більшості великих графів пояснюється витратами на обчислення мережі за відсутності пропорційного скорочення ітерацій у цих конкретних зразках.

Додатковий аналіз підтверджує, що вибір конфігурації моделі суттєво впливає на отримані результати. Модель L2 з найменшою кількістю параметрів демонструє найкраще скорочення ітерацій для малих і окремих середніх графів за рахунок нижчих обчислювальних витрат. Модель L3 є оптимальною для середніх графів, забезпечуючи найбільшу частоту реального прискорення. Для великих графів зі

складною топологією більш перспективною може виявитись модель L4, дослідження якої потребує ширшого часового вікна виконання.

Запропонований підхід забезпечує два ключові ефекти. По-перше, повну стабілізацію роботи силового алгоритму: детермінованість результатів усуває залежність від випадкового початкового стану та дозволяє точно прогнозувати час виконання. По-друге, реальне прискорення побудови layout у значній частині практичних сценаріїв — зокрема, у 7 із 20 зразків середнього розміру, де скорочення часу досягає 58 %. Застосування ML-евристики, таким чином, є обґрунтованим для графів середнього розміру та окремих сценаріїв з великими графами, де базовий алгоритм виконує значну кількість ітерацій.

4.4 Використання доповненої реальності у візуалізації графової моделі архітектури

У двовимірному випадку основною проблемою є обмеженість площини відображення. Зі збільшенням кількості вершин та ребер графа виникає перевантаження візуального простору (visual clutter), що проявляється у вигляді значної кількості перетинів, накладання елементів та зменшення відстаней між ними [92]. Це ускладнює відстеження зв’язків між компонентами та знижує інформативність візуалізації [93], [94], [95], [96].

Одним із ключових факторів є перекриття елементів (occlusion), коли вузли та ребра частково або повністю закривають один одного. У двовимірному просторі ця проблема є практично неминучою для графів середнього та великого розміру. Навіть при оптимізації кількості перетинів повністю усунути перекриття неможливо, що призводить до втрати інформації про структуру графа [97], [98], [99].

Додатковою складністю є накладання підписів (label overlap), що виникає при необхідності відображення текстової інформації, пов’язаної з вершинами. У контексті графів архітектури програмного забезпечення це має критичне значення, оскільки ідентифікація елементів є необхідною умовою їх аналізу. Обмежений

простір призводить до необхідності скорочення або приховування підписів, що знижує зрозумілість візуалізації.

Перехід до тривимірного представлення дозволяє частково зменшити щільність розміщення елементів, однак не усуває зазначених проблем повністю. Основним обмеженням є те, що відображення тривимірної сцени здійснюється на двовимірному екрані, що призводить до втрати інформації про глибину та ускладнює інтерпретацію просторових відносин між елементами. У результаті користувач змушений додатково взаємодіяти зі сценою, змінюючи точку зору для розуміння структури графа.

Крім того, тривимірні візуалізації на екрані часто створюють додаткові когнітивні труднощі. Зокрема, відсутність природного сприйняття глибини, обмеження кута огляду та необхідність керування камерою ускладнюють процес аналізу. У деяких випадках це може призводити до зниження ефективності сприйняття порівняно з двовимірними представленнями.

Одним із перспективних напрямків вирішення цієї проблеми є використання доповненої реальності. На відміну від віртуальної реальності, яка передбачає повне занурення користувача у штучне середовище, доповнена реальність інтегрує віртуальні об’єкти у фізичний простір, що дозволяє зберігати орієнтацію у навколишньому середовищі та знижує ризик дезорієнтації.

Крім того, використання доповненої реальності не потребує повної ізоляції користувача від реального світу, що робить її більш безпечною та зручною для практичного застосування, зокрема у контексті інженерного аналізу програмних систем. Це дозволяє використовувати AR як інструмент підтримки аналітичної діяльності без суттєвого порушення звичного робочого процесу.

Використання доповненої реальності відкриває нові можливості для представлення графових структур за рахунок інтеграції віртуальних об’єктів у фізичний простір. На відміну від традиційних підходів, у яких граф обмежений площиною екрана або віртуальним простором, у середовищі доповненої реальності візуалізація може бути розміщена безпосередньо у навколишньому середовищі користувача.

Однією з ключових переваг такого підходу є можливість ефективного використання тривимірного простору без обмежень, характерних для екранних інтерфейсів. У результаті зменшується щільність розміщення елементів, що дозволяє знизити рівень візуального перевантаження (visual clutter) та покращити загальну читабельність графа [97], [100], [101].

Доповнена реальність також дозволяє частково вирішити проблему перекриття елементів (occlusion). На відміну від статичних 2D або 3D-візуалізацій, користувач має можливість змінювати точку огляду шляхом фізичного переміщення у просторі. Це дозволяє досліджувати структуру графа з різних ракурсів та відновлювати приховані зв’язки без необхідності зміни самого layout.

Важливою особливістю є також підтримка природної навігації у просторі. Замість використання абстрактних засобів керування камерою користувач взаємодіє з графом через власне переміщення, що відповідає природним механізмам сприйняття тривимірних структур. Це знижує когнітивне навантаження, пов’язане з інтерпретацією просторових відношень між елементами.

Крім того, у середовищі доповненої реальності з’являється можливість формування візуальної ієрархії за рахунок використання глибини простору. Компоненти можуть бути розміщені на різних відстанях від користувача, що дозволяє виділяти ключові елементи та групи без перевантаження візуального каналу додатковими графічними засобами.

Ще одним важливим аспектом є покращення інтерпретованості структурних патернів. Кластери, сильно зв’язані компоненти або області підвищеної щільності зв’язків можуть сприйматися як просторові об’єкти, що мають форму та об’єм. Це спрощує їх ідентифікацію та аналіз порівняно з двовимірними представленнями.

Доповнена реальність не лише розширює простір візуалізації — вона змінює сам характер взаємодії користувача з графом. Це забезпечує більш природне та ефективне сприйняття складних структур, що є критично важливим у задачах аналізу архітектури програмного забезпечення.

Практичне застосування доповненої реальності у задачах аналізу архітектури програмного забезпечення потребує інтеграції засобів візуалізації з побудованою графовою моделлю системи. Цю інтеграцію ми реалізували програмним комплексом A.D.A.R., який поєднує етапи аналізу вихідного коду, побудови графа залежностей та його візуалізації у тривимірному середовищі.

Основою для візуалізації є граф залежностей, сформований на попередніх етапах аналізу. У процесі інтеграції графова модель трансформується у сцену доповненої реальності, де кожна вершина інтерпретується як тривимірний об’єкт, а ребра — як просторові зв’язки між ними. Початкове розміщення елементів визначається за допомогою силового алгоритму, потенційно покращеного із використанням ML-евристики, що забезпечує узгодженість просторової структури з топологією графа.

Важливою складовою є відображення додаткових характеристик елементів у візуальному представленні. Зокрема, значення метрик можуть бути закодовані через розмір, колір або інші візуальні атрибути об’єктів. Наприклад, вузли з високим рівнем зв’язаності можуть відображатися як більші за розміром, тоді як елементи, ідентифіковані як аномальні, можуть бути виділені кольором або іншим способом візуального підкреслення.

Інтерактивна взаємодія з графом реалізується через можливість зміни точки огляду та дослідження структури шляхом переміщення у фізичному просторі. Це дозволяє користувачу фокусуватися на окремих підграфах, аналізувати взаємозв’язки між компонентами та ідентифікувати структурні патерни без необхідності складних маніпуляцій із камерою.

Додатково може бути реалізована підтримка вибору окремих елементів, що дозволяє отримувати детальну інформацію про відповідні компоненти програмної системи. Такий підхід забезпечує зв’язок між абстрактним графовим представленням та конкретними фрагментами вихідного коду, що є важливим для практичного використання результатів аналізу.

Інтеграція доповненої реальності в A.D.A.R. поєднує результати формального аналізу архітектури з наочним просторовим поданням. Це створює єдине

середовище, у якому користувач може не лише виявляти аномалії, але й інтерпретувати їх у контексті структури програмної системи.

4.5 Висновки

Класична постановка задачі візуалізації як задачі оптимізації якості layout дозволяє формалізувати геометричні критерії, але не відображає ефективності з погляду сприйняття користувачем.

З метою підвищення ефективності силових алгоритмів запропоновано використання методів машинного навчання як евристики для формування початкового розміщення вершин. Проведений аналіз показав, що такий підхід дозволяє зменшити кількість ітерацій, підвищити стабільність результатів та, у випадку графів середнього і великого розміру, скоротити час побудови layout без зміни базових властивостей алгоритму.

Додатково було розглянуто використання доповненої реальності як засобу візуалізації, що дозволяє ефективніше використовувати тривимірний простір та зменшувати когнітивне навантаження. Показано, що інтеграція графової моделі у AR-середовище забезпечує покращення читабельності за рахунок зменшення перекриттів, підтримки природної навігації та можливості формування візуальної ієрархії.

Результатом розділу є комбінований підхід: силові алгоритми — основа побудови layout, GCN-модель — евристика для стабільного початкового розміщення, доповнена реальність — засіб просторового подання. Разом вони утворюють пайплайн, який поєднує формальну точність аналізу з інтуїтивним сприйняттям.

РОЗДІЛ 5

ПРОГРАМНИЙ КОМПЛЕКС A.D.A.R.

5.1 Архітектура програмного комплексу

Для апробації запропонованих підходів ми розробили програмний комплекс A.D.A.R., що містить інструменти для автоматизованого аналізу вихідного коду, пошуку антипатернів та перегляду згенерованої візуальної моделі. Вихідний код розробленої системи доступний у відкритому доступі на GitHub [102].

Зважаючи на швидкість, безпечність, відкритість вихідного коду та стрімкий розвиток було обрано мову Swift. Важливою перевагою Swift є унікальна будова її компілятора (рис. 5.1), що складається з трьох частин — front end, middle end та back end [103], [104].

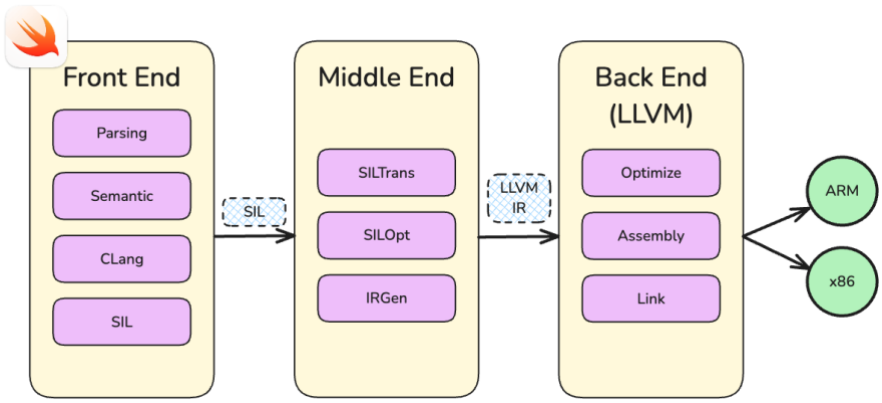


Рис. 5.1 Будова компілятора Swift

Перша частина компілятора відповідає за синтаксичний та семантичний розбір коду, інтеграцію C-подібних мов та генерацію проміжної мови SIL (Swift Intermediate Representation). На другому етапі компілятор оптимізує проміжний код відповідно до режиму збірки і готує IR (Intermediate Representation) для фінального етапу. Бекендом компілятора є LLVM [105], що після додаткових оптимізацій перетворює кодову базу в машинний код відповідно до заданої архітектури процесора та виконує компонування. Завдяки тому, що LLVM підтримує також C,

C++ та Objective-C, кодову базу проекту на Swift можна легко розширити модулями написаними іншими мовами.

Хоча мова на сьогодні є повністю крос-платформною і програми написані нею можуть працювати на macOS, Windows та багатьох популярних дистрибутивах Linux, природнім середовищем для роботи зі Swift є операційні системи компанії Apple, зокрема macOS та iOS. Саме тому ми сконцентрувались на підтримці саме цих операційних систем — macOS починаючи з версії 13, а iOS починаючи з версії 16.

Перевагою застосування macOS для нашого комплексу є не лише широка популярність цієї системи серед розробників, але й висока надійність та наявність великого різноманіття необхідних залежностей та бібліотек для роботи з мовою.

З іншого боку перевагою використання iOS є наявність всього необхідного інструментарію для реалізації програм, що дозволяють працювати з машинним навчанням та в доповненій реальності. Окрім того сучасні пристрої на базі iOS мають хороше апаратне забезпечення. Сучасні чипи дозволяють виконувати складні обчислення на спеціальних додаткових ядрах GPU та ANE (Apple Neural Engine). Іншим важливим фактором є доступність точних і швидких датчиків. Пристрої здатні швидко опрацьовувати дані з камери, акселерометра та гіроскопа, дані з яких є фундаментом для технології доповненої реальності.

При створенні програмного комплексу A.D.A.R. ми чітко визначили, що окрім стандартних вимог до швидкості, передбачуваності та підтримуваності вихідного коду, важливими є також потреба в модульності архітектури та глибокій інтеграції в існуючий інструментарій розробки.

Важливість модульності обумовлюється потребою в розширюваності системи та здатності до адаптації під потреби різних стеків технологій. Це, зокрема, стосується й підтримки аналізу інших об’єктно-орієнтованих мов програмування. Можливість імплементувати та легко інтегрувати власні реалізації окремих компонентів забезпечує можливість розширення ряду підтримуваних технологій.

З іншого боку важливою є глибока інтеграція в існуючі інструменти розробки, що спрощує застосування комплексу інженерами-програмістами в реальному

середовищі. Варто підкреслити, що добре декомпонована архітектура дозволяє уникнути надмірного зв'язування реалізації всього комплексу з конкретними технологіями, адже робота з конкретними інструментами залишається на рівні деталей імплементації окремих модулів.

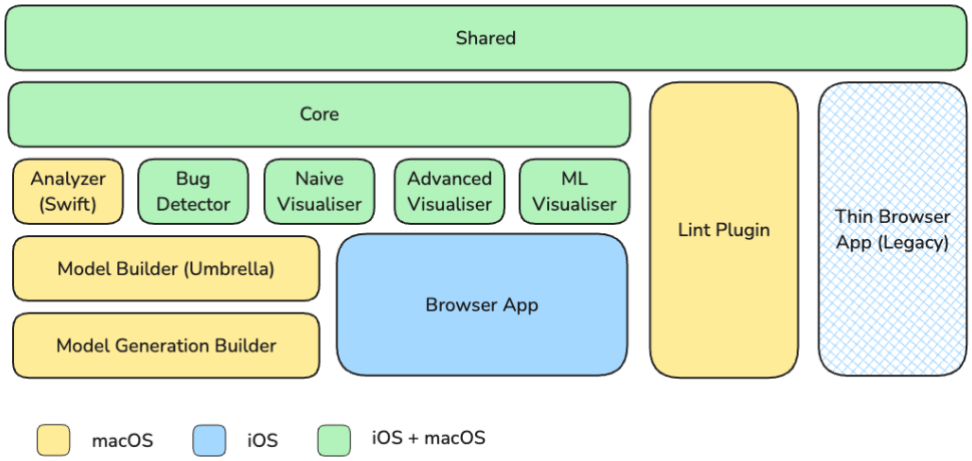


Рис. 5.2 Архітектура програмного комплексу A.D.A.R.

Наш комплекс організовано як набір пакетів Swift Package Manager (SPM), що на сьогодні є стандартом для Swift проектів. На рисунку 5.2 зображено набір модулів комплексу. На ньому нижні модулі залежать від верхніх, а підтримка конкретних операційних систем позначена відповідними кольорами. Як видно з діаграми особливістю нашої декомпозиції є можливість забезпечити сумісність більшості внутрішніх модулів з обома операційними системами.

Основою комплексу є модуль Shared, що містить декларації нашої графової моделі архітектури, а саме вузлів, ребер та додаткових структур, що містять метадані про ці елементи, зокрема і звіту про знайдені антипатерни. Фактично для роботи з моделлю та файлами, що її містять, достатньо інтегрувати лише цей модуль.

Пакет Core містить декларації протоколів (інтерфейсів) аналізатора та візуалізатора, інструменти для роботи з графами та додаткові розширення для роботи зі швидкими обчисленнями, зокрема фреймворку SIMD (Single Input Multiple Data). Цей модуль слугує основою для побудови не лише існуючих, але й майбутніх власних реалізацій аналізаторів для підтримки інших мов

програмування; візуалізаторів — для застосування нових підходів до layout; та детекторів антипатернів — для впровадження власних евристик та індикаторів архітектурних проблем. Важливо підкреслити, що Core гарантує сумісність між різними реалізаціями механізмів пайплайну аналізу/візуалізації. Натомість, для модулів, що не беруть участі в пайплайні, достатньо інтегрувати лише Shared частину.

Модуль аналізатора для мови Swift — Swift Analyzer — є одним з найцікавіших компонентів системи. Він містить реалізацію розбору вихідного коду та наповнення моделі візуалізації. Зважаючи на те, що для роботи з вихідним кодом на нашому рівні, зокрема з графом символів, цей модуль вимагає наявності компялтора Swift, тому не може працювати на мобільній платформі. Окрім того розбір коду є складним з точки зору обчислень процесом і потребує відповідних ресурсів.

Аналізатор залежно від режиму використання використовує готові суб-продукти компіляції, або ж ініціює компіляцію самостійно, обходить граф символів, а також за допомогою патерна visitor проходить по частинах абстрактних синтаксичних дерев, що містять тіла функцій. Результатом роботи аналізатора є граф компонентів з метаданими, але без обрахованого розміщення для кожного з них.

В поточній реалізації комплексу ми постачаємо одразу 3 різні реалізації візуалізатора, які можна використовувати залежно від потреб чи обмежень системи. Наївний візуалізатор імплементує класичну версію силового алгоритму, просунутий реалізовує алгоритм з оптимізацією Барнса-Хата, а візуалізатор з машинним навчанням використовує запропоновану нами евристику у формі нейронних мереж на базі CoreML і постачає набір вже готових моделей. Усі надані реалізації є взаємозамінними з точки зору інтеграції в інструмент.

Модулем, що відповідає за ще одну ключову ланку в пайплайні, є детектор антипатернів. Для створення звіту він використовує готові дані згенеровані аналізатором та візуалізатором. Це дозволяє зберегти атомарність та уникнути зайвих залежностей від деталей реалізації інших частин.

Ще одним важливим модулем є Model Builder. Цей модуль слугує обгорткою, що інкапсулює набір існуючих частин інструменту залежно від потреб кінцевого користувача. Окрім цього цей пакет є точкою синхронізації різних версій аналізаторів, візуалізаторів.

Решта модулів є програмними інструментами для кінцевих користувачів і сильно інтегровані в середовище розробки, або ж потребують специфічних залежностей чи апаратного забезпечення і тому не є крос-платформними.

Завдяки декомпозиції програмного комплексу нам також вдалось забезпечити можливість детального та повного тестування. Більшість модулів покрито не лише unit, але й інтеграційними тестами. Окрім того інтеграційними тестами покрито і стандартну конфігурацію модуля Model Builder, таким чином забезпечуючи end-to-end тестування. Загальний рівень покриття тестами всіх частин комплексу становить 95%, що відкриває можливості для безпечного інтенсивного розвитку проекту.

5.2 Інтеграція комплексу в середовище розробки

Інтеграція нових інструментів в екосистему вже існуючих є важливою умовою безперешкодного їх впровадження. В цьому контексті нашим завданням було забезпечити наявність зручних механізмів для підключення та використання інструментарію.

Природнім способом постачання допоміжних інструментів для розробки мовою Swift є спеціальні плагіни SPM. За своєю суттю це згрубша стандартні пакети з вихідним кодом, що однак можуть виконуватись в процесі збирання програм, зокрема при компіляції.

Інтеграція в процес збирання програми є простою і вимагає лише три кроки тривіальних додаткових налаштувань.

По-перше розробник повинен задекларувати залежність свого проекту від комплексу A.D.A.R. в конфігураційному файлі проекту. Після цього менеджер автоматично встановлює всі необхідні компоненти. Наша інтеграція працює автоматично з будь-якими проектами SPM та проектами інтегрованого середовища

розробки (IDE) Xcode, що дозволяє задовольнити більшість потреб нашої головної цільової аудиторії.

Другим кроком є підключення скрипта генерації візуальної моделі як власного скрипта етапу збирання проекту. Цей скрипт повинен містити лише одну команду запуску підпрограми Model Generation Builder. Завдяки цьому автоматично щоразу у разі значущих змін в проекті візуальна модель буде побудована як стандартного частина процесу збирання проекту.

Третім кроком є підключення модуля Lint Plugin для інтерактивного виводу попереджень про потенційні антипатерни прямо у вихідному коді користувача. Підключення виконується простою декларацією увімкнення плагіна.

Використання цього плагіну є опціональним, однак в процесі апробації наших результатів ми отримали чіткий запит спільноти щодо можливості працювати з детектором аномалій безпосередньо в середовищі розробки.

Після разової конфігурації користувач отримує готовий до використання файл, що містить візуальну модель архітектури модуля, а також інтерактивну підсвітку індикаторів поганого коду в IDE після кожної збірки проекту після значущих змін. Слід зауважити, що значущість змін середовище визначає автоматично з метою оптимізації ресурсів, наші інструменти інтегруються в уже існуючий і відлагоджений механізм.

Файл з готовою моделлю записується в приховану папку .adar в корені проекту. Цей файл можна легко експортувати на мобільний пристрій для перегляду в спеціальному застосунку. Окрім цього саме на базі цього файлу працює детектор антипатернів, а відповідно і індикація в коді.

З огляду на евристичний характер роботи нашого детектора ми додали гнучку систему налаштувань плагіна. Оскільки визначення доречності та правдивості індикацій для кожного конкретного випадку залишається на розсуд розробника, ми пропонуємо можливість опціонально налаштувати їх критичність.

Стандартно в Xcode існує два рівні індикацій: попередження та помилка. Зрозуміло, що ці індикації позначаються різними кольорами (жовтим та червоним, відповідно). Однак принципова відмінність полягає в тому, що наявність індикацій

помилок є умовою аварійного завершення процесу збирання. Цей механізм часто використовується для недопущення використання в коді конструкцій, які команда вважає критично небезпечними. Прикладами таких інструкцій, зокрема в стандартній конфігурації SwiftLint є доступ до nullable даних без відповідної перевірки чи явне ігнорування можливості повернення виняткової ситуації з функції.

За замовчуванням всі індикації, згенеровані нашими інструментами, вважаються попередженнями і не є критичними для процесу збирання. Однак за потреби користувач може створити в корені проекту конфігураційний файл .adarconfig, що перезаписує значення критичності (попередження, помилка чи ігнорування) для кожного виду індикацій (див. лістинг 5). Програма автоматично використовує його за наявності.

```
{
  "rules": {
    "high_degree_node": "warning",
    "low_cohesion_edge": "warning",
    "cycle_edge": "error",
```

Лістинг 5. Приклад вмісту конфігураційного файлу .adarconfig

Ще одним суттєвим механізмом контролю індикацій є підтримка спеціальних інструкцій (adar_enable rule; adar_disable:<next> rule) для увімкнення чи вимкнення конкретних попереджень в коді. Цей механізм є необхідним у разі, коли після детального аналізу розробник визначив, що попередження в цьому конкретному випадку є хибно позитивним спрацюванням евристики і підкреслює скоріше нетиповість підходів в рамках загального контексту модуля, аніж серйозну архітектурну помилку.

Приклади індикацій в IDE, а також способу вимкнення наведено в Додатку Г.

Насамкінець підмітимо, що модулі Model Generation Builder та Lint Plugin працюють на macOS і не доступні для мобільної платформи саме через специфіку

застосування. Однак з технічної точки зору за умови доступності інструментів для розробки на мобільних системах, наприклад iPadOS, в майбутньому, міграція цих модулів є тривіальною і зводиться лише до декларації підтримки відповідної операційної системи.

5.3 Деталі реалізації та методи зменшення когнітивного навантаження в програмі-переглядачі

Однією з головних цілей створення нашого інструменту є забезпечення можливості легкого сприйняття структури з високим рівнем абстракції, якою є архітектура програмного модуля. Саме тому вимоги до програми-переглядача є вкрай суворими.

Перш за все програма-переглядач повинна працювати швидко і надійно. У разі поганої технічної реалізації інструмент стає незручним в експлуатації, що фактично нівелює будь-яку користь від його застосування.

По-друге, важливим аспектом є високий рівень інтеграції в мобільну операційну систему, особливо з огляду на використання власного типу файлів. Особливістю операційних систем, на яких ми закцентували свою увагу, є високий рівень безпеки, що підтримується зокрема строгими обмеженнями доступу до певних частин операційної системи та файлів користувача. Саме тому чітка відповідність стандартам розробки та протоколам безпеки є необхідною.

По-третє, вкрай бажаною є можливість огляду візуальної моделі багатьма користувачами в режимі реального часу. Серед головних сценаріїв застосування розробленого нами інструменту окрім проведення аудиту якості є також ознайомлення нових членів команди з архітектурою модуля чи її аналіз з навчальною метою. Саме можливість синхронного перегляду єдиної візуальної моделі задовольняє потреби цих сценаріїв.

Насамкінець найважливішою вимогою до переглядача є зменшення когнітивного навантаження користувача. Створення чіткого і однозначного подання моделі архітектури з можливістю адаптації під різні потреби користувача є необхідною умовою полегшення сприйняття.

Програма-переглядач є одним з модулів нашого програмного комплексу і реалізована у вигляді iOS застосунку. Цей застосунок також працює на iPadOS, зважаючи на спорідненість цих мобільних платформ.

Для зменшення ймовірності неправильного використання, зокрема відкривання та редагування, файлів, що містять дані про візуальні моделі архітектури, ми прийняли рішення запропонувати власний формат файлів .adar. Фактично вміст файлу це документ JSON, що відповідає нашій власній схемі. Однак важливим запобіжником порушення цілісності даних є ускладнення нештатного використання файлу сторонніми програмами. Наш застосунок декларує підтримку формату файлів .adar на рівні операційної системи, що забезпечує йому пріоритетність.

В основу архітектури застосунку закладено можливість багатокористувацьких сесій перегляду. Існує два сценарії роботи переглядача: сервер та клієнт (див. п. 1 Додатку Д). Для роботи в режимі сервера користувач повинен відкрити файл з візуальною моделлю і запустити сесію доповненої реальності відповідно до простих інструкцій (навести камеру пристрою на неоднорідний простір та здійснити плавні рухи для калібрації датчиків і запуску трекінгу). В режимі клієнта користувачу не потрібен файл з візуальною моделлю, він отримає дані для відображення від застосунка-сервера при підключенні для спільної сесії. Зрозуміло, що потреба в налаштуванні сесії доповненої реальності залишається, адже дані трекінгу є унікальними для конкретного апаратного забезпечення. Пошук серверів в цьому режимі відбувається автоматично у фоновому режимі. Очевидною є можливість роботи застосунку в режимі одного користувача. В такому випадку строго кажучи застосунок є сервером без клієнтів.

Для імплементації зв’язку під пристроями в межах багатокористувацької сесії ми обрали фреймворк Multipeer Connectivity (MC). Головною його перевагою є лаконічний програмний інтерфейс API, а також автоматичне керування транспортним рівнем. Фреймворк постійно моніторить наявність та надійність зв’язку між пристроями через спільну WiFi мережу та Bluetooth і автоматично перемикається на кращий канал передачі, адаптуючись під умови середовища.

Коли додаток працює в режимі сервера він постійно опрацьовує потік даних власної сесії доповненої реальності, ефективно серіалізує їх та надсилає через сесію МС. Застосунки в режимі клієнта, що належать до однієї сесії МС натомість постійно отримують дані з сервера, розпаковують їх і завантажують у власні сесії доповненої реальності. Таким чином кожен пристрій має свою сесію доповненої реальності, що сильно залежить від апаратної частини, але опрацьовують однаковий набір даних. Оскільки ці дані містять інформацію про навколишнє середовище, точки інтересу і віртуальні об'єкти, з точки зору користувачів їхні сесії доповненої реальності виглядають єдиною спільною сесією.

Для реалізації доповненої реальності в застосунку ми обрали консервативну комбінацію фреймворків — ARKit та SceneKit. ARKit це фреймворк, що дозволяє ефективно працювати з апаратними даними і частково автоматизує трекінг середовища та рендеринг віртуальних об'єктів. SceneKit це фреймворк для ефективного рендерингу тривимірних об'єктів за допомогою апаратного забезпечення мобільних пристроїв. На сьогодні для цієї задачі більш популярним є фреймворк RealityKit, особливо з огляду на реалістичність текстур, візуальної адаптації до реального середовища та можливістю симуляції певних фізичних взаємодій. Однак в рамках нашої роботи доповнена реальність є радше інструментом взаємодії з віртуальними об'єктами, а не центральною складовою системи. В нашому випадку зменшення накладних обчислень є значно важливішим, ніж реалізм структурних елементів умовної моделі. Використання саме SceneKit дозволяє більш ефективно застосовувати обмежені ресурси пристрою для роботи з більшими за обсягом візуальними моделями.

Модель архітектури програми подається у формі віртуального доповнення середовища. З огляду на розміри моделей ми вирішили додати наочності самому поданню. Вузли, що позначають різні види елементів подаються у вигляді різних фігур: класи представлено у формі кулі, властивості — у формі конусів, а методи у формі кубів. Для інформативності модель кожного вузла доповненої інтерактивною текстовою міткою, що містить власну назву елемента. Ці інтерактивні мітки

автоматично зберігають анфасне положення відносно спостерігача при зміні положення пристрою в просторі.

Направленість ребер інструмент показує за допомогою відповідних стрілок на їхніх кінцях. Вага ребра не є явною величиною у поданні. Вага впливає на силовий алгоритм для розміщення вузлів у просторі, тому фактично довжина ребра зазвичай обернено пропорційно корелює з його вагою. Важливою інформацією про ребро є вид зв'язку, який воно позначає. Для зменшення кількості тексту в поданні ми позначаємо вид зв'язку кольором ребра.

Спосіб подання моделі архітектури наведено в п. 2 Додатку Д.

В нашому дослідженні ми ставили за мету розробити інструмент, що може працювати з довільними програмними модулями. Зрозуміло, що візуальною моделлю архітектури довільного модуля буде деякий довільний граф. В попередньому розділі ми детально проаналізували проблематику формального визначення якості візуалізації в контексті сприйняття і значну невизначеність в роботі з довільними структурами. З огляду на це ми дійшли висновку, що для зменшення когнітивного навантаження користувача візуальне представлення має бути гнучким та легко адаптуватися до потреб користувача. Для цього ми реалізували ряд технік для створення інтерактивної візуалізації.

Хоча тривимірна візуалізація частково вирішує проблему великої кількості об'єктів, що накладаються, для великих щільних графів проблема залишається. Тривіальним вирішенням цієї проблеми є зміна ракурсу. Застосування доповненої реальності завдяки природній навігації частково саме по собі вирішує завдання зміни ракурсу. Однак цього часто не достатньо, оскільки спостерігач природним чином рухається лише в горизонтальній площині. Тому ми додатково передбачили можливість зміни положення графа та масштабування.

Ми закцентували свою увагу на афінних перетвореннях (переміщення, обертання, масштабування) з кількох причин. По-перше такі перетворення не руйнують структуру графа, що особливо важливо зважаючи на цінність детермінованої природи нашої візуалізації. По-друге ці перетворення можна ефективно виконувати на графічному процесорі за допомогою операцій над

матрицями і таким чином досягти хорошої швидкодії. По-третє для цих перетворень можна легко встановити відношення в множину стандартних жестів для сенсорних екранів: для переміщення використовується дотик (tap), для обертань — протягування (pan), а для масштабування — щипок (pinch) (див. рис. 5.3).

Рис. 5.3 Стандартні жести для сенсорних екранів: дотик, протягування і щипок.

В застосунку також передбачено можливість масштабувати розмір самих елементів графа, що є корисним інструментом адаптації подання моделі залежно від щільності графа. В поєднанні з можливістю масштабування всієї структури це налаштування дозволяє тонко конфігурувати рівень інформаціного шуму на зображенні.

Природним недоліком інструментів на основі доповненої реальності є сильна залежність від середовища, зокрема освітлення. В ході апробацій ми зіштовхнулись з діаметрально протилежними потребами різних користувачів. Певні групи користувачів зазначали, що навігація в доповненій реальності є зручною, а контраст між реальними та віртуальними об’єктами допомагає краще фокусуватись. Інші групи користувачів вказували на необхідність навпаки адаптувати колір віртуальних об’єктів до загальної яскравості середовища для кращого сприйняття. Окремі користувачі зауважували загальну складність сприйняття доповненої реальності і висловлювали ідеї розмиття чи приглушування зображення реального середовища для більш чіткої роботи з віртуальними об’єктами.

У зв’язку з цим ми розробили три режими адаптації віртуальних об’єктів до реального середовища. В стандартному режимі об’єкти не адаптуються і є дуже

контрастними. Тут зауважимо, що можливість огляду середовища залишається важливою для навігації і спрощеного сприйняття. В режимі світлової адаптації програма визначає загальний рівень освітлення в кадрі і відповідно адаптує кольори віртуальних об’єктів, досягаючи ефекту однорідності. В режимі повного контрасту ми повністю виключаємо дані з камери з процесу рендерингу. Таким чином ми зберігаємо переваги навігації за допомогою переміщення пристрою в просторі, але зберігаємо зображення максимально контрастним і чітким (див. п. 3 Додатку Д).

Оскільки, як було згадано в попередньому розділі, перевантаження кольорами може несприятливо впливати на загальне сприйняття інформації ми також передбачили додатковий режим виділення лише індикаторів антипатернів в архітектурі. В цьому режимі всі вузли та ребра моноколірні, а оранжевим кольором позначаються лише ті елементи, на які відповідно до звіту детектора необхідно звернути увагу. Цей режим добре працює для сценаріїв швидкого аудиту стану архітектури без глибокого занурення (див. п. 4 Додатку Д).

Завдяки тому, що такі частини комплексу як візуалізатор і детектор антипатернів є крос-платформними, ми інтегрували їх в застосунок-переглядач. Це дозволило реалізувати інтерактивний режим роботи з підграфами (див. п. 5 Додатку Д). Нявність цього режиму важлива через те, що часто працювати з цілим графом складно через великий розмір чи високу щільність. Для побудови візуальної моделі частини архітектури користувач може обрати довільний вузол і визначити довжину найбільшого шляху між вузлами в цьому підграфі. На рівні застосунку на мобільній платформі здійснюється швидке розміщення вже цієї частини загального графа за допомогою нашого евристичного візуалізатора. Також відбувається новий аналіз зв’язків і пошук антипатернів вже на цьому рівні. Цікаво, що наявність виявлених проблем на рівні загальної моделі і їх відсутність на півні підграфа часто свідчить про неправильне застосування добре спроектованої частини системи.

Поєднання просторової навігації AR, афінних перетворень жестами, режимів адаптації освітлення та інтерактивного аналізу підграфів формує набір інструментів, що дозволяє адаптувати подання архітектури до конкретного сценарію — від швидкого аудиту стану системи до детального дослідження

окремих підсистем. Детермінована природа візуалізації при цьому зберігається: жодне з цих перетворень не змінює топологію графа.

5.4 Висновки

У розділі описано програмний комплекс A.D.A.R., розроблений для апробації та практичного застосування запропонованих у роботі методів аналізу й візуалізації архітектури програмних модулів.

Комплекс організовано як набір модулів Swift Package Manager із чітким розмежуванням відповідальностей: модуль Shared містить декларації графової моделі, Core визначає протоколи пайплайну, Swift Analyzer реалізує побудову графа на основі SymbolKit, три взаємозамінні реалізації візуалізатора (наївний, з оптимізацією Барнса-Хата та з ML-евристикою) забезпечують гнучкість вибору, а детектор антипатернів формує звіт незалежно від решти компонентів. Загальний рівень покриття тестами становить 95%, що підтверджує надійність реалізації та готовність до подальшого розвитку.

Інтеграцію комплексу в існуючий інструментарій розробки реалізовано через SPM-плагіни: Model Generation Builder автоматично будує візуальну модель як частину процесу збирання, а Lint Plugin забезпечує інтерактивну підсвітку індикаторів антипатернів безпосередньо в IDE. Гнучка система конфігурації (.adarconfig) дозволяє регулювати критичність кожного виду індикацій залежно від потреб команди.

Програма-переглядач реалізована як iOS-застосунок з підтримкою багатокористувацьких сесій через Multipeer Connectivity. Для зменшення когнітивного навантаження реалізовано афінні перетворення графа жестами, три режими адаптації до освітлення, режим монохромного виділення антипатернів та інтерактивний режим роботи з підграфами. Останній дозволяє будувати й аналізувати вибрану частину архітектури безпосередньо на пристрої, що є особливо ефективним для модулів із великою кількістю компонентів.

ВИСНОВКИ

У дисертаційній роботі вирішено актуальну науково-прикладну задачу підвищення ефективності аналізу архітектури програмних модулів шляхом розробки методів і моделей автоматизованого виявлення архітектурних антипатернів на основі графового представлення та використання когнітивно орієнтованих засобів візуалізації у середовищі доповненої реальності.

Основні результати роботи полягають у наступному:

1. Проведено аналіз існуючих підходів до дослідження архітектури програмного забезпечення, який показав, що формальні методи, зокрема метрики та статичний аналіз, дозволяють оцінювати окремі характеристики системи, однак не забезпечують цілісного уявлення про її структуру. Встановлено, що традиційні засоби візуалізації мають обмеження, пов’язані з перевантаженням візуального простору та складністю інтерпретації складних графових структур.

2. Розроблено графову модель представлення архітектури програмних модулів, яка дозволяє формалізувати структуру залежностей між компонентами системи. Запропонована модель забезпечує можливість подальшого аналізу архітектури як єдиної системи взаємопов’язаних елементів та є основою для автоматизованого виявлення архітектурних антипатернів.

3. Розроблено метод автоматизованого формування графової моделі на основі аналізу вихідного коду програмної системи. Метод забезпечує побудову графа залежностей без необхідності ручного втручання, що дозволяє застосовувати його до реальних програмних систем у процесі їх розробки та супроводження.

4. Запропоновано підхід до виявлення архітектурних антипатернів на основі аналізу структурних характеристик графа залежностей та їх статистичної оцінки. Підхід дозволяє виявляти відхилення у структурі системи шляхом порівняння характеристик елементів із типовими значеннями, що забезпечує адаптивність до особливостей конкретної програмної системи.

5. Досліджено метрики якості візуалізації графів та встановлено їх обмеження у контексті аналізу архітектури програмного забезпечення. Показано, що оптимізація

формальних метрик не гарантує покращення сприйняття структури графа користувачем, що обумовлює необхідність врахування когнітивних аспектів при побудові візуалізації.

6. Обґрунтовано доцільність використання силових алгоритмів як базового підходу до візуалізації графової моделі архітектури. Встановлено, що такі алгоритми забезпечують інтуїтивно зрозуміле відображення структури залежностей між компонентами системи та формують природні просторові конфігурації графа.

7. Запропоновано використання методів машинного навчання як евристики для формування початкового розміщення вершин графа. На основі експериментів із 44 зразками Swift-проєктів встановлено, що запропонований підхід повністю стабілізує роботу силового алгоритму: розмах результатів за кількістю ітерацій зменшується до нуля для всіх конфігурацій моделі, тоді як для випадкового початкового розміщення середній розмах становить 3017 ітерацій, а максимальний — 8416. Додатково, у 35 % зразків середнього розміру (7 із 20) модель L3 забезпечує реальне скорочення часу побудови layout — до 58 % у найкращих випадках.

8. Розроблено підхід до візуалізації архітектури програмного забезпечення у середовищі доповненої реальності. Показано, що використання тривимірного простору дозволяє зменшити когнітивне навантаження та покращити сприйняття складних структур порівняно з традиційними двовимірними підходами. Використання просторової навігації дозволяє досліджувати структуру графа з довільних ракурсів без маніпуляцій із камерою, що усуває одне з ключових обмежень традиційних 3D-інструментів на екранних інтерфейсах.

Отримані результати дозволяють сформувати узагальнений підхід до аналізу архітектури програмного забезпечення, який поєднує формалізоване графове представлення, автоматизоване виявлення архітектурних антипатернів, використання силових алгоритмів для побудови візуального представлення та застосування методів машинного навчання для підвищення ефективності обчислень. Використання доповненої реальності забезпечує врахування

особливостей сприйняття інформації користувачем та підвищує практичну корисність отриманих результатів.

Практичне значення роботи полягає у можливості застосування запропонованого підходу для аналізу архітектури реальних програмних систем, проведення архітектурних рев'ю та підтримки процесів рефакторингу. Результати роботи можуть бути використані при розробці програмних інструментів аналізу архітектури, а також у навчальному процесі підготовки фахівців у галузі програмної інженерії.

Перспективи подальших досліджень пов'язані з удосконаленням методів машинного навчання для більш точної апроксимації початкового розміщення графа, масштабуванням запропонованого підходу для обробки великих програмних систем та інтеграцією розроблених методів у повноцінні інструменти підтримки аналізу архітектури програмного забезпечення.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. S. van Wageningen, T. Mchedlidze, і A. C. Telea, «Same Quality Metrics, Different Graph Drawings», серп. 2025, Дата звернення: 2026. 29 січ. [URL]. Доступний у: https://arxiv.org/abs/2508.15557v1
2. N. Rios, R. O. Spínola, M. Mendonça, і C. Seaman, «The most common causes and effects of technical debt: First results from a global family of industrial surveys», *Int. Symp. Empir. Softw. Eng. Meas.*, жовт. 2018, doi: 10.1145/3239235.3268917.
3. International Organization for Standardization, International Electrotechnical Commission, і Institute of Electrical and Electronics Engineers, «Systems and software engineering-Vocabulary», 2017. Дата звернення: 2026. 03 лют. [URL]. Доступний у: www.iso.org
4. U. Iftikhar, N. B. Ali, J. Börstler, і M. Usman, «A tertiary study on links between source code metrics and external quality attributes», *Inf. Softw. Technol.*, т. 165, січ. 2024, doi: 10.1016/J.INFSOF.2023.107348.
5. ISO/IEC/IEEE, «ISO/IEC/IEEE 29119-1:2022(en), Software and systems engineering — Software testing — Part 1: General concepts». Дата звернення: 2026. 20 січ. [URL]. Доступний у: https://www.iso.org/obp/ui/es/#iso:std:iso-iec-ieee:29119:-1:ed-2:v1:en
6. J. P. Miguel, D. Mauricio, і G. Rodríguez, «A Review of Software Quality Models for the Evaluation of Software Products», *Int. J. Softw. Eng. Appl.*, т. 5, вип. 6, С. 31—53, листоп. 2014, doi: 10.5121/IJSEA.2014.5603.
7. A. Rahman, A. Agrawal, R. Krishna, і A. Sobran, «Characterizing the influence of continuous integration: Empirical results from 250+ open source and proprietary projects», *SWAN 2018 - Proc. 4th ACM SIGSOFT Int. Workshop Softw. Anal. Co-Located FSE 2018*, С. 8—14, листоп. 2018, doi: 10.1145/3278142.3278149.
8. D. Ståhl, T. Mårtensson, і J. Bosch, «Continuous practices and Devops: Beyond the buzz, what does it all mean?», *Proc. - 43rd Euromicro Conf. Softw. Eng. Adv. Appl. SEAA 2017*, т. 2017-January, С. 440—448, верес. 2017, doi: 10.1109/SEAA.2017.8114695.

9. E. Soares, G. Sizilio, J. Santos, D. A. da Costa, і U. Kulesza, «The effects of continuous integration on software development: a systematic literature review», *Empir. Softw. Eng.*, т. 27, вип. 3, трав. 2022, doi: 10.1007/S10664-021-10114-1.
10. S. Bernardo *ma ін.*, «Software Quality Assurance as a Service: Encompassing the quality assessment of software and services», *Future Gener. Comput. Syst.*, т. 156, С. 254—268, лип. 2024, doi: 10.1016/J.FUTURE.2024.03.024.
11. F. Losavio і L. Chirinos, «Quality Characteristics for Software Architecture *», *J. OBJECT Technol.*, т. 2, вип. 2, С. 133—150, 2003.
12. E. N. Haner Kırğıl і T. Erçelebi Ayyıldız, «Predicting Software Cohesion Metrics with Machine Learning Techniques», *Appl. Sci. 2023 Vol 13 Page 3722*, т. 13, вип. 6, С. 3722, берез. 2023, doi: 10.3390/APP13063722.
13. A. Abbad-Andaloussi, «On the relationship between source-code metrics and cognitive load: A systematic tertiary review», *J. Syst. Softw.*, т. 198, С. 111619, квіт. 2023, doi: 10.1016/J.JSS.2023.111619.
14. M. Mattsson, H. Grahn, і F. Mårtensson, «Software Architecture Evaluation Methods for Performance, Maintainability, Testability, and Portability», *Second Int. Conf. Qual. Softw. Archit. QoSA*, 2006, Дата звернення: 2026. 10 лют. [URL]. Доступний у: https://urn.kb.se/resolve?urn=urn:nbn:se:bth-9216
15. I. Fatima і P. Lago, «A Review of Software Architecture Evaluation Methods for Sustainability Assessment», *Proc. - IEEE 20th Int. Conf. Softw. Archit. Companion ICSA-C 2023*, С. 191—194, 2023, doi: 10.1109/ICSA-C57050.2023.00050.
16. M. Sahlabadi, R. C. Muniyandi, Z. Shukur, і F. Qamar, «Lightweight Software Architecture Evaluation for Industry: A Comprehensive Review», *Sensors*, т. 22, вип. 3, лют. 2022, doi: 10.3390/S22031252.
17. B. A. Kitchenham, T. Dybå, і M. Jørgensen, «Evidence-based software engineering», *Proc. - Int. Conf. Softw. Eng.*, т. 26, С. 273—281, 2004, doi: 10.1109/ICSE.2004.1317449.
18. M. Abu-Faraj, «A Survey on Software Quality Assurance», *J. Eng. Appl. Sci.*, т. 14, вип. 15, С. 5111—5122, груд. 2019, doi: 10.36478/JEASCI.2019.5111.5122.

19. J. Maarleveld, J. Guo, і D. Feitosa, «A systematic mapping study on graph machine learning for static source code analysis», *Inf. Softw. Technol.*, т. 183, С. 107722, лип. 2025, doi: 10.1016/J.INFSOF.2025.107722.
20. E. Fregnan, T. Baum, F. Palomba, і A. Bacchelli, «A survey on software coupling relations and tools», *Inf. Softw. Technol.*, т. 107, С. 159—178, берез. 2019, doi: 10.1016/J.INFSOF.2018.11.008.
21. Y. Zhang *та ін.*, «Software Architecture Recovery with Information Fusion», *ESECFSE 2023 - Proc. 31st ACM Jt. Meet. Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng.*, С. 1535—1547, листоп. 2023, doi: 10.1145/3611643.3616285/SUPPL_FILE/FSE23MAIN-P372-P-VIDEO.MP4.
22. L. L. Silva, M. T. Valente, і M. A. Maia, «Co-change patterns: A large scale empirical study», *J. Syst. Softw.*, т. 152, С. 196—214, чер. 2019, doi: 10.1016/J.JSS.2019.03.014.
23. D. Sas, P. Avgeriou, I. Pigazzini, і F. Arcelli Fontana, «On the relation between architectural smells and source code changes», *J. Softw. Evol. Process*, т. 34, вип. 1, С. e2398, січ. 2022, doi: 10.1002/SMR.2398.
24. M. Jean de Dieu, P. Liang, M. Shahin, C. Yang, і Z. Li, «Mining architectural information: A systematic mapping study», *Empir. Softw. Eng.*, т. 29, вип. 4, С. 79-, лип. 2024, doi: 10.1007/S10664-024-10480-6/METRICS.
25. D. Fuchß *та ін.*, «Who's Who? LLM-assisted Software Traceability with Architecture Entity Recognition», *ACM Trans. Auton. Adapt. Syst.*, верес. 2025, doi: 10.1145/3807453.
26. F. Palomba, G. Bavota, M. D. Penta, F. Fasano, R. Oliveto, і A. D. Lucia, «On the diffuseness and the impact on maintainability of code smells: a large scale empirical investigation», *Empir. Softw. Eng.*, т. 23, вип. 3, С. 1188—1221, чер. 2018, doi: 10.1007/S10664-017-9535-Z/TABLES/11.
27. W. Zhao *та ін.*, «Software Architecture Recovery Augmented With Semantics», *IEEE Trans. Softw. Eng.*, т. 52, вип. 01, С. 338—359, січ. 2026, doi: 10.1109/TSE.2025.3620670.

28. S. Ducasse і D. Pollet, «Software architecture reconstruction: A process-oriented taxonomy», *IEEE Trans. Softw. Eng.*, т. 35, вип. 4, С. 573—591, 2009, doi: 10.1109/TSE.2009.19.
29. T. Lutellier *та ін.*, «Comparing Software Architecture Recovery Techniques Using Accurate Dependencies», *Proc. - Int. Conf. Softw. Eng.*, т. 2, С. 69—78, серп. 2015, doi: 10.1109/ICSE.2015.136.
30. R. Li, P. Liang, M. Soliman, і P. Avgeriou, «Understanding software architecture erosion: A systematic mapping study», *J. Softw. Evol. Process*, т. 34, вип. 3, С. e2423, берез. 2022, doi: 10.1002/SMR.2423.
31. H. Mumtaz, P. Singh, і K. Blincoe, «A systematic mapping study on architectural smells detection», *J. Syst. Softw.*, т. 173, С. 110885, берез. 2021, doi: 10.1016/J.JSS.2020.110885.
32. H. S. De Andrade, E. Almeida, і I. Crnkovic, «Architectural bad smells in Software Product Lines: An exploratory study», *ACM Int. Conf. Proceeding Ser.*, 2014, doi: 10.1145/2578128.2578237.
33. M. Y. Mhawish і M. Gupta, «Predicting Code Smells and Analysis of Predictions: Using Machine Learning Techniques and Software Metrics», *J. Comput. Sci. Technol.*, т. 35, вип. 6, С. 1428—1445, листоп. 2020, doi: 10.1007/S11390-020-0323-7/METRICS.
34. R. Verdecchia, P. Kruchten, P. Lago, і I. Malavolta, «Building and evaluating a theory of architectural technical debt in software-intensive systems», *J. Syst. Softw.*, т. 176, С. 110925, чер. 2021, doi: 10.1016/J.JSS.2021.110925.
35. S. Diehl, «Software visualization: Visualizing the structure, behaviour, and evolution of software», *Softw. Vis. Vis. Struct. Behav. Evol. Softw.*, С. 1—187, 2007, doi: 10.1007/978-3-540-46505-8/SAVE-RESEARCH.
36. M. A. Storey, «Theories, tools and research methods in program comprehension: Past, present and future», *Softw. Qual. J.*, т. 14, вип. 3, С. 187—208, 2006, doi: 10.1007/S11219-006-9216-4/METRICS.

37. M. Burch, W. Huang, M. Wakefield, H. C. Purchase, D. Weiskopf, і J. Hua, «The State of the Art in Empirical User Evaluation of Graph Visualizations», *IEEE Access*, т. 9, С. 4173—4198, 2021, doi: 10.1109/ACCESS.2020.3047616.
38. Paul. Clements, Rick. Kazman, і Mark. Klein, «Evaluating software architectures : methods and case studies», С. 323, 2009.
39. E. B. Allen і T. M. Khoshgoftaar, «Measuring coupling and cohesion: An information-theory approach», *Int. Softw. Metr. Symp. Proc.*, С. 119—127, 1999, doi: 10.1109/METRIC.1999.809733.
40. A. S. Abdelfattah, T. Cerny, D. Taibi, і S. Vegas, «Comparing 2D and Augmented Reality Visualizations for Microservice System Understandability: A Controlled Experiment», *IEEE Int. Conf. Program Comprehension*, т. 2023-May, С. 135—145, 2023, doi: 10.1109/ICPC58990.2023.00028.
41. M. Alshakhouri, J. Buchan, і S. G. MacDonell, «Synchronised Visualisation of Software Process and Product Artefacts: Concept, Design and Prototype Implementation», *Inf. Softw. Technol.*, т. 98, С. 131—145, трав. 2021, doi: 10.1016/j.infsof.2018.01.008.
42. R. Wettel і M. Lanza, «Code city: 3D visualization of large-scale software», *Proc. - Int. Conf. Softw. Eng.*, С. 921—922, 2008, doi: 10.1145/1370175.1370188.
43. R. Koschke, «Software visualization in software maintenance, reverse engineering, and re-engineering», *J. Softw. Maint. Res. Pract.*, т. 15, вип. 2, С. 87—109, берез. 2003, doi: 10.1002/SMR.270.
44. N. Chotisarn *ma ін.*, «A systematic literature review of modern software visualization», *J. Vis.*, т. 23, вип. 4, С. 539—558, серп. 2020, doi: 10.1007/S12650-020-00647-W/METRICS.
45. V. Dashuber і M. Philippsen, «Trace visualization within the Software City metaphor: Controlled experiments on program comprehension», *Inf. Softw. Technol.*, т. 150, С. 106989, жовт. 2022, doi: 10.1016/J.INFSOF.2022.106989.
46. P. Rachow і M. Riebisch, «An Architecture Smell Knowledge Base for Managing Architecture Technical Debt», *Proc. - Int. Conf. Tech. Debt 2022 TechDebt 2022*, С. 1—10, серп. 2024, doi: 10.1145/3524843.3528092.

47. W. Hasselbring, A. Krause, і C. Zirkelbach, «ExplorViz: Research on software visualization, comprehension and collaboration», *Softw. Impacts*, т. 6, С. 100034, листоп. 2020, doi: 10.1016/J.SIMPA.2020.100034.
48. A. Tornhill і M. Borg, «Code Red: The Business Impact of Code Quality - A Quantitative Study of 39 Proprietary Production Codebases», *Proc. - Int. Conf. Tech. Debt 2022 TechDebt 2022*, С. 11—20, серп. 2024, doi: 10.1145/3524843.3528091.
49. О. Франків, «Використання доповненої реальності для візуалізації архітектур програмних модулів», *Наукові записки НаУКМА Комп'ютерні науки*, т. 5, С. 26—30, лют. 2022, doi: 10.18523/2617-3808.2022.5.26-30.
50. О. Франків і М. Глибовець, «Автоматизована візуалізація компонентів архітектури програми для мови Swift», *Кібернетикаа та системний аналіз*, С. 190—198, 2024, doi: 10.34229/KCA2522-9664.24.6.16.
51. S. R. Chidamber і C. F. Kemerer, «A Metrics Suite for Object Oriented Design», *IEEE Trans. Softw. Eng.*, т. 20, вип. 6, С. 476—493, 1994, doi: 10.1109/32.295895.
52. Y. Mei *та ін.*, «Deriving Thresholds of Object-Oriented Metrics to Predict Defect-Proneness of Classes: A Large-Scale Meta-Analysis», *https://doi.org/10.1142/S0218194023500110*, т. 33, вип. 5, С. 651—695, квіт. 2023, doi: 10.1142/S0218194023500110.
53. Martin. Fowler і Kent. Beck, «Refactoring : improving the design of existing code», 2019.
54. Z. Wu, X. Chen, і S. U. J. Lee, «A systematic literature review on Android-specific smells», *J. Syst. Softw.*, т. 201, С. 111677, лип. 2023, doi: 10.1016/J.JSS.2023.111677.
55. О. О. Франків, «Візуалізація архітектур програмних модулів для виявлення вад проєктування», *Когнітивні дослідження результати виклики та перспективи матеріали першої всеукраїнської наукової конференції*, С. 321—324, трав. 2024.
56. F. Palomba, R. Oliveto, і A. De Lucia, «Investigating code smell co-occurrences using association rule learning: A replicated study», *MaLTeSQuE 2017 - IEEE Int. Workshop Mach. Learn. Tech. Softw. Qual. Eval. Co-Located SANER 2017*, С. 8—13, берез. 2017, doi: 10.1109/MALTESQUE.2017.7882010.

57. D. Yu, Y. Xu, L. Weng, J. Chen, X. Chen, і Q. Yang, «Detecting and Refactoring Feature Envy Based on Graph Neural Network», *Proc. - Int. Symp. Softw. Reliab. Eng. ISSRE*, т. 2022-October, С. 458—469, 2022, doi: 10.1109/ISSRE55969.2022.00051.
58. A. Sousa, L. Rocha, і R. Britto, «Architectural Technical Debt - A Systematic Mapping Study», *ACM Int. Conf. Proceeding Ser.*, С. 196—205, верес. 2023, doi: 10.1145/3613372.3613399.
59. P. Gnoyke, S. Schulze, і J. Krüger, «Evolution patterns of software-architecture smells: An empirical study of intra- and inter-version smells», *J. Syst. Softw.*, т. 217, С. 112170, листоп. 2024, doi: 10.1016/J.JSS.2024.112170.
60. R. C. Martin, «Agile Software Development,Principles, Patterns, and Practices 1/E», 2014, Дата звернення: 2026. 15 січ. [URL]. Доступний у: www.EBooksWorld.ir
61. P. Avgeriou *ma ін.*, «Technical Debt Management: The Road Ahead for Successful Software Delivery», *Proc. - 2023 IEEEACM Int. Conf. Softw. Eng. Future Softw. Eng. ICSE-FoSE 2023*, С. 15—30, 2023, doi: 10.1109/ICSE-FOSE59343.2023.00007.
62. А. А. Суліменко, О. О. Франків, і А. А. Нагнибіда, «Програмний комплекс Stability Assurance Tool: еволюція та розвиток для автоматизованої оцінки стабільності та зрозумілості коду Swift», *Наукові записки НаУКМА Компютерні науки*, т. 8, С. 232—237, листоп. 2025, doi: 10.18523/2617-3808.2025.8.232-237.
63. S. L. Pfleeger і B. A. Kitchenham, «Evidence-Based Software Engineering Guidelines Revisited», *IEEE Trans. Softw. Eng.*, т. 51, вип. 3, С. 814—819, 2025, doi: 10.1109/TSE.2025.3526730.
64. I. Pigazzini, F. A. Fontana, і B. Walter, «A study on correlations between architectural smells and design patterns», *J. Syst. Softw.*, т. 178, С. 110984, серп. 2021, doi: 10.1016/J.JSS.2021.110984.
65. Len. Bass, Paul. Clements, і Rick. Kazman, «Software Architecture in Practice, 4th Edition», С. 464, 2021.
66. A. Ouellet і M. Badri, «Combining object-oriented metrics and centrality measures to predict faults in object-oriented software: An empirical validation», *J. Softw. Evol. Process*, т. 36, вип. 4, С. e2548, квіт. 2024, doi: 10.1002/SMR.2548.

67. J. Al Dallal, «Improving the applicability of object-oriented class cohesion metrics», *Inf. Softw. Technol.*, т. 53, вип. 9, С. 914—928, верес. 2011, doi: 10.1016/J.INFSOF.2011.03.004.
68. E. N. H. Kirğil і T. E. Ayyildiz, «Analysis of Lack of Cohesion in Methods (LCOM): A Case Study», у *2021 2nd International Informatics and Software Engineering Conference (IISEC)*, груд. 2021, С. 1—4. doi: 10.1109/IISEC54230.2021.9672419.
69. M. Lanza і R. Marinescu, «Object-oriented metrics in practice: Using software metrics to characterize, evaluate, and improve the design of object-oriented systems», *Object-Oriented Metr. Pract. Using Softw. Metr. Charact. Eval. Improve Des. Object-Oriented Syst.*, С. 1—205, 2006, doi: 10.1007/3-540-39538-5/SAVE-RESEARCH.
70. T. Sharma, P. Singh, і D. Spinellis, «An empirical investigation on the relationship between design and architecture smells», *Empir. Softw. Eng.*, т. 25, вип. 5, С. 4020—4068, верес. 2020, doi: 10.1007/S10664-020-09847-2.
71. О. О. Франків, «Автоматизоване виявлення вад архітектури програмного модуля з використанням графової моделі візуалізації», *Наукові записки НаУКМА Комп'ютерні науки*, т. 8, С. 167—173, листоп. 2025, doi: 10.18523/2617-3808.2025.8.167-173.
72. H. C. Purchase, «Metrics for Graph Drawing Aesthetics», *J. Vis. Lang. Comput.*, т. 13, вип. 5, С. 501—516, жовт. 2002, doi: 10.1006/JVLC.2002.0232.
73. C. Ware, H. Purchase, L. Colpoys, і M. McGill, «Cognitive measurements of graph aesthetics», *Inf. Vis.*, т. 1, вип. 2, С. 103—110, чер. 2002, doi: 10.1057/PALGRAVE.IVS.9500013.
74. V. Yoghourdjian, Y. Yang, T. Dwyer, L. Lawrence, M. Wybrow, і K. Marriott, «Scalability of Network Visualisation from a Cognitive Load Perspective», *IEEE Trans. Vis. Comput. Graph.*, т. 27, вип. 2, С. 1677—1687, лют. 2021, doi: 10.1109/TVCG.2020.3030459.
75. T. M. J. Fruchterman і E. M. Reingold, «Graph drawing by force-directed placement», *Software—Practice Exp.*, т. 21, вип. 11, С. 1129—1164, листоп. 1991, doi: 10.1002/SPE.4380211102.

76. F. Zhong *ma ін.*, «Force-Directed Graph Layouts Revisited: A New Force Based on the T-Distribution», *IEEE Trans. Vis. Comput. Graph.*, т. 30, вип. 7, С. 3650—3663, лип. 2024, doi: 10.1109/TVCG.2023.3238821.
77. C. Both, N. Dehmamy, R. Yu, і A. L. Barabási, «Accelerating network layouts using graph neural networks», *Nat. Commun. 2023 141*, т. 14, вип. 1, С. 1560-, берез. 2023, doi: 10.1038/s41467-023-37189-2.
78. J. Barnes, P. Hut, J. Barnes, і P. Hut, «A hierarchical O(N log N) force-calculation algorithm», *Natur*, т. 324, вип. 6096, С. 446—449, 1986, doi: 10.1038/324446A0.
79. J. X. Zheng, S. Pawar, і D. F. M. Goodman, «Graph Drawing by Stochastic Gradient Descent», *IEEE Trans. Vis. Comput. Graph.*, т. 25, вип. 09, С. 2738—2748, верес. 2019, doi: 10.1109/TVCG.2018.2859997.
80. F. Grötschla, J. Mathys, R. Veres, R. Wattenhofer, і E. Zurich, «CoRe-GD: A Hierarchical Framework for Scalable Graph Visualization with GNNs», *Int. Conf. Learn. Represent.*, т. 2024, С. 37081—37103, трав. 2024.
81. O. H. Kwon, T. Crnovrsanin, і K. L. Ma, «What Would a Graph Look Like in this Layout? A Machine Learning Approach to Large Graph Visualization», *IEEE Trans. Vis. Comput. Graph.*, т. 24, вип. 1, С. 478—488, січ. 2018, doi: 10.1109/TVCG.2017.2743858.
82. Q. Wang, Z. Chen, Y. Wang, і H. Qu, «A Survey on ML4VIS: Applying Machine Learning Advances to Data Visualization», *IEEE Trans. Vis. Comput. Graph.*, т. 28, вип. 12, С. 5134—5153, груд. 2022, doi: 10.1109/TVCG.2021.3106142.
83. A. Grover і J. Leskovec, «Node2vec: Scalable feature learning for networks», *Proc. ACM SIGKDD Int. Conf. Knowl. Discov. Data Min.*, т. 13-17-August-2016, С. 855—864, серп. 2016, doi: 10.1145/2939672.2939754.
84. HiraokaYasuaki, ImotoYusuke, MeehanKillian, і YachimuraToshiaki, «Topological node2vec», *J. Mach. Learn. Res.*, січ. 2024, doi: 10.5555/3722577.3722711.
85. R. Huang, H. Gai, R. Jing, і F. Wan, «Word Spectral Visualization Base on Fruchterman-Reingold Optimized ForceAtlas2», *J. Electr. Syst.*, т. 20, вип. 2, С. 07—15, квіт. 2024, doi: 10.52783/JES.1086.

86. M. K. Rahman, M. H. Sujon, і A. Azad, «Force2Vec: Parallel Force-Directed Graph Embedding», *2020 IEEE Int. Conf. Data Min. ICDM*, т. 2020-November, С. 442—451, листоп. 2020, doi: 10.1109/ICDM50108.2020.00053.
87. Y. Wang, Z. Jin, Q. Wang, W. Cui, T. Ma, і H. Qu, «DeepDrawing: A Deep Learning Approach to Graph Drawing», *IEEE Trans. Vis. Comput. Graph.*, т. 26, вип. 1, С. 676—686, січ. 2020, doi: 10.1109/TVCG.2019.2934798.
88. L. Giovannangeli, F. Lalanne, D. Auber, R. Giot, і R. Bourqui, «Toward Efficient Deep Learning for Graph Drawing (DL4GD)», *IEEE Trans. Vis. Comput. Graph.*, т. 30, вип. 2, С. 1516—1532, лют. 2024, doi: 10.1109/TVCG.2022.3222186.
89. X. Wang, K. Yen, Y. Hu, і H. W. Shen, «SmartGD: A GAN-Based Graph Drawing Framework for Diverse Aesthetic Goals», *IEEE Trans. Vis. Comput. Graph.*, т. 30, вип. 8, С. 5666—5678, 2024, doi: 10.1109/TVCG.2023.3306356.
90. О. Франків, «Машинне навчання в покращенні візуалізації просторової моделі архітектури програми», *Вісник Київського національного університету імені Тараса Шевченка фізико-математичні науки*, т. 80, вип. 1, С. 164—173, лип. 2025, doi: 10.17721/1812-5409.2025/1.22.
91. C. Walshaw, «A Multilevel Algorithm for Force-Directed Graph-Drawing», *J. Graph Algorithms Appl.*, т. 7, вип. 3, С. 253—285, січ. 2003, doi: 10.7155/JGAA.00070.
92. W. Huang, S. H. Hong, і P. Eades, «Effects of crossing angles», *IEEE Pac. Vis. Symp. 2008 PacificVis - Proc.*, С. 41—46, 2008, doi: 10.1109/PACIFICVIS.2008.4475457.
93. T. Kosch, J. Karolus, J. Zagermann, H. Reiterer, A. Schmidt, і P. W. WoÅ°niak, «A Survey on Measuring Cognitive Workload in Human-Computer Interaction», *ACM Comput. Surv.*, т. 55, вип. 13s, груд. 2023, doi: 10.1145/3582272/ASSET/A51C32E5-CEC3-46D7-8590-B868B7B6AB38/ASSETS/GRAPHIC/CSUR-2021-0464-INLINE3.JPG.
94. R. Tamassia, «Handbook of graph drawing and visualization», *Handb. Graph Draw. Vis.*, С. 1—839, серп. 2013, doi: 10.1201/b15385.
95. V. Filipov, A. Arleo, і S. Miksch, «Are We There Yet? A Roadmap of Network Visualization from Surveys to Task Taxonomies», *Comput. Graph. Forum*, т. 42, вип. 6, С. e14794, верес. 2023, doi: 10.1111/CGF.14794.

96. Y. Suzuki, F. Wild, і E. Scanlon, «Measuring cognitive load in augmented reality with physiological methods: A systematic review», *J. Comput. Assist. Learn.*, т. 40, вип. 2, С. 375—393, квіт. 2024, doi: 10.1111/JCAL.12882.
97. R. Zhang і O. S. Son, «A Measurement Model for Visual Complexity in HCI: Focusing on Visual Elements in Mobile GUI Design», *Electron. 2025 Vol 14 Page 942*, т. 14, вип. 5, С. 942, лют. 2025, doi: 10.3390/ELECTRONICS14050942.
98. T. von Landesberger *ma ін.*, «Visual Analysis of Large Graphs: State-of-the-Art and Future Research Challenges», *Comput. Graph. Forum*, т. 30, вип. 6, С. 1719—1749, верес. 2011, doi: 10.1111/J.1467-8659.2011.01898.X.
99. A. Krause-Glau, M. Hansen, і W. Hasselbring, «Collaborative program comprehension via software visualization in extended reality», *Inf. Softw. Technol.*, т. 151, С. 107007, листоп. 2022, doi: 10.1016/J.INFSOF.2022.107007.
100. J. Hatvani, D. Csatári, M. Á. Fehér, Á. Várhidy, J. Seo, і G. Cserey, «Enhancing human spatial awareness through augmented reality technologies», *Biomed. Eng. Lett.*, т. 15, вип. 6, С. 995, листоп. 2025, doi: 10.1007/S13534-025-00502-7.
101. N. Wenk, J. Penalver-Andres, K. A. Buetler, T. Nef, R. M. Müri, і L. Marchal-Crespo, «Effect of immersive visualization technologies on cognitive load, motivation, usability, and embodiment», *Virtual Real. 2021 271*, т. 27, вип. 1, С. 307—331, серп. 2021, doi: 10.1007/S10055-021-00565-8.
102. О. Франків, *ADAR - Architecture Displayer in AR*. (2026. 01 чер.). Swift. Дата звернення: 2026. 03 чер. [URL]. Доступний у: https://github.com/gf0x/ADAR
103. «Swift Intermediate Language (SIL) — Swift 2.2 documentation». Дата звернення: 2026. 03 чер. [URL]. Доступний у: https://apple-swift.readthedocs.io/en/latest/SIL.html
104. M. Kraus і V. Haupert, «The Swift Language from a Reverse Engineering Perspective», у *Proceedings of the 2nd Reversing and Offensive-oriented Trends Symposium*, у ROOTS ’18. New York, NY, USA: Association for Computing Machinery, листоп. 2018, С. 1—12. doi: 10.1145/3289595.3289599.
105. C. Lattner і V. Adve, «LLVM: a compilation framework for lifelong program analysis & transformation», у *International Symposium on Code Generation and*

Optimization, 2004. CGO 2004., берез. 2004, С. 75—86. doi: 10.1109/CGO.2004.1281665.

ДОДАТКИ

ДОДАТОК А

СПИСОК ПУБЛІКАЦІЙ ЗДОБУВАЧА

Наукові праці, в яких опубліковані основні наукові результати дисертації:

1. Франків, О. (2022). Використання доповненої реальності для візуалізації архітектур програмних модулів. Наукові записки НаУКМА. Комп'ютерні науки, 5, 26—30. https://doi.org/10.18523/2617-3808.2022.5.26-30
2. Франків, О., & Глибовець, М. (2024). Автоматизована візуалізація компонентів архітектури програми для мови Swift. Кібернетикаа та системний аналіз, 190—198. https://doi.org/10.34229/KCA2522-9664.24.6.16
3. Франків, О. (2025). Машинне навчання в покращенні візуалізації просторової моделі архітектури програми. Вісник Київського національного університету імені Тараса Шевченка. Фізико-математичні науки, 80(1), 164—173. https://doi.org/10.17721/1812-5409.2025/1.22
4. Франків, О. О. (2025). Автоматизоване виявлення вад архітектури програмного модуля з використанням графової моделі візуалізації. Наукові Записки НаУКМА. Комп'ютерні Науки, 8, 167—173. https://doi.org/10.18523/2617-3808.2025.8.167-173

Наукові праці, які засвідчують апробацію матеріалів дисертації:

1. Франків, О. О. (2024). Візуалізація архітектур програмних модулів для виявлення вад проєктування. У В. Є. Снитюк (Ред.), Когнітивні дослідження: результати, виклики та перспективи: матеріали Першої всеукраїнської наукової конференції (24 травня 2024 р., Київ, Україна) (с. 321–324). Видавництво «Каравела». ISBN 978-960-801-899-0.
2. Франків, О. О. (2024). Машинне навчання для візуалізації моделей архітектур програм на мобільних пристроях. У М. М. Глибовець & Т. В. Панченко (Ред.), Теоретичні та прикладні аспекти побудови програмних систем: праці 15

міжнародної науково-практичної конференції (23–24 грудня 2024 р., Київ, Україна) (с. 54–56). Видавництво НаУКМА.

3. Франків, О. О. (2025). Вдосконалення візуалізації моделей архітектур програм для зменшення когнітивного навантаження користувача. У М. М. Глибовець & Т. В. Панченко (Ред.), Теоретичні та прикладні аспекти побудови програмних систем: праці 16 міжнародної науково-практичної конференції (24–25 листопада 2025 р., Київ, Україна) (с. 58–60). Видавництво НаУКМА.

Наукові праці, які додатково відображають наукові результати дисертації:

1. Суліменко, А. А., Франків, О. О., & Нагнибіда, А. А. (2025). Програмний комплекс Stability Assurance Tool: Еволюція та розвиток для автоматизованої оцінки стабільності та зрозумілості коду Swift. Наукові записки НаУКМА. Комп'ютерні науки, 8, 232—237. https://doi.org/10.18523/2617-3808.2025.8.232-237

ДОДАТОК Б

ЗАТВЕРДЖУЮ
Директор ТОВ "Девелопекс",
Лимар Олександр
«10» квітня 2026 р.

АКТ
про дослідне випробування результатів
дисертаційного дослідження здобувача кафедри інформатики
Національного університету «Києво-Могилянська академія»
Франківа Олександра Олександровича

Даний акт складений про те, що програмний комплекс ADAR (Architecture Displayer in Augmented Reality) — інструмент для автоматизованої візуалізації архітектури програмних модулів з використанням технологій доповненої реальності, розроблений під час виконання дисертаційної роботи Франківа Олександра Олександровича на тему «Автоматизований аналіз та візуалізація в доповненій реальності архітектури програмних модулів для виявлення антипатернів», пройшов дослідне випробування у ТОВ "Девелопекс" (м. Київ).

Запропонований програмний комплекс дозволяє автоматизовано аналізувати вихідний код програмного модуля та будувати його графову модель архітектури без додаткової проєктної документації. Розроблений модуль виявляє структурні вади проєктування: компоненти з надмірною зв'язністю, низькою згуртованістю та циклічні залежності між компонентами. Застосування комплексу значно прискорило процес аудиту та аналізу програмних проєктів, дозволивши виявляти потенційно проблемні ділянки коду в рази швидше порівняно з ручним аналізом. Отримані результати підтверджено детальним ручним аналізом вихідного коду реальних iOS-проєктів.

Використання запропонованого програмного комплексу дозволить розробникам програмного забезпечення інтуїтивно виявляти та усувати структурні вади архітектури безпосередньо на основі вихідного коду, без необхідності розробки чи підтримки окремої проєктної документації, а також інтегрувати засоби візуального аналізу архітектури у процес розроблення програмного забезпечення.

Даний акт не є підставою для взаємних фінансових розрахунків.

ДОДАТОК В

УКРАЇНА
МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «КИЄВО-МОГИЛЯНСЬКА АКАДЕМІЯ»
04070, м. Київ, вул. Г. Сковороди, 2, тел.: (044)425-60-59, www.ukma.edu.ua, Код ЄДРПОУ 16459396

22.06.2026 № 14/851

на № _____

АКТ

про впровадження результатів дисертації на здобуття ступеня доктора філософії Франківа Олександра Олександровича «Автоматизований аналіз та візуалізація в доповненій реальності архітектури програмних модулів для виявлення антипатернів»

за спеціальністю

122 - комп'ютерні науки

Цей акт складено про те, що результати дисертаційного дослідження Франківа Олександра Олександровича на тему «Автоматизований аналіз та візуалізація в доповненій реальності архітектури програмних модулів для виявлення антипатернів», виконаного на здобуття наукового ступеня доктора філософії за спеціальністю 122 «Комп'ютерні науки» під керівництвом д.ф.-м.н., професора Глибовця М.М., впроваджено в навчальний процес сертифікатної програми «Програмування для iOS» Національного університету «Києво-Могилянська академія» для студентів спеціальностей 121 “Інженерія програмного забезпечення” та 122 “Комп’ютерні науки”.

Впровадження зазначених результатів сприяє формуванню у студентів практичних навичок архітектурного аналізу iOS-проєктів, розуміння структурних вад програмних систем та обґрунтованого підходу до рефакторингу коду.

Віцепрезидент з науково-педагогічної роботи Національного університету «Києво-Могилянська академія» Луцишин З.О.

2102

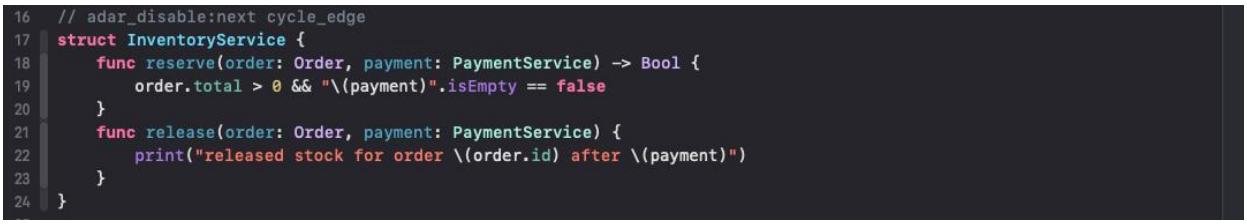
ДОДАТОК Г

Приклади індикації антипатернів у вихідному коді

1. Попереджувальна індикація

2. Індикація критичної помилки

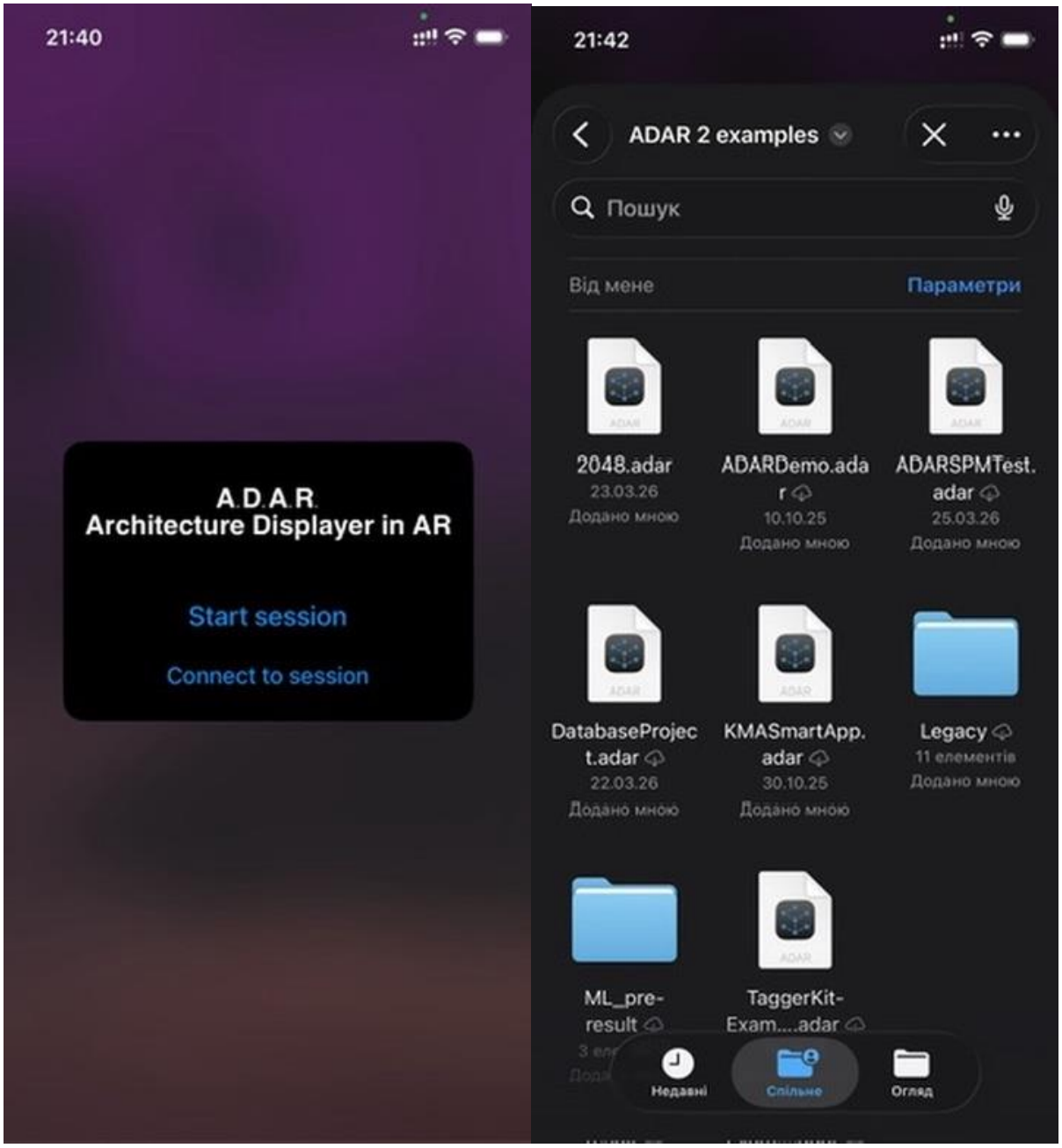
3. Вимкнення індикацій за допомогою директиви у коді



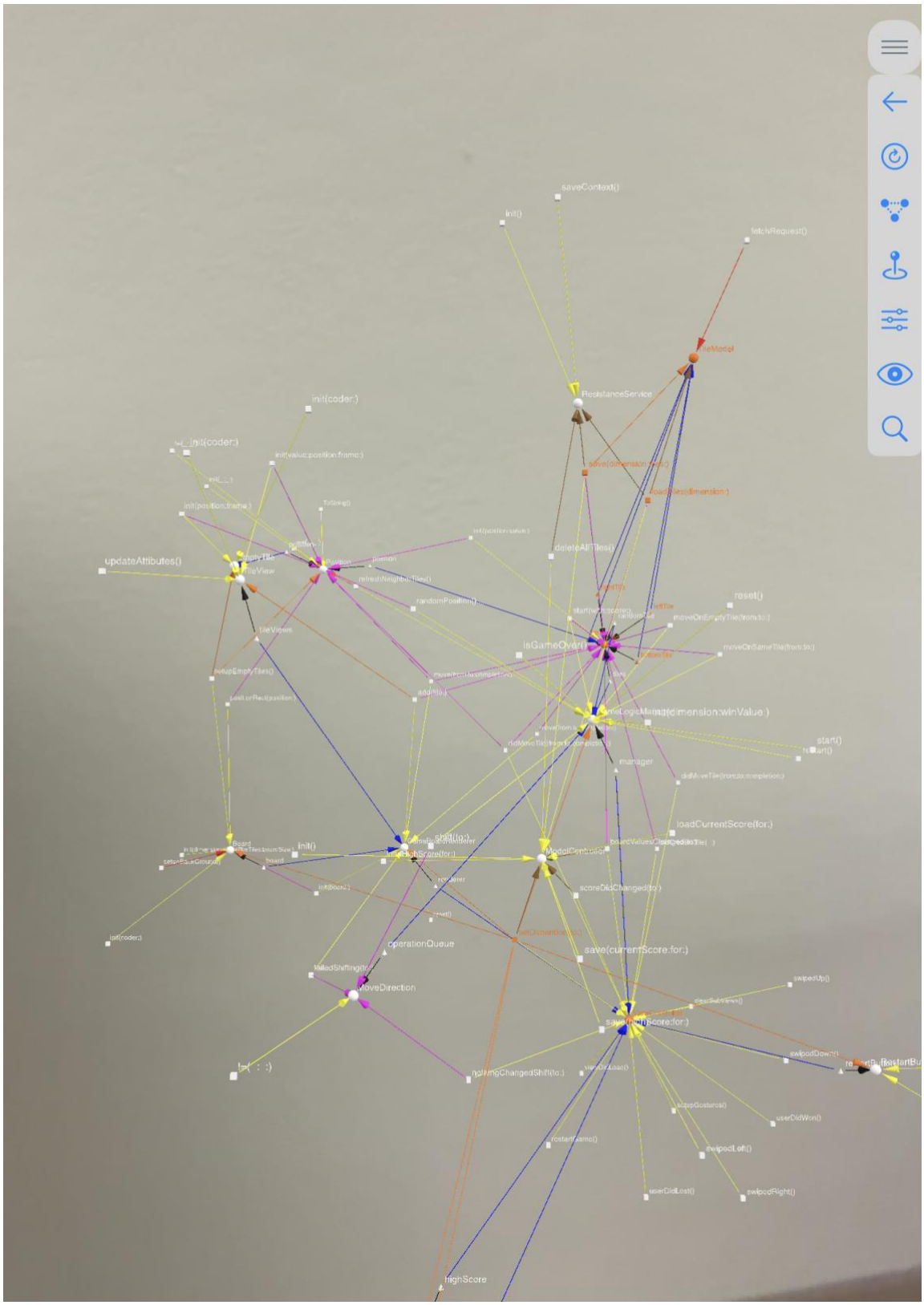
ДОДАТОК Д

Знімки екрану програми-переглядача A.D.A.R.

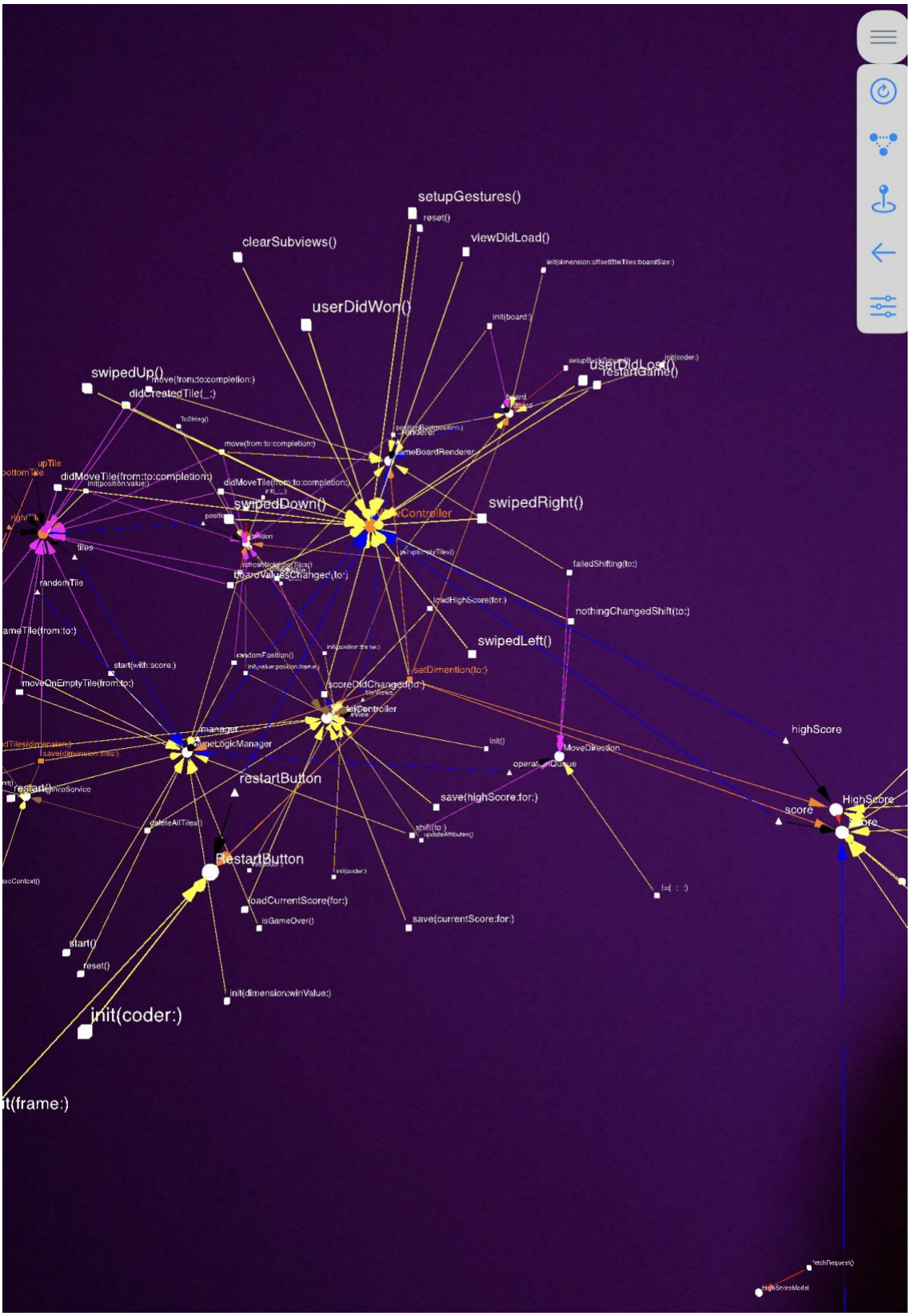
1. Вибір режиму роботи застосунку та вибір файлів для перегляду.



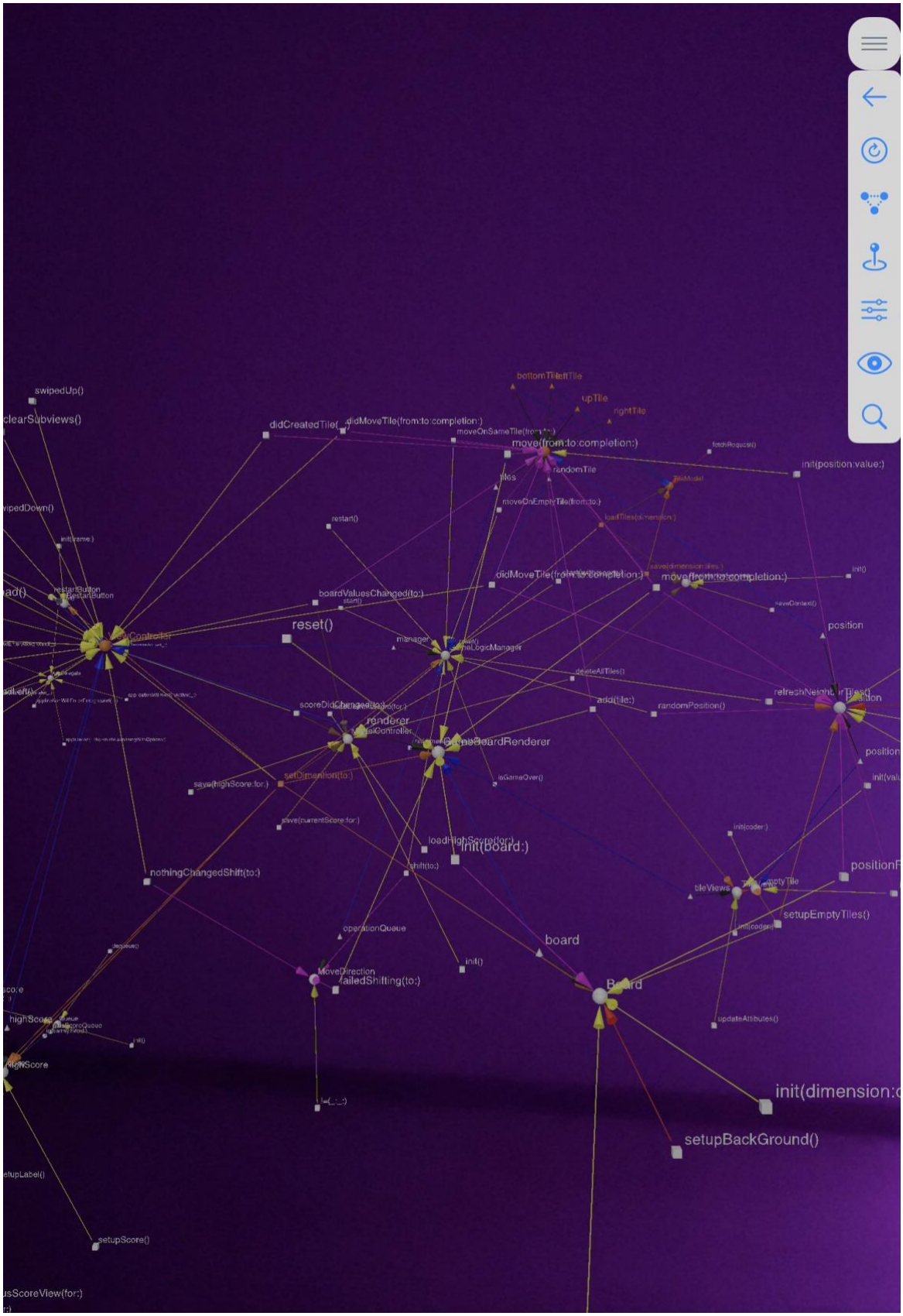
2. Стандартний спосіб подання візуальної моделі архітектури в програмі-переглядачі.



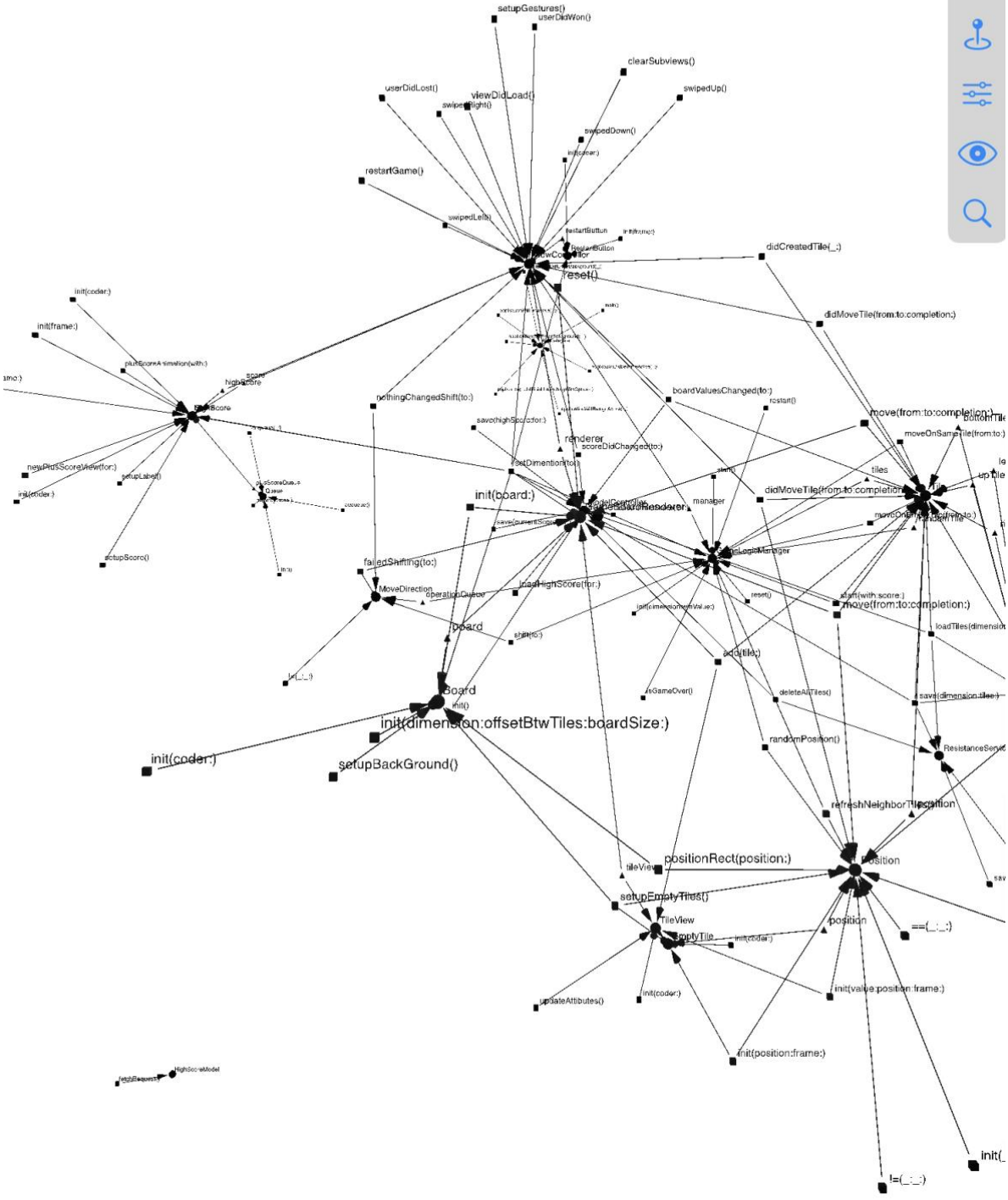
3. Режими адаптації віртуального контенту до реального середовища.
    a. Стандартний режим



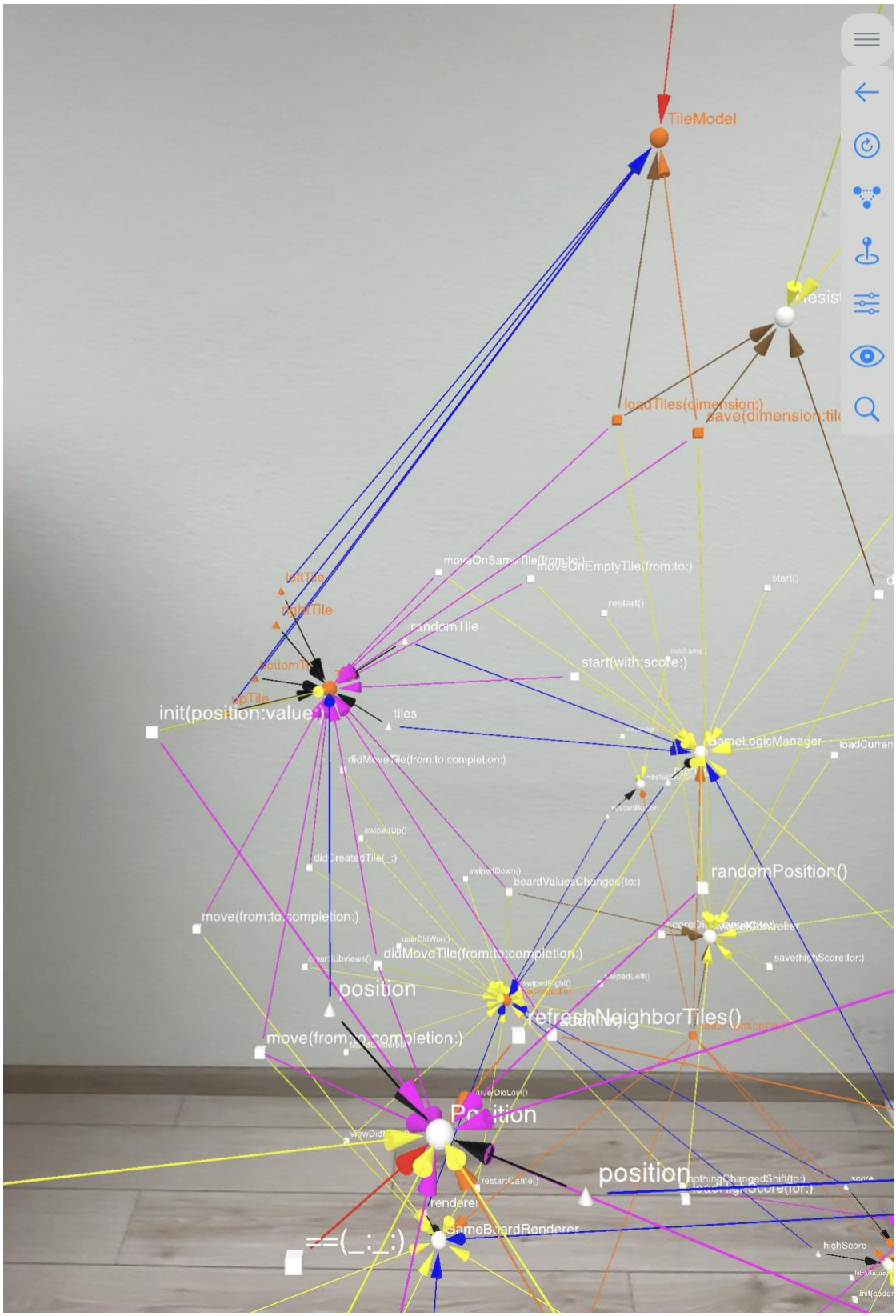
b. Режим світлової адаптації



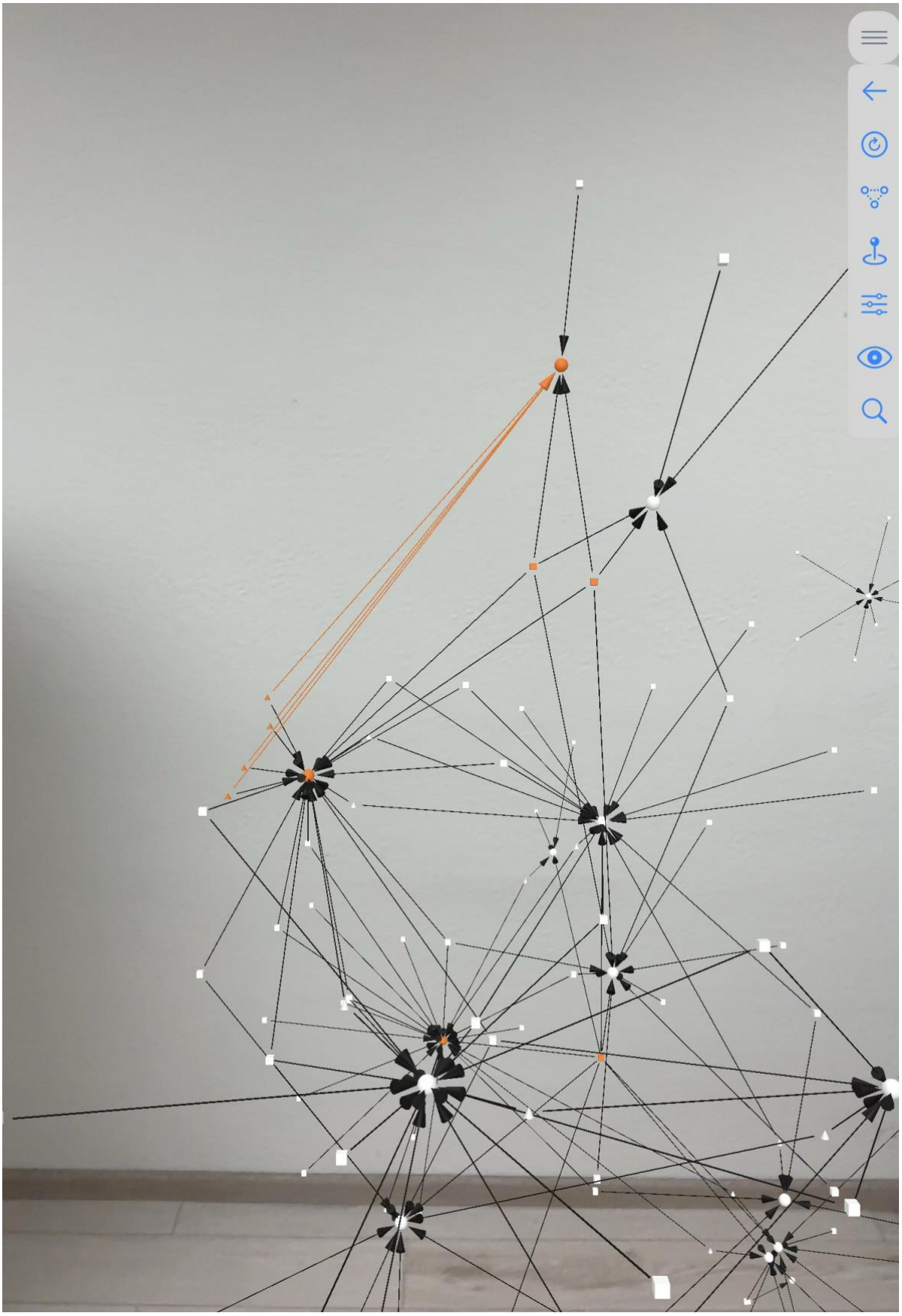
с. Режим повного контрасту



4. Режими індикації елементів архітектури
    a. Стандартний режим (без індикації)



b. Режим індикації антипатернів



ФРАНКІВ ОЛЕКСАНДР ОЛЕКСАНДРОВИЧ

Результат перевірки підпису	Підпис вірний
П.І.Б.	ФРАНКІВ ОЛЕКСАНДР ОЛЕКСАНДРОВИЧ
РНОКПП	3570502370
Організація (установа)	ФІЗИЧНА ОСОБА
Код ЄДРПОУ	
Посада	
Час підпису (підтверджено кваліфікованою позначкою часу для даних від Надавача)	17:15:57 28.06.2026
Сертифікат виданий	КНЕДП АЦСК АТ КБ "ПРИВАТБАНК"
Серійний номер	5E984D526F82F38F040000001DE77D0183908007
Тип носія особистого ключа	Захищений
Алгоритм підпису	dstu4145
Тип підпису	Кваліфікований
Формат підпису	CAdES-T
Сертифікат	Кваліфікований