УДК 004.415
DOI https://doi.org/10.32782/2663-5941/2023.4/11

Куликовська Н.А.
Національний університет «Запорізька політехніка»

Руденко В.В.
Національний університет «Запорізька політехніка»

Тіменко А.В.
Національний університет «Запорізька політехніка»

Шкарупило В.В.
Національний університет біоресурсів і природокористування України

ДОСЛІДЖЕННЯ ЧАСУ ЗБИРАННЯ ДОДАТКІВ, ПОБУДОВАНИХ НА ОСНОВІ СУЧАСНИХ СТРАТЕГІЙ РОЗРОБЛЕННЯ

У статі розглянуто стратегії розроблення додатків, які охоплюють різні підходи до організації та створення програмних систем. Монолітна архітектура передбачає розроблення всієї програми як єдиного інтегрованого блоку, що забезпечує простоту та легкість тестування та обслуговування. Модульна архітектура поділяє програми на незалежні модулі з чітко визначеними інтерфейсами, забезпечуючи кращу зручність обслуговування, багаторазове використання та масштабованість. Це дозволяє паралельну роботу, покращує право власності, інкапсулює код і полегшує динамічну доставку та повторне використання коду. Однак впровадження модульності може призвести до певних недоліків, таких як збільшення складності конфігурації збірки та довший час синхронізації Gradle із занадто великою кількістю модулів. Проведено дослідження, де результати показали, що кількість модулів безпосередньо впливає на час збірки. Тестування проводилось в три етапи. Перший це оригінальний проєкт, побудований за принципами модульності. Наступним кроком була зроблена його копія та об'єднані підмодулі функцій в один модуль. Третій етап, тестувався додаток з тим самим кодом, але об'єднаний в один модуль. 59-модульний проєкт продемонстрував на 37% довший час створення порівняно з одномодульним проєктом. Поетапні збірки в багатомодульному проєкті також займали значно більше часу. Результати експерименту показали, що конфігурація з одним модулем продемонструвала найменший час збирання, тоді як підхід із багатьма модулями призвів до довшої тривалості компіляції. Загалом експеримент дав цінну інформацію про компроміси між монолітною та модульною архітектурами, підкреслюючи важливість вибору найбільш прийнятного підходу на основі вимог до проєкту та бажаних результатів.

Ключові слова: *модульна архітектура, час збирання, додатки, моделювання, архітектура.*

Постановка проблеми. Стратегії розробки додатків включають різні підходи до організації та створення програмних систем. Монолітна архітектура означає розробку всієї програми як єдиного інтегрованого блоку, що дозволяє забезпечити простоту та легкість тестування та обслуговування. Проте, цьому підходу може бракувати масштабованості для великих проєктів. З іншого боку, модульна архітектура розділяє програми на незалежні модулі з чітко визначеними інтерфейсами, надаючи кращу обслуговуваність, багаторазове використання та масштабованість, проте це може призвести до довшого часу розробки, оскільки модулі потрібно компілювати окремо. Крім того, архітектура мікросервісів переносить модульність на екстремальний рівень, розбиваючи додатки на невеликі незалежні сервіси, що взаємодіють через API, що надає високу масштабованість і ізоляцію помилок, але це призводить до підвищеної складності і довшого часу розробки через потребу окремої компіляції та розгортання кількох сервісів. Модульність протягом тривалого часу була фундаментальною концепцією розроблення програмного забезпечення, і вона продовжує широко застосовуватися в сучасній практиці розроблення програмного забезпечення.

Аналіз останніх досліджень і публікацій. Вибір стратегії активно досліджується та обговорюється в галузі розроблення програмного забезпечення в наш час [1, с. 78]. Багато науковців розглядають переваги, недоліки та компроміси архітектурних підходів. Мартін Фаулер, відомий

розробник програмного забезпечення та автор, багато писав про архітектури програмного забезпечення, включаючи дискусію про монолітні та мікросервіси [2, с. 42]. Крім того, Роберт К. Мартін, консультант із програмного забезпечення та автор, зробив внесок у цю тему [3, с. 10]. Його книга «Чиста архітектура: Посібник для майстра зі структури та дизайну програмного забезпечення» заглиблюється в важливість модульності в розробленні програмного забезпечення. Інші дослідники, такі як Еойн Вудс і Майкл Фізерс, також поділилися думками щодо архітектурного вибору між монолітними та модульними додатками. Їх роботи, зокрема «Архітектура систем програмного забезпечення: робота із зацікавленими сторонами з використанням точок зору та перспектив» та «Ефективна робота з застарілим кодом» відповідно, надають цінні погляди на цю тему [4, с. 394; 5, с. 340].

Більше того, модульний підхід до розроблення архітектурної складової сучасних розподілених вебсистем, зокрема систем, побудованих на основі вебсервісів, є дієвим механізмом одержання реконфігурованих та масштабованих рішень, що відповідають ad-hoc-природі актуальних сценаріїв запитів до таких систем [6, с. 5]. У свою чергу, при збиранні названих систем постають до вирішення задачі контролю не лише функціональних характеристик, зокрема несуперечності програмно-алгоритмічної складової [7, с. 148], а й нефункціональних характеристик – супутніх часових витрат [8, с. 91]. Такий підхід відповідає, зокрема, концепції модульного тестування, згідно якого висновок стосовно успішності проходження досліджуваним модулем складових заданих тестових наборів робиться на підставі контролю показників і функціональних, і нефункціональних характеристик [9, с. 16].

У сукупності зазначені вище наукові публікації підкреслюють важливість вибору правильного архітектурного підходу на основі розміру, складності та вимог до масштабованості проєкту. Дослідження підкреслює переваги модульних додатків з точки зору зручності обслуговування, багаторазового використання, або визнають простоту та легкість монолітних додатків. Таким чином розробникам важливо ретельно обмірковувати вибір стратегії розроблення програмного забезпечення.

Метою статті є дослідження монолітної та модульної архітектури побудови застосунків. Аналіз переваг та недоліків кожної з них. Тестування розробленого проєкту на основі даних архітектур та дослідження часу збирання.

Виклад основного матеріалу. Багато мов програмування та фреймворків підтримують модульність, що дозволяє розробникам створювати модульні системи [10, с. 210]. Крім того, шаблони архітектури програмного забезпечення, такі як мікросервіси та компонентні архітектури, значною мірою покладаються на принципи модульності [11, с. 11]. Однак ступінь впровадження модульності може відрізнятися в різних проєктах і організаціях. Деякі програмні системи все ще страждають від монолітних конструкцій, де модульність мінімальна, що призводить до проблем з обслуговуванням і масштабованістю. З іншого боку, більш сучасні проєкти, як правило, охоплюють модульні практики та шаблони

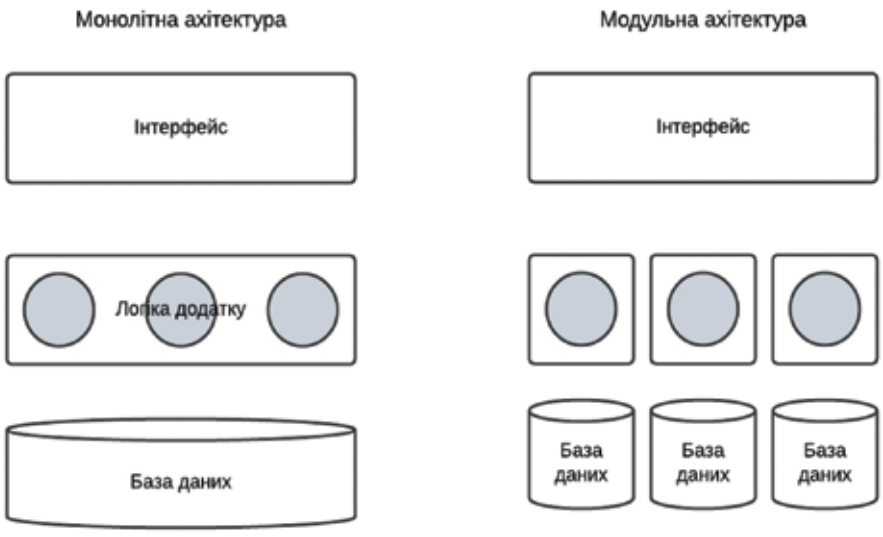

Рис. 1. Сучасні стратегії розроблення програмного забезпечення

проєктування, виграючи від покращеної організації коду та легшої співпраці (рис. 1).

Модульність пропонує кілька переваг у розробленні програмного забезпечення:

– масштабованість. Модульні проєкти дотримуються принципу поділу проблем, що дозволяє локалізувати зміни та зменшити каскадні ефекти. Це забезпечує кращу масштабованість у міру зростання та розвитку проєкту;

– паралельну роботу. Завдяки чітким межам модулів розробники у великих командах можуть працювати паралельно більш ефективно, зменшуючи конфлікти контролю версій і підвищуючи продуктивність;

– право власності. Модулі можуть мати виділених власників, відповідальних за технічне обслуговування, виправлення помилок і перевірку коду, сприяючи підзвітності та спрощеним процесам розроблення;

– інкапсуляція. Ізольований код у модулях легше читати, розуміти, тестувати та підтримувати, що сприяє загальній якості коду.

– динамічна доставка. Модульність підтримує функції динамічної доставки, що дозволяє доставляти певні функції програми умовно або завантажувати на вимогу;

– багаторазове використання. Належна модульність дає змогу спільно використовувати код і полегшує створення кількох програм на різних платформах, використовуючи одну основу.

Хоча модульність має переваги, вона також має певні недоліки:

– занадто багато модулів. Надмірна кількість модулів може призвести до збільшення складності конфігурації збірки та довшого часу синхронізації Gradle. Підтримка численних модулів тягне за собою постійні витрати та може вплинути на конфігурацію проєкту;

– недостатньо модулів. Занадто мало модулів або великі тісно з'єднані модулі можуть нагадувати монолітну архітектуру, зводячи нанівець деякі переваги модульності. Завеликі модулі, які не мають чітко визначених цілей, слід розглядати для розділення;

– занадто складно. Рішення про застосування модульності залежить від розміру та складності кодової бази. У деяких випадках, коли очікується, що проєкт не перевищить певний поріг, переваги масштабованості та часу створення можуть бути незастосовні, що робить модульність менш вигідною. Важливо ретельно оцінити потреби проєкту, перш ніж використовувати повністю модульний підхід.

Під час експерименту було взято вихідний код одного багатомодульного реального проєкту. На основі цього додатку було сформовано три конфігурації: багато модулів, менше модулів і лише один модуль. Мета цього експерименту полягала в тому, щоб проаналізувати, як кількість модулів вплинула на час проєкту. Порівнюючи продуктивність і характеристики кожної конфігурації було визначити оптимальний баланс між модульністю та монолітністю додатку.

Початковий проєкт мав 59 модулів, які складалися з 864 файлів Kotlin і 1 файлу Java. Крім того, він включав 65 файлів активів, 1 файл прототипу та 161 файл ресурсів (рис. 2).

Тестування проводилось в три етапи. Перший це оригінальний проєкт, побудований за принци-

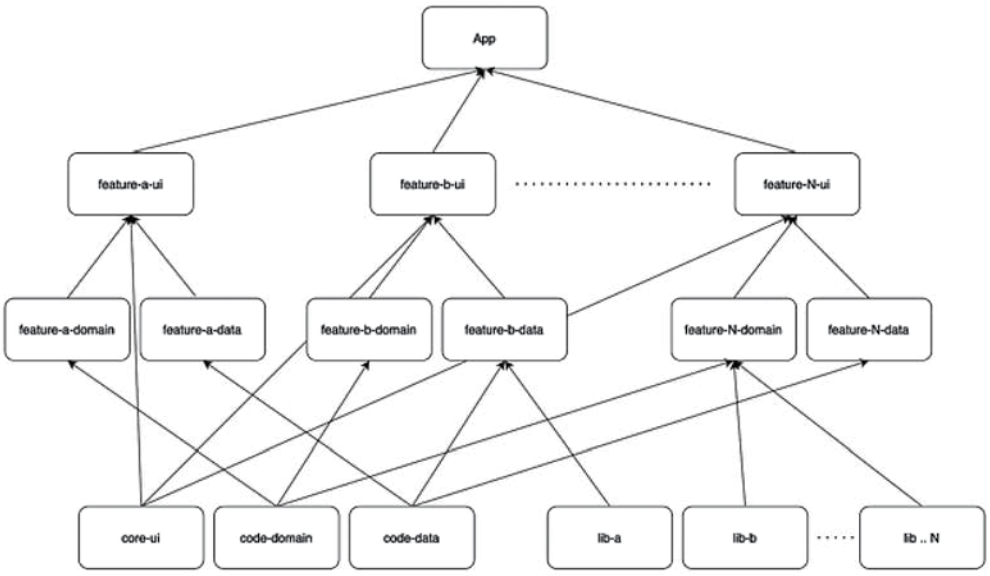

Рис. 2. Проєкт із 59 модулів

пами модульності. Наступним кроком була зроблена його копія та об'єднані підмодулі функцій в один модуль. Наприклад, функція А має модулі даних, домену та інтерфейсу користувача. Було зроблено переміщення даних та класів домену в модуль інтерфейсу користувача, таким чином, маючи 3 модулі для 1 функції, отримали лише один. За таким алгоритмом вдалось зменшити кількість модулів до 33 (рис. 3).

Третій етап, тестувався додаток з тим самим кодом, але об'єднаний в один модуль.

Перелічимо операції, які вимірювались при збиранні проєкту:

– ./gradlew clean – важливе завдання при перемиканні гілки проєкту;

– ./gradlew assembleDevDebug – операція виконується кожного разу, коли натискається «Створити проєкт» або «Запустити програму» в Android Studio;

– ./gradlew assembleDevDebug –offline – використання офлайн-режиму;

– ./gradlew assembleDevRelease – перевірка якості збірки;

– ./gradlew assembleDevDebug – але без очищення проєкту, щоб перевірити, як кількість модулів впливає на інкрементні збірки.

Завдання assembleDevDebug в монолітному проєкті показує найкращий час (табл. 1). Середній час на 5 запусках – 75,4 с. Проєкт з 33 модулями працював на 19,6 секунди довше, тобто на 26% довше. А 59-модульний проєкт працював на 37% довше. Як бачите, монолітний проєкт показує найкращий час збирання.

На тривалість завдання assembleDevRelease найбільше впливає кількість модулів (табл. 1). Чим більше модулів, тим довше він працюватиме. Складання збірки випуску для проєкту з 59 модулів займає на 50% більше часу, порівняно з проєктом з одним модулем.

Завдання assembleDevDebug, без очищення після кожного виконання, показує, що поетапне збирання значно довше в багатомодульному проєкті (табл. 1).

Результати експерименту показали, що конфігурація з одним модулем продемонструвала найменший час збирання, тоді як підхід із багатьма модулями призвів до довшої тривалості компіляції. Загалом експеримент дав цінну інформацію про компроміси між монолітною та модульною архітектурами, підкреслюючи важливість вибору найбільш прийнятного підходу на основі вимог до проєкту та бажаних результатів.

Висновки. У даній роботі ми розглянули дві основні архітектурні стратегії розробки програмних додатків: монолітна та модульна. Монолітна архітектура передбачає розроблення програми як єдиного інтегрованого блоку, що спрощує тестування та обслуговування. З іншого боку, модульна архітектура дозволяє розділити програму на незалежні модулі з чітко визначеними інтерфейсами, що сприяє кращому обслуговуванню, багаторазовому використанню та масштабованості. Проведене дослідження показало, що впровадження модульності може призвести до деяких недоліків, зокрема збільшення складності конфігурації збірки та збільшення часу синхронізації Gradle при наявності великої кількості модулів. Результати показали, що кількість модулів безпосередньо впливає на час збірки. У випадку експерименту з тестуванням різних підходів, проєкт з монолітною архітектурою демонстрував найкращі результати,

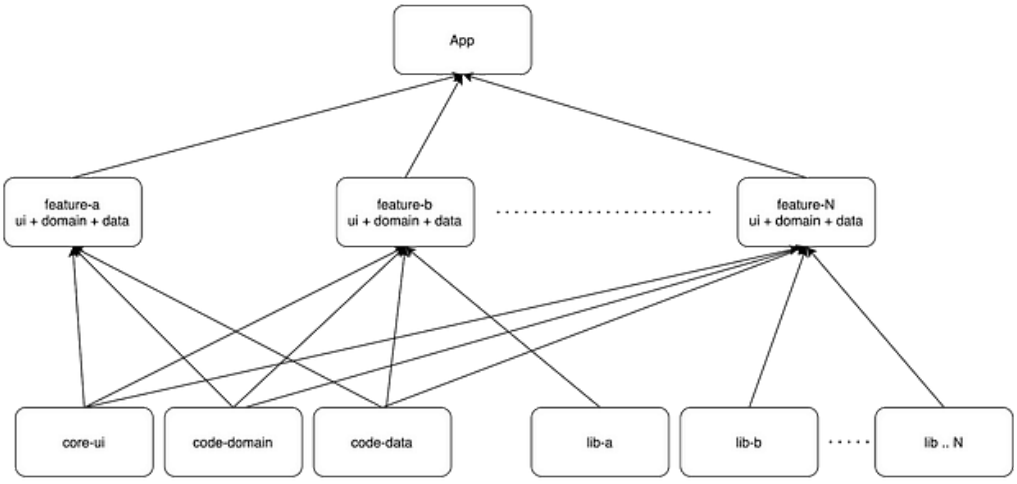

Рис. 3. Проєкт із 33 модулів

Таблиця 1

Тестування операцій

Показники	Операція assembleDevDebug			Операція assembleDevRelease			Операція assembleDevDebug без очищення		
	1 модуль	33-модуля	59-модулів	1 модуль	33-модуля	59-модулів	1 модуль	33-модуля	59-модулів
1 запуск (с)	80	98	102	185	261	331	2	7	13
2 запуск (с)	74	95	104	174	229	249	2	7	11
3 запуск (с)	74	94	102	194	238	260	2	7	10
4 запуск (с)	75	94	106	216	268	283	2	7	11
5 запуск (с)	74	94	104	225	336	368	2	7	10
Середнє значення	75,4	95	103,6	198,8	266,4	298,2	2	7	11
Дельта (с)	0	19,6	28,2	0	67,6	99,4	0	5	9
Дельта (%)	0	26	37	0	34	50	0	250	450

з найменшим часом збирання порівняно з багатомодульним проєктом. Таким чином, дана робота надає цінну інформацію про компроміси між монолітною та модульною архітектурами та підкреслює важливість вибору оптимального підходу залежно від вимог до проекту та бажаних результатів.

Список літератури:

1. ROS-based architecture for fast digital twin development of smart manufacturing robotized systems / C. Saavedra Sueldo et al. Annals of operations research. 2022. URL: https://doi.org/10.1007/s10479-022-04759-4 (date of access: 15.07.2023).

2. Pallewatta S., Kostakos V., Buyya R. Placement of Microservices-based IoT Applications in Fog Computing: A Taxonomy and Future Directions. ACM Computing Surveys. 2023. URL: https://doi.org/10.1145/3592598 (date of access: 21.07.2023).

3. Multi-service prevention programs for pregnant and parenting women with substance use and multiple vulnerabilities: Program structure and clients' perspectives on wraparound programming / D. Rutman et al. BMC Pregnancy and Childbirth. 2020. Vol. 20, no. 1. URL: https://doi.org/10.1186/s12884-020-03109-1 (date of access: 22.07.2023).

4. Jaskot K., Przyłucki S. Analysis of selected features of application based on monolithic and microservice architecture. Journal of Computer Sciences Institute. 2022. Vol. 25. P. 393–400. URL: https://doi.org/10.35784/jcsi.3061 (date of access: 22.07.2023).

5. Nugraha A. R., Talita A. S. Mobile Application Architecture Restructuring with Microservice Approach. Journal of Information Technology and Computer Science. 2021. Vol. 5, no. 3. URL: https://doi.org/10.25126/jitecs.202053239 (date of access: 25.07.2023).

6. Shkarupylo V. A technique of DEVS-driven validation. 2016 13th International Conference on Modern Problems of Radio Engineering. Telecommunications and Computer Science (TCSET), Lviv, Ukraine, 23–26 February 2016. 2016. URL: https://doi.org/10.1109/tcset.2016.7452097 (date of access: 25.07.2023).

7. Шкарупило В.В., Чемерис О.А., Душеба В.В. Оцінювання просторової складності задачі формальної верифікації, вирішуваної методом перевірки на моделі. Вчені записки Таврійського національного університету імені В.І. Вернадського, серія «Технічні науки», 2020. Том 31 (70), № 5. С. 147–151. URL: https://doi.org/10.32838/2663-5941/2020.5/24 (дата звернення 25.07.2023).

8. Шкарупило В.В., Душеба В.В., Скрупський С.Ю., Блінов І.В. Стратифікована модель подання нефункціональних характеристик системи критичного призначення при проєктуванні. Електронне моделювання, 2022. Т. 44, № 2. С. 90–106. ISSN 0204–3572. URL: https://doi.org/10.15407/emodel.44.02.090 (дата звернення 25.07.2023).

9. JUnit, OO design patterns. CSE 331, Spring 2011, Section 4 cheat sheet. URL: https://courses.cs.washington.edu/courses/cse331/11sp/sections/section4-cheat-sheet.pdf (date of access: 25.07.2023).

10. Methodology for Performance Analysis of Distributed Knowledge-Based Systems / N. Kulykovska et al. Computer Modeling and Intelligent Systems. 2021. Vol. 2864. P. 206–215. URL: https://doi.org/10.32782/cmis/2864-18 (date of access: 28.07.2023).

11. Ganguli A., Slyne F., Ruffini M. Real-time, low latency virtual DBA hypervisor for SLA-compliant multi-service operations over shared Passive Optical Networks. Optical Fiber Communication Conference, San Diego California. Washington, D.C., 2023. URL: https://doi.org/10.1364/ofc.2023.tu3f.2 (date of access: 28.07.2023).

Kulykovska N.A., Rudenko V.V., Timenko A.V., Shkarupylo V.V. INVESTIGATING BUILD TIMES FOR APPLICATIONS BUILT ON MODERN DEVELOPMENT STRATEGIES

The article is devoted to application development strategies that cover various approaches to the organization and creation of software systems. Monolithic architecture involves the development of the entire program as a single integrated unit, which ensures simplicity and ease of testing and maintenance. A modular architecture divides applications into independent modules with well-defined interfaces, providing better maintainability, reusability, and scalability. It enables parallel operation, improves ownership, encapsulates code, and facilitates dynamic code delivery and reuse. However, implementing modularity can lead to some disadvantages, such as increasing the complexity of the build configuration and longer synchronization time of Gradle with too many modules. A study was conducted where the results showed that the number of modules directly affects the build time. Testing was conducted in three stages. The first is an original project built on the principles of modularity. The next step was to make a copy of it and combine the function submodules into one module. The third stage, the application with the same code, but combined into one module, was tested. A 59-module project demonstrated a 37% longer build time compared to a single-module project. Staged builds in a multi-module project also took significantly longer. Experimental results showed that the single-module configuration demonstrated the lowest build time, while the multi-module approach resulted in longer compilation times. Overall, the experiment provided valuable insight into the trade-offs between monolithic and modular architectures, highlighting the importance of choosing the most appropriate approach based on project requirements and desired outcomes.

Key words: *modular architecture, build time, applications, modeling, architecture.*