УДК 004.415.5
DOI: 10.18523/2617-3808.2025.8.167-173

Франків О. О.

АВТОМАТИЗОВАНЕ ВИЯВЛЕННЯ ВАД АРХІТЕКТУРИ ПРОГРАМНОГО МОДУЛЯ З ВИКОРИСТАННЯМ ГРАФОВОЇ МОДЕЛІ ВІЗУАЛІЗАЦІЇ

У цій статті описано новий підхід до автоматизованого виявлення вад проєктування програмного модуля з використанням моделі візуалізації архітектури у формі графа із застосуванням додаткових алгоритмів аналізу. Запропонований спосіб дозволяє виявити поширені вади та інформацію про них у легкому для сприйняття поданні з використанням лише вихідного коду програми. У статті розглянуто чотири ознаки наявності помилок при проєктуванні в контексті ООП, зокрема порушеної згуртованості, зв'язності і циклічних залежностей, а також способи їх виявлення та представлення у моделі архітектури. На реальних прикладах виконано автоматизоване виявлення вад із використанням розробленого інструменту для мови Swift з підтвердженням їх наявності у процесі детального аналізу кодової бази.

Ключові слова: програмування, програмний модуль, архітектура програмного забезпечення (ПЗ), оцінювання якості ПЗ, візуалізація архітектури, автоматизований статичний аналіз, шаблони проєктування, метрики, зв'язність програмного коду, згуртованість програмного коду.

Вступ

На сьогодні важливим викликом у галузі розроблення програмного забезпечення є підтримка належної якості продукту. Попри те, що наразі доступна велика кількість інструментів автоматичного виявлення помилок у програмному коді, значна частка з них концентрується на виявленні доволі простих помилок на рівні синтаксису та іноді семантики [6; 8]. Іншою проблемою наявних інструментів є обмежені можливості представлення інформації про програму, що робить процес обробленя результатів нетривіальним.

Раніше в статтях [1; 2] було описано підхід до моделювання архітектури програми у формі графа, в якій вузли відображають класи та їхні компоненти, а ребра — зв'язки між ними, такі як належність, використання, наслідування тощо. Естетично прийнятне розміщення цієї моделі в тривимірному просторі за силовим алгоритмом і з відображенням у доповненій реальності дає можливість спростити сприйняття складних залежностей на інтуїтивному рівні на основі стереометричних метафор. У процесі дослідження було розроблено програмний комплекс A.D.A.R. — Architecture Displayer in Augmented Reality, для автоматичної генерації та відображень моделі архітектури.

Попри те, що розроблений інструмент вдало будував візуалізацію архітектури, питання виявлення помилок проєктування залишалося поза контекстом. У цій статті ми розглянемо автоматизацію виявлення вад, що стала можливою завдяки значним удосконаленням візуальної моделі, а також автоматизації аналізу аномалій з використанням вбудованих статистичних прийомів. Важливою є також можливість індикації потенційно проблемних місць у структурі програми безпосередньо в моделі.

Вади проєктування в ООП і способи їх візуалізації

Парадигма об'єктно-орієнтованого програмування не є новою. На сьогодні існує велика кількість досліджень [4; 9], у яких запропоновано та перевірено різноманітні метрики, що дозволяють із певною точністю перевірити дотримання базових принципів ООП. Використання метрик дає змогу не лише отримати конкретне значення рівня дотримання принципів, а й розробляти ефективні рішення для їх обчислення. З іншого боку, числове представлення складних концепцій не є інтуїтивним, знач-

но залежить від технологій, а ймовірний некоректний результат може ще більше ускладнити детальний аналіз [10].

Розглянемо деякі поширені помилки проєктування, що виникають унаслідок порушення принципів ООП, та можливі підходи до їх інтуїтивної візуалізації.

Концентрація логіки програми в межах одного класу є небажаною з огляду на утворення сильної зв'язності і, як наслідок, надлишкових залежностей, що своєю чергою призводить до підвищення ризику аномалій при зміні коду та крихкості логіки загалом. Такі метрики, як CBO (Coupling Between Object) — зв'язність між об'єктами і LOCM (Lack of Cohesion in Methods) — відсутність згуртованості методів, можна просто обчислити, провівши статичний аналіз вихідного коду, та виявити ознаки вад проєктування. Однак у класичному випадку вони обчислюються для цілого проєкту і радше свідчать про якість коду в цілому. Очевидно, їх можна обчислювати і для окремих класів, що значно звужує область пошуку проблемних місць, але в такому разі слабким місцем такого підходу є потреба в гнучкому визначенні порогу критичності для кожної з метрик, що не може бути універсальним, адже залежить від специфіки конкретної програми.

На противагу такому підходу розглянемо велику зв'язність і низьку згуртованість у контексті графової моделі архітектури. Вочевидь ознакою великої зв'язності в області конкретного компонента (не обов'язково класу, а й властивості чи метода) є велика кількість ребер (зв'язків), що сполучають цей компонент з іншими. Отже вузли, що мають аномально великий степінь у графовій моделі, потенційно є центрами областей великої зв'язності.

З іншого боку, низька згуртованість також є проблемою, що може свідчити про невдалу декомпозицію логіки. З урахуванням того, що граф архітектури є зваженим, а ваги відображають інтенсивність зв'язку між компонентами, аномально мала вага зв'язку структурної частини (властивості чи метода) з класом порівняно з іншими свідчить про низьку згуртованість і, як наслідок, ставить під сумнів правильність структурної належності. В екстремальних випадках порушення згуртованості призводить до таких ситуацій, як «заздрісні методи» (feature envy) [5].

Слід зауважити, що, на противагу числовим представленням метрик, у графовій моделі зв'язність і згуртованість легко відображаються відстанями між компонентами, а також кольорами для додаткової індикації (рис. 1).

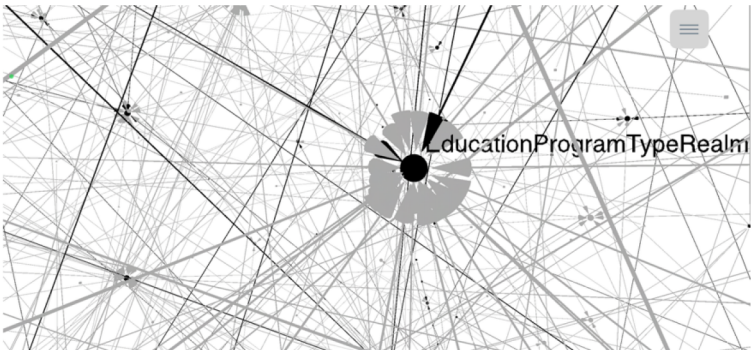

Рис. 1. Приклад вузла з великим степенем

Ще однією проблемою при проєктуванні є виникнення циклічних залежностей між компонентами. У деяких випадках такий підхід може бути виправданим, але загалом є небажаним, адже в такому разі відсутність циклу на етапі виконання часто досягається шляхом роботи в багатопотоковому середовищі, а отже є вразливим до можливого «змагального стану» (race condition).

Очевидно, що граф є вдалою структурою для пошуку циклічних залежностей, що також дозволяє використовувати ефективні підходи до пошуку циклів із погляду продуктивності. Окрім того, візуальна індикація ребер, що формують цикл, дозволяє легше сприйняти циклічну залежність незалежно від кількості ланок у ланцюзі.

Ураховуючи те, що для розміщення графу у тривимірному просторі для забезпечення високого рівня естетичності застосовується адаптація силового алгоритму, серйозні порушення в контексті естетичності подання також можуть свідчити про надмірну зв'язність компонентів. Одним із базових критеріїв естетичності, які за задумом забезпечує алгоритм, є неперетин ребер, або, менш строго, мінімізація перетинів ребер. Відповідно індикація ребер, що перетинаються чи близькі до перетину з певною похибкою, дозволяє звернути увагу на потенційно проблемну ділянку.

Тут варто зауважити, що силовий алгоритм не є стабільним, а якість відображення графа у просторі сильно залежить і від додаткових налаштувань алгоритму, зокрема критеріїв зупинки. З огляду на це порушення естетичності в частині перетину ребер є дотичною ознакою і корелює як зі структурними особливостями графа, так і з особливостями роботи алгоритму в конкретному частковому випадку.

Виявлення аномалій архітектури програми у графовій моделі

Виявлення вад проєктування за умови відсутності метаданих проєкту, таких як проєктна документація, не може передбачати перевірку шляхом встановлення відповідності еталону. Серед альтернативних варіантів можна визначити перевірку за метриками з порівнянням із загальноприйнятими рекомендованими значеннями якісних архітектур і пошук аномалій.

Сучасні програмні рішення можуть значно відрізнятись залежно від доменної області, зокрема у складності та об'ємі. Окрім того, на сьогодні широко використовують спеціалізації та видозміни ООП, як, до прикладу, ПОП (протокольно-орієнтоване програмування) чи РП (реактивне програмування). Через це еталонні значення метрик, встановлені до появи деяких сучасних технологій або ж на конкретного виду програмах, можуть виявитись нерелевантними. В такому разі користувач системи повинен самостійно налаштувати критерії, що своєю чергою негативно впливає на автоматизацію як таку, адже потребує додаткових ресурсів, зокрема для глибокого аналізу особливостей проєкту. Варто також зазначити, що точково тонко налаштовані еталони стають досить умовним критерієм якості.

Саме з огляду на ці проблеми при розробленні нашого рішення ми сконцентрувались на гнучкому виявленні аномалій як ознак імовірних порушень загальної архітектури програми. Варто зауважити, що для сценаріїв, у яких загальна якість архітектури низька в принципі, ймовірно, більш оптимальним рішенням може бути гібридний підхід із використанням узагальнених еталонних значень для мінімального порогу критерію у поєднанні з пошуком аномалій.

Для пошуку аномалій у нашому інструменті реалізовано окремий модуль AnomalyDetector. Він містить однойменний клас, екземпляри якого можуть аналізувати наявну готову графову модель і виявляти в ній ознаки потенційних порушень структури.

AnomalyDetector паралельно виконує пошук таких аномалій:

1) вузли з аномально великим степенем;
2) ребра, що позначають належність до класу, але мають аномально малу вагу;
3) цикли;
4) надмірне зближення ребер, у тому числі перетин.

На підставі цих даних програма-переглядач додає візуальну індикацію для потенційно проблемних елементів.

Вузли, що мають аномально великий степінь, очевидно, є областями великої зв'язності коду. Важливою проблемою тут є визначення, які значення степеня є аномальними. Для забезпечення гнучкості одночасно з низьким впливом на швидкодію було вирішено використати простий статистичний інструментарій. Для кожного виду вузлів у графі (класи, властивості, методи) окремо обраховується математичне сподівання, яке в цьому випадку збігається з середнім арифметичним, та дисперсія.

Поняття аномального значення вводиться класично відповідно до формули:

$$anomaly(x) = \begin{cases} true, if\ |x - \text{m}| > k * \text{s} \\ flase, otherwise \end{cases},$$

де x — значення степеню певного вузла, μ — математичне сподівання, k — пороговий коефіцієнт (за замовчуванням $k = 2$), σ — дисперсія.

Для виявлення ознак низької згуртованості між компонентами класу було значно вдосконалено частину проєкту A.D.A.R. — аналізатор. В оригінальній версії при побудові графа вага ребра, що відображає зв'язок «є методом» (класу) або «є властивістю» (класу), завжди безумовно становила 1 при області значень ваг [0;1]. Таким чином модель ніколи не відображала реальної згуртованості, а лише максимальне значення.

Для обчислення реального рівня згуртованості компонентів було взято спрощений варіант відповідних метрик, що обчислювався для кожного компонента окремо. Окрім цього, це значення обчислюється по-різному для методів і властивостей.

Слід зауважити, що поняття властивостей у сучасних мовах програмування може бути дуже відносним. До прикладу, у Swift існують обчислювані властивості (computed properties), що окрім синтаксису мало відрізняються від методів без параметрів, або ж властивості, що ініціалізуються шляхом виконання анонімної функції (closure). У цьому дослідженні ми опустили ці особливості і сконцентрувались на роботі з більш класичними поняттями методів і властивостей.

Для обчислення ваги ребра, що поєднує властивість із класом, якому вона належить, з огляду на те, що властивість не використовує метод, але метод може звертатись до неї у своєму тілі, ми використовували формулу:

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

де w (p, c) — вага ребра між вузлом властивості p та класу c; $m_{p,c}$ — кількість методів класу c, що використовують властивість p; M_c — загальна кількість методів класу c.

Для обчислення ваги ребра, що поєднує метод з класом, якому він належить, ми використовували таку формулу:

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

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

Для визначення того, чи вага ребра є аномально малою, ми використовували аналогічний підхід, як для визначення аномального значення степеня вузла.

Задача виявлення циклів у цьому контексті є тривіальною. AnomalyDetector просто долучає до звіту інформацію про знайдені цикли.

У контексті пошуку порушення критеріїв естетичності, зокрема значного наближення чи перетину ребер, важливо підкреслити, що у реальних випадках візуалізації точний перетин відрізків малоймовірний. Тому ми зробили акцент на пошуку наближень відрізків, адже перетин є частковим екстремальним випадком. Відрізки вважаються аномально наближеними, якщо найменша відстань між ними є в межах деякої невеликої відстані, що у нашому випадку становить 0,0001 умовної одиниці розміру за загального розміру куба, в який вписано граф 500 * 500 * 500 умовних одиниць.

Виявлення та локалізація проблемних частин архітектури

Із метою перевірки здатності запропонованого підходу виявляти та локалізовувати потенційно проблемні місця в архітектурі програмного модуля було проведено експерименти на реальних проєктах iOS додатків на платформі GitHub, написаних мовою Swift.

Для ілюстрації розглянемо невеликий застосунок — гру 2048 [3]. В першу чергу було проведено автоматизовану побудову архітектури програми з використанням нашого інструменту з додатковим автоматизованим аналізом аномалій. Отриману візуальну модель можна побачити на рис. 2. Чорним кольором позначено потенційно проблемні елементи, а сірим — всі інші.

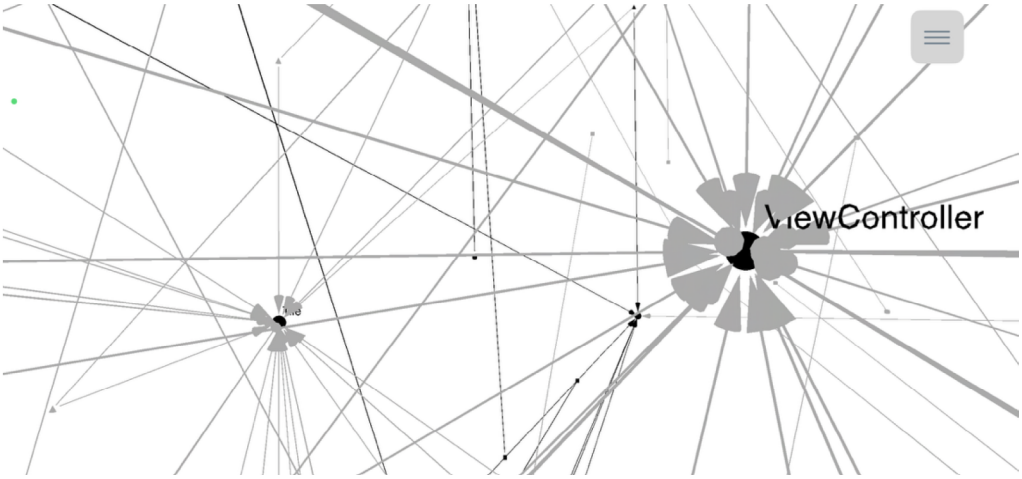

Рис. 2. Частина візуалізації проєкту 2048

Варто зауважити, що циклічних залежностей у цьому випадку не було виявлено, що зумовлено тим, що, по-перше, проєкт є досить простим, а по-друге, наявність таких залежностей є досить рідкісним явищем.

Натомість виявлено п'ять вузлів з аномально великим степенем, чотири властивості з аномально низькою згуртованістю і 10 ребер, що розміщені досить близько в умовах цієї візуалізації.

```
class ViewController: UIViewController {
      var manager: GameLogicManager!
      var score: Score!
      var highScore: HighScore!
      var renderer: GameBoardRenderer!
      var restartButton: RestartButton!
      // …
}
```

Лістинг 1. Властивості класу ViewController

Одним із вузлів із найбільшим степенем є вузол, що позначає клас ViewController. За детальнішого аналізу (лістинг 1) добре видно, що цей клас справді поєднує в собі всі частини функціоналу застосунку, що, як наслідок, робить цю частину коду крихкою.

Іншим прикладом вузла з великим степенем є вузол, що позначає метод save(dimension: Int, tiles: [Tile]). Як видно з лістингу 2, в тілі цього методу справді є ознаки порушення інкапсуляції, зокрема в частині створення екземпляру TileModel.

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

Усі чотири виявлені властивості з низьким рівнем згуртованості зі своїм класом належать до класу Tile. Цей клас містить лише декларації цих властивостей без методів. Натомість клас GameLogicManager містить метод isGameOver() -> Bool (Лістинг 3), що порушує інкапсуляцію властивостей класу Tile, працюючи безпосередньо з ними і, як наслідок, ці властивості мають вищу згуртованість із ним, аніж із власним класом.

```
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 }
            if neighbours.count != 0 {
                  return false
            }
      }
      return true
}
```

Лістинг 3. Порушення інкапсуляції в методі isGameOver() -> Bool

Що стосується виявлених наближень ребер графу, то вони за детального аналізу здебільшого є ознакою загальної великої зв'язності проєкту.

Отже, з використанням автоматизованої побудови візуалізації архітектури програми з виявленням вад проєктування можна інтуїтивно легко виявити проблемні області коду програми.

Висновки

У цій статті розглянуто вдосконалення інструменту для автоматизованої побудови візуальної графової моделі архітектури програмного модуля з акцентом на виявлення та локалізацію потенційних вад проєктування. Пошук порушення зв'язності та згуртованості між компонентами програми дозволяє скоротити процес пошуку областей, вразливих до змін, і сконцентрувати ресурс для детального аналізу відповідно.

Новий модуль AnomalyDetector, працюючи лише з моделлю архітектури, дозволяє швидко та гнучко виявляти ознаки проблемного коду завдяки використанню простого статистичного інструментарію.

На прикладі реального застосунку проілюстровано відповідність між виявленими аномаліями в моделі та реальними проблемами безпосередньо в коді, що демонструє результативність, ефективність та практичну цінність запропонованого рішення.

Список літератури

1. Франків О. О. Використання доповненої реальності для візуалізації архітектур програмних модулів [Електронний ресурс] / О. О. Франків // Наукові записки НаУКМА. — 2023. — Т. 5. — С. 26–30. — Режим доступу: https://doi.org/10.18523/2617-3808.2022.5.26-30.
2. Франків О. О. Автоматизована візуалізація компонентів архітектури програми для мови Swift [Електронний ресурс] / О. О. Франків, М. М. Глибовець // Кібернетика та системний аналіз. — 2024. — Т. 60 (6). — С. 190–198. — Режим доступу: https://doi.org/10.1007/s10559-024-00737-9.
3. Bobrov A. 2048 [Electronic resource] / A. Bobrov. — GitHub, 2020. — Mode of access: https://github.com/aibobrov/2048.
4. Chidamber S. R. A metrics suite for object oriented design / S. R. Chidamber, C. F. Kemerer // IEEE Transactions on Software Engineering. — 1994. — Vol. 20, iss. 6. — Pp. 476–493. — Mode of access: https://doi.org/10.1109/32.295895.
5. Fowler M. Refactoring: Improving the design of existing code / M. Fowler, K. Beck, J. Brant, W. Opdyke, D. Roberts. — Boston, MA: Addison Wesley Professional, 1999. — 464 p.
6. Lenarduzzi V. A Critical Comparison on Six Static Analysis Tools: Detection Agreement and Precision [Electronic resource] / V. Lenarduzzi, F. Pecorelli, N. Saarimäki, S. Lujan, F. Palomba // SSRN Electronic Journal. — 2022. — Mode of access: https://doi.org/10.2139/ssrn.4044439.
7. Li R. Understanding software architecture erosion: A systematic mapping study [Electronic resource] / R. Li, P. Liang, M. Soliman, P. Avgeriou // Journal of Software: Evolution and Process. — 2022. — Vol. 34 (3). — e2423. — Mode of access: https://doi.org/10.1002/smr.2423.
8. Schneider S. Comparison of static analysis architecture recovery tools for microservice applications [Electronic resource] / S. Schneider, A. Bakhtin, X. Li, J. Soldani, A. Brogi, T. Černý, R. Scandariato // Empirical Software Engineering. — 2025. — Vol. 30. — P. 128. — Mode of access: https://doi.org/10.1007/s10664-025-10686-2.
9. Tang M.-H. An empirical study on object-oriented metrics [Electronic resource] / M.-H. Tang, M.-H. Kao, M.-H. Chen // Proceedings of the Sixth International Software Metrics Symposium. — 1999. — Pp. 242–249. — Mode of access: https://doi.org/10.1109/metric.1999.809745.
10. Voas J. What happened to software metrics? [Electronic resource] / J. Voas, R. Kuhn // Computer. — 2017. — Vol. 50, no. 5. — Pp. 88–98. — Mode of access: https://doi.org/10.1109/MC.2017.144.

References

Bobrov, A. (2020). *2048* [Source code]. GitHub. https://github.com/aibobrov/2048.

Chidamber, S. R., & Kemerer, C. F. (1994). A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, *20* (6), 476–493. https://doi.org/10.1109/32.295895.

Fowler, M., Beck, K., Brant, J., Opdyke, W., & Roberts, D. (1999). *Refactoring: Improving the design of existing code*. Addison Wesley Professional.

Frankiv, O. O. (2023). Application of augmented reality for software module architecture visualization. *Scientific Notes of NaUKMA*, *5*, 26–30. https://doi.org/10.18523/2617-3808.2022.5.26-30 [in Ukrainian].

Frankiv, O. O., & Hlybovets, M. M. (2024). Automated visualization of software architecture components for Swift. *Cybernetics and Systems Analysis*, *60* (6), 190–198. https://doi.org/10.1007/s10559-024-00737-9 [in Ukrainian].

Lenarduzzi, V., Pecorelli, F., Saarimäki, N., Lujan, S., & Palomba, F. (2022). A critical comparison on six static analysis tools: Detection agreement and precision. *SSRN Electronic Journal*. https://doi.org/10.2139/ssrn.4044439.

Li, R., Liang, P., Soliman, M., & Avgeriou, P. (2022). Understanding software architecture erosion: A systematic mapping study. *Journal of Software: Evolution and Process*, *34* (3), e2423. https://doi.org/10.1002/smr.2423.

Schneider, S., Bakhtin, A., Li, X., Soldani, J., Brogi, A., Černý, T., & Scandariato, R. (2025). Comparison of static analysis architecture recovery tools for microservice applications. *Empirical Software Engineering*, *30*, 128. https://doi.org/10.1007/s10664-025-10686-2.

Tang, M.-H., Kao, M.-H., & Chen, M.-H. (1999). An empirical study on object-oriented metrics. In *Proceedings of the Sixth International Software Metrics Symposium* (pp. 242–249). https://doi.org/10.1109/metric.1999.809745.

Voas, J., & Kuhn, R. (2017). What happened to software metrics? *Computer*, *50* (5), 88–98. https://doi.org/10.1109/MC.2017.144.

O. Frankiv

AUTOMATED DETECTION OF SOFTWARE MODULE ARCHITECTURE FLAWS USING A GRAPH-BASED VISUALIZATION MODEL

This paper introduces a novel approach to the automated detection of software module design flaws through a graph-based architectural visualization model, enhanced with specialized analysis algorithms. The method is designed to identify common structural and logical issues in codebases, relying solely on static information derived from the source code. Unlike traditional approaches that require extensive runtime instrumentation or manual inspection, this solution produces a visual and interpretable model of the software architecture, highlighting problematic areas automatically. The visualization is not merely illustrative — it serves as an analytical layer capable of surfacing critical design issues in a comprehensible and compact form.

The proposed method extends the capabilities of architectural visualization by integrating algorithms that automatically detect signs of architectural degradation. This includes the ability to pinpoint segments of the code that demonstrate structural inefficiencies, making the analysis both accessible and actionable. As a result, developers and architects are provided with immediate feedback about design quality, which can be used to guide refactoring efforts and architectural improvements.

The paper focuses on four specific indicators of poor design in the context of object-oriented programming: weakened cohesion, excessive coupling, the presence of cyclic dependencies, and architectural violations in module boundaries. Techniques for detecting and representing these patterns in the architectural model are discussed in detail. An anomaly detection-based approach is proposed to ensure the solution remains both performant and adaptable, allowing it to be applied across a diverse set of projects varying in size, domain, and technological stack.

The approach has been validated on real-world Swift codebases using a custom-built analysis tool. The tool successfully identified structural flaws, which were confirmed through manual inspection and deeper analysis of the source code, demonstrating its effectiveness and practical utility.

Keywords: programming, software module, software architecture, software quality assessment, architecture visualization, automated static analysis, design patterns, metrics, code coupling, code cohesion.

Матеріал надійшов 07.07.2025