# РОЗДІЛ ІІ. ІНЖЕНЕРІЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

УДК 004.02
DOI https://doi.org/10.26661/2786-6254-2025-2-05

## ОГЛЯД ПІДХОДІВ ДО ОРГАНІЗАЦІЇ БІЗНЕС-ЛОГІКИ В ПОБУДОВІ МІКРОСЕРВІСНИХ СИСТЕМ

**Бейрак Д. Я.**
*аспірант кафедри інженерії програмного забезпечення*
*Державний університет «Житомирська політехніка»*
*вул. Чуднівська, 103, Житомир, Україна*
*orcid.org/0009-0006-5089-3603*
*dm.beirak@gmail.com*

**Вакалюк Т. А.**
*доктор педагогічних наук, професор,*
*завідувач кафедри інженерії програмного забезпечення*
*Державний університет «Житомирська політехніка»*
*вул. Чуднівська, 103, Житомир, Україна*
*orcid.org/0000-0001-6825-4697*
*tetianavakaliuk@gmail.com*

***Ключові слова:*** *мікросервісна архітектура, бізнес-логіка, предметно орієнтоване проєктування, DDD, event sourcing, CQRS, патерни.*

Натепер питання побудови архітектури мікросервісів стає все більш актуальним, оскільки даний тип архітектури дозволяє проєктувати системи з низькою зв'язністю, які мають низку переваг перед монолітами: можливість горизонтального масштабування, краще розділення системи на складові частини (сервіси), кожен окремий з яких простіше розвивати та підтримувати, можливість більш ефективного використання ресурсів та інші. Зазначені вище та низка інших причин приводять до зростання популярності такого типу архітектури в індустрії, що позначається на виборі архітекторів та інженерів програмного забезпечення стосовно впровадження мікросервісної архітектури як у побудові нових систем, так і як вектора розвитку успадкованих монолітних систем, які все частіше переписуються на мікросервіси. Проблематика, що стосується питань проєктування мікросервісних систем, має велику кількість різноманітних аспектів, одним із них є вибір організації бізнес-логіки разом із низкою супутніх патернів, технологій та інструментів. Вплив такого вибору неможливо переоцінити: бізнес-логіка є реалізацією предметної області, у якій працює бізнес, тому вибір відповідних патернів і підходів до її організації має прямий вплив не тільки на якість реалізації системи, а й на вартість її розширення та підтримки в майбутньому. У статті розглядаються методи, принципи й інструменти, призначені для організації бізнес-логіки в мікросервісних системах, розглядаються патерни, призначені для використання в умовах простих і складних доменів, підходи до організації роботи з командами та запитами в системах, що використовують події. Значна увага приділяється парадигмі предметно орієнтованого проєктування, як найперспективнішій у застосуванні під час розроблення систем зі складною предметною областю.

# APPROACHES TO BUSINESS LOGIC IMPLEMENTATION IN MICROSERVICE SYSTEMS IN THE SCIENTIFIC LITERATURE

**Beirak D. Ya.**
*Postgraduate Student at the Department of Software Engineering*
*Zhytomyr Polytechnic State University*
*Chudnivska str., 103, Zhytomyr, Ukraine*
*orcid.org/0009-0006-5089-3603*
*dm.beirak@gmail.com*

**Vakaliuk T. A.**
*Doctor of Pedagogical Sciences, Professor,*
*Head of the Department of Software Engineering*
*Zhytomyr Polytechnic State University*
*Chudnivska str., 103, Zhytomyr, Ukraine*
*orcid.org/0000-0001-6825-4697*
*tetianavakaliuk@gmail.com*

***Key words:*** *microservice architecture, business logic, domain-driven design, DDD, event sourcing, CQRS, patterns.*

Currently, the issue of building a microservice architecture is becoming increasingly relevant, as this type of architecture allows you to design systems with low coupling, which have a number of advantages over monoliths: the possibility of horizontal scaling, better division of the system into components (services), each of which is easier to develop and maintain, the possibility of more efficient use of resources, etc. The above-mentioned and a number of other reasons lead to the growing popularity of this type of architecture in the industry, which affects the choice of software architects and engineers regarding the implementation of microservice architecture both when designing new systems and as a vector of development for legacy monolithic systems, which are increasingly being migrated to microservices. The issues related to the design of microservice systems include a large number of different aspects, and the choice of business logic implementation is one of them, along with a number of related patterns, technologies, and tools. The impact of such a choice cannot be overestimated: business logic is the implementation of the business domain in code, and therefore the choice of appropriate patterns and approaches has a direct impact not only on the quality of the system implementation, but also on the cost of its expansion and support in the future. This paper examines methods, principles, and tools designed to organize business logic in microservice systems, considers patterns designed to be used in simple and complex domains, approaches to organizing work with commands and requests in event-driven systems. Considerable attention is paid to the paradigm of domain-driven design, as the most promising in application when developing systems within complex domains.

**Вступ.** Натепер усе більше нових проєктів створюється на основі мікросервісної архітектури, а також усе більше успадкованих систем переводяться на даний тип архітектури. Велике значення має питання організації бізнес-логіки, тобто вибору патернів і підходів описання предметної області у вихідному коді. Від даного вибору залежить не тільки якість архітектури системи, а також вартість її розширення та підтримки в майбутньому, якість комунікації розробників і менеджменту. Дані чинники роблять актуальними дослідження наявних патернів і підходів реалізації бізнес-логіки в контексті побудови мікросервісної архітектури. **Огляд літератури.** Проблеми, пов’язані з організацією бізнес-логіки в системах з мікросервісною архітектурою, у науковій літературі та публікаціях розглядали різні автори та вчені. Зокрема, загальні проблеми та патерни описували Кріс Річардсон (Chris Richardson), Сем Ньюман (Sam Newman) і Мартін Фаулер (Martin Fowler). Питання організації бізнес-логіки з використанням предметно орієнтованого проєкту-

вання присутні у працях Еріка Еванса (Eric Evans) та Влада Хононова (Vlad Khononov). Застосування даного підходу у функціональній парадигмі висвітлювали Скотт Влашин (Scott Wlaschin) і Уберто Барбіні (Uberto Barbini). Також питання, пов'язані з організацією бізнес-логіки, розглядали Штефан Капферер (Stefan Kapferer), Олаф Циммерманн (Olaf Zimmermann), Владімір Хоріков (Vladimir Khorikov), Джерфі Палермо (Jeffrey Palermo), Дерек Колі (Derek Colley), Клейр Стейнер (Clare Stanier), Штефан Лізер (Stefan Lieser), Стенлі Ліма (Stanley Lima), Жаймі Корія (Jaime Correia), Філіпі Араужю (Filipe Araujo), Жорже Кардозо (Jorge Cardoso), Ішмаель Мота (Ismael Mota), Міхіел Оверейм (Michiel Overeem), Мартен Шпур (Marten Spoor), Слінге Янсен (Slinger Jansen), Шак Брінкемпер (Sjaak Brinkkemper), Озан Оскан (Ozan Özkan), Ондер Бабур (Önder Babur), Марк ван ден Бранд (Mark van den Brand), Метаяс Верейс (Mathias Verraes), Ребека Верфс-Брок (Rebecca Wirfs-Brock), Хуля Вулар (Hulya Vural), Мурат Коюнджу (Murat Koyuncu) та інші.

**Метою статті** є детальний аналіз підходів і патернів організації бізнес-логіки в контексті побудови мікросервісної архітектури.

**Результати.** Розроблення бізнес-логіки в системі з мікросервісною архітектурою ускладнене тим, що вона є розподіленою між багатьма сервісами. Кріс Річардсон (Chris Richardson) зазначає необхідність подолання двох викликів, пов'язаних з такого типу розробленням: неможливість прямого виклику коду, що міститься в іншому сервісі, та транзакційні обмеження, які накладає розподілена природа мікросервісної архітектури [1]. Перший виклик також може впливати на необхідність вирішення проблеми створення розподіленої моделі й ускладнювати процес тестування [2]. Другий виклик обмежує можливість застосування ACID-транзакцій лише в рамках кожного окремо взятого сервісу, унеможливлює такий тип транзакційності між ними [1].

Дані обмеження впливають на підхід до організації бізнес-логіки та вибору архітектурного стилю сервісу. Традиційна для монолітних систем шарова архітектура (layered architecture) не враховує взаємодії системи з кількома сховищами даних та іншими системами, а також за її використання часто виникає залежність шару бізнес-логіки від шару доступу до даних (хоча зворотне спрямування даної залежності можливе) [1, 3]. Можливою альтернативою шаровій архітектурі є гексагональна (також відома під назвою «порти й адаптери»), а також її варіації: onion-архітектура та «чиста» архітектура [1, 4]. У своїй основі кожна з них має принцип інверсії залежностей, що насамперед диктує залежність коду доступу до даних від ядра сервісу – бізнес-логіки, а також передбачає винесення іншого інфраструктурного коду та коду інтерфейсу користувача на периферію застосунку [4; 5].

Сервіс, побудований із застосуванням гексагональної архітектури, має вхідні і вихідні порти, а також вхідні і вихідні адаптери. Порти використовуються для взаємодії бізнес-логіки із зовнішніми частинами застосунку: наприклад, вхідний порт може являти собою публічні методи для доступу до функціональності бізнес-логіки, що в об'єктно орієнтованих мовах програмування зазвичай реалізується через відповідний інтерфейс, а вихідний порт може являти собою інтерфейс репозиторія – колекцію операцій доступу до даних. Адаптери розташовуються на периферії сервісу та використовуються для взаємодії із зовнішніми сервісами та частинами системи. Так, вхідні адаптери приймають запити й обробляють їх шляхом виклику відповідних вхідних портів. Прикладом вхідного адаптера може слугувати контролер з кінцевими точками REST API (REST API endpoints) або клієнт брокера повідомлень (message broker client). Вихідні адаптери викликаються вихідним портом (або реалізовують його, якщо він є інтерфейсом), та викликають зовнішній сервіс або іншу частину системи. Прикладами вихідних портів можуть бути класи, що реалізують операції доступу до даних, публікують події або проксують зовнішні виклики [1].

У класичній роботі Мартіна Фаулера [6] розглянуто три можливі патерни організації бізнес-логіки: сценарій транзакції (transaction script), доменну модель (domain model) та табличний модуль (table module).

У мікросервісній архітектурі, у його початковому вигляді, застосовується лише сценарій транзакції. Суть цього патерну полягає в організації бізнес-логіки у формі колекції процедур (методів, функцій), кожна з яких містить код для обробки однієї бізнес-транзакції (запиту, операції). Стан системи водночас зберігається окремо. Даний підхід є великою мірою процедурним, тому підходить лише для проєктування сервісів із простою бізнес-логікою [1, 6].

Коли ж бізнес-логіка все ще залишається простою, але дані системи є досить складними, щоб ускладнити використання сценарію транзакцій, можна застосувати патерн активного запису (active record). Він дозволяє абстрагувати рядок у таблиці, інкапсулює до нього доступ за допомогою CRUD-операцій (операції створення, читання, оновлення, видалення), тим самим спрощує відображення об'єкта в оперативній пам'яті на схему бази даних [6, 7]. В об'єкти, що реалізовані як активний запис, також можна помістити бізнес-логіку, проте її відсутність перетворює їх на анемічні доменні моделі (anemic domain model),

що є антипатерном для систем із крупними доменами [7, 8].

За необхідності імплементації більш складної бізнес-логіки, у монолітних системах, написаних у стилі об'єктно орієнтованого програмування (далі – ООП), можливе застосування доменної моделі [6], проте в разі мікросервісної архітектури більше підходить предметно орієнтоване проєктування (domain-driven design, DDD) [1], використання якого також можливе у програмуванні у функціональній та інших парадигмах [9, 10].

Складники DDD зосереджені навколо побудови моделей предметної області. Патерни, що застосовуються під час побудови застосунків з використанням DDD, поділяються на стратегічні і тактичні [7].

До стратегічних відносять такі патерни, як піддомен (subdomain), єдина мова (ubiquitous language), обмежений контекст (bounded context) і контекстна карта (context map). Розглянемо їх детальніше.

Піддомен – складник предметної області, бізнес-домену. Виділяють три типи піддоменів: ключові піддомени (core subdomains), які відрізняють бізнес компанії від інших і надають їй конкурентну перевагу; загальні піддомени (generic subdomains) – складні в імплементації, але однакові в різних бізнесах активності, які не надають компанії конкурентної переваги, але є невід'ємним складником продукту (наприклад, системи аутентифікації та авторизації, прийому платежів); підтримувальні піддомени (supporting subdomains) – нескладні в імплементації піддомени, які не надають бізнесу конкурентну перевагу (рішення, що базуються на ETL- (*extract, transform, load* – витяг, перетворення, завантаження) або CRUD-операціях) [7].

Єдина мова – корпоративна термінологія, що описує всі поняття бізнес-домену, якою послуговуються всі працівники та стейкхолдери. Має бути точною та консистентною. У рамках обмеженого контексту кожне поняття повинно мати лише одне значення; синонімічні терміни не допускаються. Єдина мова має бути сформована в термінах бізнесу та відповідати ментальній моделі доменних експертів. Наповнення єдиної мови має покривати лише ті аспекти предметної області, що використовуються в комунікації, розробленні та трансляції ідей; аспекти домену, що не використовуються в бізнес-контексті, не включаються в єдину мову, хоча можуть її розширювати, за потреби, у майбутньому. Важливим нюансом є підтримка єдиної мови всіма учасниками. Інструментами підтримки можливе використання глосарія, тестів мовою Gherkin, спеціалізованих аналізаторів коду тощо [7]. У більш широкому розумінні, можлива інтеграція DDD з підходом керованої поведінкової розробки (behavior-driven development, BDD). Даний підхід передбачає використання специфікації системи, що находить від стейкхолдерів у форматі історій користувачів (user stories), як єдиної мови [11].

Обмежений контекст – окремо виділений контекст доменної моделі на основі тлумачень понять єдиної мови, що конфліктують. Консистентність єдиної мови та кількість виділених обмежених контекстів мають протилежну залежність: що консистентніше єдина мова, то менше потрібно обмежених контекстів. Обмежені контексти застосовуються коли необхідно мати кілька моделей того самого поняття або концепції. Інакше кажучи, обмежений контекст визначає межі застосування єдиної мови, що описує частину предметної області в застосуванні до вирішення конкретних проблем.

На відміну від піддоменів, які виділяються із предметної області на основі стратегії ведення бізнесу, обмежені контексти будуються як архітектурні рішення на основі вибору границь моделей. У застосуванні до мікросервісної архітектури кожен обмежений контекст має бути реалізований як окремий сервіс; отже, обмежений контекст може містити один або більше піддоменів. З позиції ownership'у (відповідальності команди за розроблення частини функціоналу системи) один обмежений контекст має бути призначений одній і тільки одній команді, хоча команда може володіти кількома обмеженими контекстами одночасно [7].

У проєктуванні обмежених контекстів важливо також ураховувати частоту зміни пов'язаних одна з одною концепцій, що може слугувати приводом включення їх у єдиний обмежений контекст у деяких складних випадках моделювання предметної області – наприклад, якщо один піддомен потребує використання декількох обмежених контекстів [12].

Знаходження оптимальних точок розділення предметної області на обмежені контексти може бути також виконане за допомогою розрахунків таких показників, як аферентна зв'язність (afferent coupling), еферентна зв'язність (efferent coupling) і пов'язність (cohesion), значення яких сприяють знаходженню варіантів з найнижчою зв'язністю та найбільш високою пов'язністю [13].

У проєктуванні в об'єктно орієнтованій парадигмі використання доменних подій усередині обмеженого контексту є стандартною практикою, проте у функціональній такий підхід не вітається через те, що він створює додаткові приховані залежності. Натомість внутрішнє виконання послідовних дій (так званий workflow) будується за допомогою композиції. Особливістю зовнішнього workflow є те, що він приймає на вхід команду, а його виходом є множина подій, що є точками відо-

кремлення такого workflow від зовнішнього світу. Відповідно, вхідні команди мають валідуватися на предмет задоволення внутрішніх умов обмеженого контексту, а вихідні події перевірятися стосовно того, чи не витікають із даного обмеженого контексту приватні дані, створюючи зайву зв'язність та проблеми з безпекою [9].

Контекстна карта – діаграма, яка зображує обмежені контексти системи в розрізі їхньої інтеграції одне з одним. Типи інтеграції описуються відповідними патернами та поділяються на три групи, залежно від способу взаємодії між командами розробників, як-от: кооперація, споживач – постачальник, окремі шляхи. До патернів кооперації належать партнерство (partnership) і спільне ядро (shared kernel). У разі партнерства координація змін є двосторонньою: одна команда повідомляє іншу про зміни у своєму API, до яких друга адаптується. Застосування даного патерну передбачає високий рівень комунікації між командами й може не підходити в разі географічно розподілених команд.

Патерн спільне ядро передбачає винесення спільних для різних обмежених контекстів моделей у єдине загальне місце – вихідні файли коду або бібліотеку. Даний патерн вносить сильну залежність між обмеженими контекстами, до яких його застосовано, і має використовуватися лише тоді, коли ціна дублювання коду є вищою за ціну координації змін у спільній кодовій базі. Тип інтеграції споживач – постачальник порушує баланс між командами в сенсі диктування однією з них формату контракту та передбачає використання трьох патернів, як-от: конформіст (conformist), запобіжний шар (anti-corruption layer) і сервіс з відкритим протоколом (open-host service). Патерн конформіст передбачає безумовне прийняття споживачем моделі обмеженого контексту постачальника. Його застосування виникає тоді, коли контракт постачальника є індустріальним стандартом або просто добре підходить для потреб споживача. Патерн запобіжний шар дозволяє споживачу транслювати моделі постачальника у більш зручний формат контракту. Його використання виправдане, коли безкомпромісне прийняття сторонньої моделі небажане або неможливе: обмежений контекст споживача може бути ключовим піддоменом, який не варто піддавати зовнішньому впливу, модель постачальника може бути незручною або часто змінюватися тощо.

Патерн сервіс з відкритим протоколом, навпаки, передбачає пристосування постачальника до потреб споживачів: для кожного з них створюється окремий публічний протокол, орієнтований на потреби споживача. Застосування даного патерну дозволяє постачальнику гнучкіше розвивати реалізацію власного функціоналу без зайвого впливу на обмежені контексти споживачів, а також дає можливість впроваджувати більш ліберальне версіонування власного API.

Останній тип інтеграції є водночас однойменним патерном – окремі шляхи (separate ways). Він застосовується тоді, коли жодна колаборація між командами неможлива або неефективна, тому простіше дублювати функціональність у різних обмежених контекстах. Застосування цього патерну необхідно уникати за інтеграції ключових піддоменів [7]. Оскільки контекстна карта відображає не лише суто технічну інформацію про архітектуру застосунку, а й зв'язки між командами та способи їхньої взаємодії, іноді можна вдаватися до зворотного маневру Конвея (inverse Conway maneuver), щоб забезпечити узгодженість архітектури зі структурою компанії [9]. Контекстна карта також відіграє значну роль у проєктуванні та прототипуванні системи. Цей процес може бути виконаний за допомогою спеціалізованих інструментів, як-от Context Mapper. Даний інструмент дозволяє використовувати спеціальну DSL-мову Context Mapper DSL (CML), за допомогою якої описуються піддомени, обмежені контексти, складається контекстна карта, а також описуються історії користувачів і сценарії використання. Фактично, CML дозволяє описати всю стратегічну частину DDD [14].

До тактичних патернів відносять сутність (entity), об'єкт-значення (value object), агрегат (aggregate), фабрику (factory), репозиторій (repository) і сервіс (service). Розглянемо їх детальніше.

Сутність (Entity) – об'єкт предметної області, який має властиву йому неперервну в часі індивідуальність існування (ідентичність), яка зберігається в нього навіть якщо його неідентифікуючі атрибути зміняться [9, 15]. Таким чином, дві сутності, які мають однакові атрибути, але різні ідентифікатори, є різними сутностями [1]. Ідентифікуючий атрибут однозначним чином відрізняє одну сутність від іншої [15]. Прикладами сутностей є користувач, замовлення, продукт, рахунок-фактура, банківська транзакція [9, 15]. Сутності використовуються лише як складники агрегатів.

Об'єкт-значення (Value object) – об'єкт предметної області, який не має власної ідентичності (індивідуального існування), а є набором окремих полів і повністю ними визначається. Однакових значень цих атрибутів у двох об'єктах-значеннях досить для того, щоб уважати їх повністю взаємозамінними [1, 15]. З погляду функціональної парадигми об'єкти-значення мають бути незмінними (immutable) [9]. Атрибути, що утворюють об'єкт-значення, мають бути єдиним концептуальним цілим. Прикладами об'єктів-значень є колір, літера, назва, адреса, локація, дата, об'єкт грошей [9, 15].

Агрегат (Aggregate) – сукупність взаємопов'язаних об'єктів, що створюють єдине ціле. Складається з кореневого об'єкта (сутності) і одного чи багатьох інших об'єктів (сутностей і об'єктів-значень) [1]. Кожен агрегат має межу, яка визначає об'єкти, з яких складається агрегат.

Кореневий об'єкт – єдиний складник агрегату, що містить унікальний ідентифікатор, на який і тільки на який можуть посилатися всі зовнішні об'єкти (агрегати). Також він відповідає за перевірку та дотримання інваріантів (бізнес-правил сутностей предметної області) та внутрішньої консистентності агрегату. Об'єкти, розташовані всередині агрегату, можуть посилатися один на одного без обмежень. Посилання на некореневі сутності можуть бути надані іншим об'єктам лише на час виконання якоїсь однієї операції; некореневі об'єкти можуть мати лише локальну ідентичність, тобто бути унікальними лише в межах агрегату. Кореневий об'єкт також може передавати зовнішнім клієнтам копії об'єктів-значень, які вже не матимуть зв'язку з агрегатом. Лише кореневі об'єкти можуть бути отримані через запити в базу даних – інші об'єкти, що утворюють агрегат, мають отримуватися лише через зв'язок з кореневим об'єктом [1, 15].

З погляду транзакційності даних агрегати мають використовуватися як атомарні одиниці. Одна транзакція може створювати або оновлювати лише один агрегат, що дає змогу гарантувати, що транзакція виконуватиметься в межах одного сервісу [1, 9]. Агрегати використовуються для моделювання бізнес-об'єктів предметної області [1]. Зазвичай їх варто робити якомога меншими та включати в них лише ті об'єкти, які мають бути у строгій консистентності [7]. Прикладом агрегату може бути замовлення, що містить позиції, які самі собою є окремими об'єктами, але водночас замовлення розглядається саме як єдине ціле [16].

Фабрика (Factory) – об'єкт або метод, що реалізує процес створення або відновлення складних об'єктів і агрегатів. Для проєктування фабрик можуть використовуватися як дизайн-патерни «банди чотирьох» (фабричний метод (factory method), абстрактна фабрика (abstract factory), будівельник (builder)), так і аналогічні засоби в інших парадигмах програмування [1, 15].

Репозиторій (Repository) – об'єкт, що реалізує механізм доступу до об'єктів, що зберігаються в базі даних [1].

Сервіс (Service) – об'єкт, що не має стану та реалізує логіку, яку не можна віднести ані до сутності, ані до об'єкта-значення. Дозволяє також координувати роботу декількох агрегатів [1, 7].

Незважаючи на те, що DDD дозволяє спроєктувати систему максимально близькою до природи предметної області, цей підхід також не позбавлений деяких проблемних аспектів. Один із них – так звана трилема вибору між чистотою доменної моделі (код бізнес-логіки не повинен мати залежностей поза власним процесом виконання та, відповідно, мати посилання лише на примітивні типи й інші доменні об'єкти), її повнотою (код, що стосується дотримання бізнес-логіки, повинен бути у відповідному шарі системи та не бути фрагментованим) та продуктивністю. Ця трилема виникає в ситуації, коли частина бізнес-логіки не може водночас бути складником агрегату та не порушувати його чистоти (наприклад, через необхідність запиту в репозиторій). Перенесення цієї частини коду в інший шар (наприклад, у контролер) призводить до фрагментованості, а вичитування всього масиву даних і передача його в агрегат з метою виконання фільтрації в ньому (замість виконання цієї роботи запитом до бази даних) призводять до зниження продуктивності. У такому разі рекомендується компромісне рішення – перенесення позапроцесового коду в контролер, що хоч і призведе до фрагментації бізнес-логіки, але буде найменшим компромісом порівняно з іншими варіантами [17].

Наступний виклик, який може виникнути під час застосування практик DDD, полягає в тому, що цей підхід має високу складність для розробників, не знайомих з його принципами й особливостями. Це спричиняє додаткові матеріальні та когнітивні витрати, особливо за переходу на дану парадигму [18].

У мікросервісній архітектурі, зокрема за використання DDD, широко застосовується патерн доменна подія (domain event). Доменна подія – це об'єкт, що продукується агрегатом, коли з ним відбувається щось важливе: створення, зміна стану тощо, і який може прийняти й обробити будь-яка зацікавлена сторона. В об'єктно орієнтованому програмуванні доменні події являють собою класи, а у функціональній парадигмі для цього можуть використовуватися, наприклад, визначення типів і моноїди [1, 9, 19]. Назви доменних подій мають бути створені за допомогою дієслів минулого часу (наприклад, "OrderCreated" – «замовлення створено», "OrderCancelled" – «замовлення скасовано» тощо) [1, 9]. У доменних подіях зазвичай містяться метадані, як-от унікальний ідентифікатор, часова відмітка (timestamp), ідентифікатор користувача, який вніс зміни, та інші [1].

Оскільки часто обробнику події необхідні додаткові дані, що відображають суть змін стану системи, а не лише повідомляють про факт таких змін, для запобігання додатковому запиту з його боку використовується так зване збагачення подій (event enrichment). Наприклад, подія "OrderCreated" може вже містити деталі щойно створеного замовлення, отже, спожива-

чам даної події не потрібно буде додатково запитувати цю інформацію під час її оброблення. Такий підхід може дещо знизити підтримуваність (maintainability) завдяки тому, що потенційно класи (або інші структури даних), якими описуються події, потребуватимуть змін кожного разу, коли змінюватимуться вимоги споживачів. Іншим недоліком може бути спроба задовільнити потреби багатьох різних споживачів події. Проте це досить рідкі сценарії, які не мають критичного характеру [1].

Одним із підходів визначення доменних подій є event storming. Він передбачає комунікацію з доменними експертами, яка складається із трьох етапів, як-от: мозковий штурм подій – доменні експерти пропонують можливі доменні події системи; визначення тригерів подій – доменні експерти мають визначити тригери, які продукують події (дії користувачів, зовнішня система, інша доменна подія, плин часу); ідентифікація агрегатів – доменні експерти повинні визначити які агрегати споживають які команди та продукують відповідні події [1].

Незважаючи на те, що успішне виконання команд (як-от дії користувачів), фактично, створює події, ці концепції необхідно чітко розрізняти. Команди – це повідомлення, які надходять зовні, уособлюють запит на зміну стану системи, їх виконання може бути як успішним, так і невдалим; звідси випливає імперативна форма їхніх назв (наприклад, “CreateOrder” – «створити замовлення»). Події ж уособлюють зміну стану, яка щойно відбулася, тому не можуть бути виконані невдало; звідси, відповідно, і форма їхніх назв у минулому часі (“OrderCreated” – «замовлення створено») [19].

Доменні події публікуються агрегатами, проте небажано, щоб вони напряму публікували повідомлення у брокер повідомлень через те, що для них неможливе впровадження залежностей (dependency injection), реалізація даної можливості призведе до змішування бізнес-логіки й інфраструктурних питань. Кращим підходом буде використання сервісів, які можуть застосовувати впровадження залежностей (отже, і отримати через даний механізм посилання на API відправки повідомлень) та, відповідно, взяти на себе відповідальність за відправку подій. Таким чином, агрегати генерують події кожного разу, коли змінюється їхній стан, та повертають їх сервісам. Наприклад, сервіс може зробити запит у сховище даних через репозиторій, що поверне екземпляр агрегату, на якому сервісом буде викликано метод, що приймає параметри, що змінюють його стан, та який поверне сервісу список подій, які він опублікує через брокер повідомлень. Іншим можливим підходом може бути акумулювання агрегатом необхідних для відправки подій у публічному полі, яке потім вичитується сервісом для публікації подій. Недоліком першого варіанту є необхідність повертання порожніх подій у разі, коли методи нічого не повертають, а недоліками другого є необхідність застосування механізмів інкапсуляції коду, що забезпечує сервісу доступ до списку подій і повторюватиметься, а також ускладнення доступу до даного поля об’єктам, які не є кореневими [1].

Традиційний спосіб організації персистентності даних системи відбувається за допомогою збереження їх у базу даних або нереляційне сховище. Такий підхід має низку недоліків. По-перше, використання реляційної бази даних підходить лише для обмеженого набору даних, які за своєю природою можуть бути представлені у формі таблиць. Оскільки здебільшого дані предметних областей потребують додаткових перетворень для збереження в реляційній базі даних, виникає так званий об’єктно-реляційний розрив (object-relational impedance mismatch), а використання ORM-фреймворків (ORM – object-relational mapping, об’єктно-реляційне відображення) не є, загалом, ефективним і зручним [1, 20]. По-друге, традиційний підхід не дозволяє нативно зберігати історію змін, що може бути корисним для аудиту системи або відновлення її попереднього стану. По-третє, для традиційного підходу збереження даних, публікація подій не є частиною бізнес-логіки й має бути реалізовано окремо, що підвищує ризик виникнення проблем із синхронізацією поточного стану після оновлень [1].

Проте реалізація персистентності можлива не лише через безпосереднє збереження стану у сховище даних. Альтернативою цьому підходу є патерн event sourcing, який передбачає збереження стану агрегату як послідовності подій, де кожна така подія являє собою черговий випадок зміни стану. Тоді відтворення стану агрегату відбувається через повторюване послідовне застосування змін, привнесених кожною подією [1]. Event sourcing також підходить для застосування у функціональній парадигмі, бо передбачає не зміну даних, а лише збереження незмінних записів змін цих даних, на основі яких можна реконструювати поточний стан системи [21].

Події, що змінюють стан агрегатів, зберігаються у сховищі подій (event store). Кожна подія містить ідентифікатор і тип події, ідентифікатор і тип сутності (або агрегату), до якої вона має стосунок, та корисне навантаження, як дані зміни стану, передбачені самою подією. Стан агрегату змінюється як результат наступного процесу: надходить команда, яка містить запит на виконання визначеної дії, яка валідується агрегатом, який, якщо не виникає помилок і виключень, генерує список

подій зміни стану, після чого вони зберігаються у сховищі подій, а відповідні обробники агрегатів, яким адресовані ці події, приймають їх і застосовують. Важливо, що ці обробники не мають закінчувати своє виконання невдачею, тому що перехід системи в новий стан уже відбувся, а також мають бути ідемпотентними – тобто повинні правильно обробляти ситуацію, коли подія, що вже була оброблена, надійте повторно. У разі, коли надходить декілька подій, що мають оновити той самий агрегат, необхідно застосовувати заходи, що забезпечують транзакційність такого оновлення (наприклад, оптимістичне блокування (optimistic locking)). Такі запобіжні механізми (як-от видавець опитувань (polling publisher) і відстеження транзакційного журналу (transaction log tailing)) мають використовуватися також під час публікування подій, бо це має бути атомарною операцією. Оскільки деякі агрегати можуть мати велику кількість оновлень та, відповідно, велику кількість подій, з яких необхідно їх конструювати, для оптимізації об'єму сховища даних і продуктивності застосовують знімки (snapshot) n-ої кількості попередніх подій, «згортаючи» їх у єдину подію, що відразу містить дані поточних змін [1].

Важливим питанням у застосуванні патерну event sourcing є еволюція подій, бо не завжди цей процес може бути зворотним. Наприклад, розширення моделі через додавання нового поля не перешкоджає застосуванню попередніх знімків стану системи, але звуження моделі через перейменування, видалення або зміну типу поля, як і зміна назви агрегату або пов'язаної з ним події, уже є несумісним із попередньою версією агрегату. Способом вирішення цієї проблеми є застосування спеціального компонента, що оновлює події до нової версії в момент завантаження зі сховища. Такий компонент має назву upcaster [1].

Ще одним питанням, яке постає під час проєктування системи з event sourcing, є потенційна необхідність виправлення помилок у попередніх подіях або їх перестановка, у разі чого поточний стан був би іншим. Ключ до вирішення цієї проблеми дає патерн ретроактивна подія (retroactive event) – тобто подія, яка б вплинула на поточний стан системи, якби була оброблена у визначеній часовій точці в минулому. Використання цього поняття дозволяє осмислювати процес виправлення помилок, як наступні три паралельні моделі (parallel model): некоректна реальність (incorrect reality) – поточний стан, що не враховує ретроактивну подію; коректна гілка (corrected reality) – стан системи, якого вона набула б, якби відбулася обробка ретроактивної події; коректна реальність (corrected reality) – фінальний стан системи, у якому вона має опинитися. Здебільшого коректна реальність збігатиметься з коректною гілкою, проте необов'язково. Суть методу зводиться до ідентифікації моменту необхідності застосування ретроактивної події – точки розгалуження (branch point), – повернення стану системи в дану точку та виправлення помилки [22, 23]. Даний підхід отримав розвиток з позиції питання спостережуваності (observability) системи. Хоча event sourcing і дозволяє відстежувати всі бізнес-події, у журналі подій (event log) часто не вистачає необхідної інформації для відстеження причинно-наслідкових зв'язків і взаємодій бізнес-подій у системі, що ускладнює процес налагодження. Способом вирішення даної проблеми є використання трасування (tracing), що надає спосіб відновлення причинно-наслідкового потоку виконання для конкретних запитів разом із метаданими, пов'язаними із внутрішнім станом системи, як-от змінні та часові відмітки. Отже, у розробників з'являється додатковий інструмент пошуку всіх, пов'язаних з окремою проблемою, подій за допомогою ідентифікатора трасування [24].

До переваг застосування event sourcing належать надійна публікація подій, збереження історії змін агрегатів, практично повне уникнення об'єктно-реляційного розриву через те, що зберігаються окремі події, а не агрегований стан [1], а також той факт, що моделювання домену з використанням event sourcing наближає вихідний код програми до того, що відбувається в реальному світі, на відміну від спроб абстрагувати модель [19]. До недоліків відносять складність у реалізації, труднощі з еволюціюванням подій і видаленням даних, а також можливе ускладнення процесу читання даних, коли для задоволення запиту необхідно спершу вичитати та застосувати всі пов'язані з даним агрегатом події, лише після чого можна буде виконати передбачувану запитом фільтрацію та отримати шуканий результат. Системи, що містять такі запити, повинні використовувати інший патерн, який оптимізує процес читання: CQRS (command and query responsibility segregation, розділення відповідальності командних запитів) [1].

Основна ідея патерну CQRS полягає в розділенні сховищ запису та читання. Фактично, таке розділення відбувається на рівні команд, що змінюють агрегат (тобто виконують операції створення, зміни та видалення), та запитів, що вичитують його поточний стан. Частина, що відповідає за роботу з командами, також може застосовувати операцію читання в разі, коли вона не є ресурсозатратною. Натомість усі складні та нетривіальні запити спрямовуються до частини, що відповідає за читання даних, у яку зміни пропагуються зі сховища для запису. Сховище для читання може містити спеціально підготовлені подання для складних запитів, але оптимізації можуть бути

застосовані до обох типів сховищ залежно від потреб домену, найбільш явною з яких є використання різних типів сховищ для запису та читання, замість компромісного рішення, яке виконує обидві операції повільніше, ніж спеціалізовані. Використання CQRS не лише дозволяє оптимізувати роботу з командами та запитами, а й робить можливим використання патерну event sourcing та дозволяє покращити розділення обов'язків, що є одним із принципів SOLID. Окрім того, комбінація патернів event sourcing і CQRS дозволяє отримати відносно невисоку складність розроблення та підтримки складних і великих систем. Недоліком використання CQRS є неминуче ускладнення архітектури та наявність лагу реплікації – дані пропагуються зі сховища запису у сховище читання не миттєво [1, 25].

Коли домен не диктує складні умови й патерни event sourcing та CQRS можуть занадто ускладнити систему, але питання збору єдиної відповіді кінцевому клієнту на основі даних, що належать кільком мікросервісам, залишається, варто застосовувати патерн об'єднання API (API composition). Застосування даного патерну передбачає використання нового компонента, що має назву "API composer", який виконує запити в необхідні сервіси та будує остаточне подання для клієнта.

Виконувати роль API composer'а може клієнтський застосунок, API gateway, окремий спеціалізований сервіс. Перший варіант може виявитися неоптимальним, тому що між клієнтським кодом і базою даних є фаєрвол і повільніша мережа, через яку неефективно передавати зайві дані. Другий варіант буде найбільш ефективний тоді, коли споживачами API composer'а є зовнішні клієнти, як-от веб- або мобільний застосунок. Третій варіант має сенс у разі, коли в ролі споживачів виступають внутрішні сервіси на бекенді, або логіка об'єднання даних, які необхідно віддавати зовнішньому клієнту, надмірно складна для того, щоб бути реалізованою в рамках API gateway.

Незважаючи на те, що об'єднання API є простішим патерном у реалізації, ніж CQRS, він також має низку недоліків. Серед них підвищені накладні витрати на додаткові запити даних, ризик зниження доступності системи через появу нового компонента (можливим способом боротьби із цим є кешування або повернення неповних даних у разі недоступності API composer'а), а також виникнення неконсистентних даних тоді, коли дані сервісів, з яких формується остаточна відповідь, ще не узгоджені одне з одним [1].

**Висновки.** Проведений аналіз підходів до організації бізнес-логіки в побудові мікросервісних систем демонструє, що натепер у контексті побудови мікросервісів за умов складної предметної області перспективною є парадигма Domain-Driven Design (DDD). Цей підхід забезпечує високу підтримуваність коду та точну відповідність моделі бізнес-вимогам, особливо в довгостроковому розвитку системи.

Для систем з відносно простою бізнес-логікою допустиме застосування таких патернів, як Transaction Script або Active Record, які є простішими в реалізації та потребують менше зусиль на впровадження. Однак рішення на основі даних патернів показують обмежені можливості за зростання об'єму кодової бази проєкту та підвищення складності бізнес-логіки системи.

Отже, застосування того чи того підходу безпосередньо залежить від складності та динаміки предметної галузі. Для систем з високою волатильністю та складною бізнес-логікою доцільно застосовувати парадигму DDD, незважаючи на її складність та високий поріг входу. Для систем малої та середньої складності краще підходять прості патерни, що забезпечують швидку реалізацію за обмежених ресурсів.

Перспективні напрями подальших досліджень такі: оптимізація DDD для типових сценаріїв, що дозволить зменшити витрати на його впровадження; розроблення і апробація гібридних підходів, що поєднують переваги парадигми DDD і простіших патернів; формалізація метрик оцінювання архітектурних рішень у контексті бізнес-логіки мікросервісів.

## ЛІТЕРАТУРА


1. Richardson C. Microservices patterns: with examples in Java. Shelter Island, New York : Manning Publications, 2019.
2. Newman S. Building microservices: designing fine-grained systems, Second Edition. Beijing : O'Reilly Media, 2021.
3. Deursen S. van, Seemann M. Dependency injection: principles, practices, patterns. Shelter Island : Manning, 2019.
4. Lieser S. Clean Architecture vs. Onion Architecture vs. Hexagonal Architecture, CCD Akademie. URL: https://ccd-akademie.de/en/clean-architecture-vs-onion-architecture-vs-hexagonale-architektur/
5. Palermo J. The Onion Architecture : part 1. Programming with Palermo. URL: https://jeffreypalermo.com/2008/07/the-onion-architecture-part-1/
6. Fowler M. Patterns of enterprise application architecture, Nineteenth printing. in The Addison-Wesley Signature Series. Boston ; San Francisco ; New York ; Toronto ; Montreal ; London ; Munich ; Paris ; Madrid ; Capetown : Addison-Wesley, 2013.

7. Khononov V. Learning domain-driven design: aligning software architecture and business strategy. Beijing ; Boston ; Farnham ; Sebastopol ; Tokyo : O'Reilly, 2022.
8. Mota I. Anaemic Domain Model vs. Rich Domain Model. *Ensono. Insights + News.* URL: https://www.ensono.com/insights-and-news/expert-opinions/anaemic-domain-model-vs-rich-domain-model/
9. Wlaschin S. Domain modeling made functional: tackle software complexity with domain-driven design and F♯, Version: P1.0. Raleigh, North Carolina : The Pragmatic Bookshelf, 2018.
10. Fowler M. Domain Driven Design. URL: https://martinfowler.com/bliki/DomainDrivenDesign.html
11. Esther D. Behavior-Driven Development for Domain-Driven Design in Modern Software. URL: https://www.researchgate.net/publication/386136826_Behavior-Driven_Development_for_Domain-Driven_Design_in_Modern_Software
12. Verraes M., Wirfs-Brock R. Splitting a Domain Across Multiple Bounded Contexts. URL: https://verraes.net/2021/06/split-domain-across-bounded-contexts/
13. Vural H., Koyuncu M. Does Domain-Driven Design Lead to Finding the Optimal Modularity of a Microservice? *IEEE Access.* 2021. Vol. 9. P. 32721–32733. DOI: 10.1109/ACCESS.2021.3060895
14. Kapferer S., Zimmermann O. Domain-Driven Architecture Modeling and Rapid Prototyping with Context Mapper. Model-Driven Engineering and Software Development / S. Hammoudi, L. F. Pires, B. Selić (Eds.). *Communications in Computer and Information Science.* Vol. 1361. Cham : Springer International Publishing, 2021. P. 250–272. DOI: 10.1007/978-3-030-67445-8_11
15. Evans E. Domain-driven design: tackling complexity in the heart of software. Boston : Addison-Wesley, 2004.
16. Fowler M. DDD Aggregate. URL: https://martinfowler.com/bliki/DDD_Aggregate.html
17. Khorikov V. Domain model purity vs. domain model completeness (DDD Trilemma). *Enterprise Craftsmanship.* URL: https://enterprisecraftsmanship.com/posts/domain-model-purity-completeness
18. Özkan O., Babur Ö., Van Den Brand M. Refactoring with domain-driven design in an industrial context: An action research report. *Empir Software Eng.* Jul. 2023. Vol. 28. № 4. P. 94. DOI: 10.1007/s10664-023-10310-1
19. Barbini U. From objects to functions: build your software faster and safer with functional programming and Kotlin. Dallas, Texas : The Pragmatic Bookshelf, 2023.
20. Colley D., Stanier C., Asaduzzaman M. Investigating the Effects of Object-Relational Impedance Mismatch on the Efficiency of Object-Relational Mapping Frameworks. *Journal of Database Management.* Oct. 2020. Vol. 31. № 4. P. 1–23. DOI: 10.4018/JDM.2020100101
21. Khorikov V. OOP, FP, and object-relational impedance mismatch. *Enterprise Craftsmanship.* URL: https://enterprisecraftsmanship.com/posts/oop-fp-and-object-relational-impedance-mismatch/
22. Fowler M. Retroactive Event. URL: https://martinfowler.com/eaaDev/RetroactiveEvent.html
23. Fowler M. Parallel Model. URL: https://martinfowler.com/eaaDev/ParallelModel.html
24. Lima S., Correia J., Araujo F., Cardoso J. Improving observability in Event Sourcing systems. *Journal of Systems and Software*. Nov. 2021. Vol. 181. P. 111015. DOI: 10.1016/j.jss.2021.111015
25. Overeem M., Spoor M., Jansen S., Brinkkemper S. An empirical characterization of event sourced systems and their schema evolution – Lessons from industry. *Journal of Systems and Software*. Aug. 2021. Vol. 178. P. 110970. DOI: 10.1016/j.jss.2021.110970

## REFERENCES

1. Richardson, C. (2019). *Microservices patterns: With examples in Java*. Manning Publications.
2. Newman, S. (2021). *Building microservices: Designing fine-grained systems* (Second Edition). O'Reilly Media.
3. Deursen, S. van, & Seemann, M. (2019). *Dependency injection: Principles, practices, patterns*. Manning.
4. Lieser, S. (2024, February 9). Clean Architecture vs. Onion Architecture vs. Hexagonal Architecture. *CCD Akademie*. https://ccd-akademie.de/en/clean-architecture-vs-onion-architecture-vs-hexagonale-architektur/
5. Palermo, J. (2008, July 29). The Onion Architecture: Part 1. *Programming with Palermo*. https://jeffreypalermo.com/2008/07/the-onion-architecture-part-1/
6. Fowler, M. (2013). *Patterns of enterprise application architecture* (Nineteenth printing). Addison-Wesley.
7. Khononov, V. (2022). *Learning domain-driven design: Aligning software architecture and business strategy*. O'Reilly.
8. Mota, I. (2023, May 24). Anaemic Domain Model vs. Rich Domain Model. *Ensono. Insights + News*. https://www.ensono.com/insights-and-news/expert-opinions/anaemic-domain-model-vs-rich-domain-model/

9. Wlaschin, S. (2018). *Domain modeling made functional: Tackle software complexity with domain-driven design and F♯* (Version: P1.0). The Pragmatic Bookshelf.
10. Fowler, M. (2020, April 22). *Domain Driven Design*. https://martinfowler.com/bliki/DomainDrivenDesign.html
11. Esther, D. (2023, February). *Behavior-Driven Development for Domain-Driven Design in Modern Software*. https://www.researchgate.net/publication/386136826_Behavior-Driven_Development_for_Domain-Driven_Design_in_Modern_Software
12. Verraes, M., & Wirfs-Brock, R. (2021, June 14). *Splitting a Domain Across Multiple Bounded Contexts*. https://verraes.net/2021/06/split-domain-across-bounded-contexts/
13. Vural, H., & Koyuncu, M. (2021). Does Domain-Driven Design Lead to Finding the Optimal Modularity of a Microservice? *IEEE Access*, *9*, 32721–32733. https://doi.org/10.1109/ACCESS.2021.3060895
14. Kapferer, S., & Zimmermann, O. (2021). Domain-Driven Architecture Modeling and Rapid Prototyping with Context Mapper. In S. Hammoudi, L. F. Pires, & B. Selić (Eds.), *Model-Driven Engineering and Software Development* (Vol. 1361, pp. 250–272). Springer International Publishing. https://doi.org/10.1007/978-3-030-67445-8_11
15. Evans, E. (2004). *Domain-driven design: Tackling complexity in the heart of software*. Addison-Wesley.
16. Fowler, M. (2013, April 23). *DDD Aggregate*. https://martinfowler.com/bliki/DDD_Aggregate.html
17. Khorikov, V. (2020, August 4). Domain model purity vs. Domain model completeness (DDD Trilemma). *Enterprise Craftsmanship*. https://enterprisecraftsmanship.com/posts/domain-model-purity-completeness
18. Özkan, O., Babur, Ö., & Van Den Brand, M. (2023). Refactoring with domain-driven design in an industrial context: An action research report. *Empirical Software Engineering*, *28*(4), 94. https://doi.org/10.1007/s10664-023-10310-1
19. Barbini, U. (2023). *From objects to functions: Build your software faster and safer with functional programming and Kotlin*. The Pragmatic Bookshelf.
20. Colley, D., Stanier, C., & Asaduzzaman, M. (2020). Investigating the Effects of Object-Relational Impedance Mismatch on the Efficiency of Object-Relational Mapping Frameworks: *Journal of Database Management*, *31*(4), 1–23. https://doi.org/10.4018/JDM.2020100101
21. Khorikov, V. (2016, November 3). OOP, FP, and object-relational impedance mismatch. *Enterprise Craftsmanship*. https://enterprisecraftsmanship.com/posts/oop-fp-and-object-relational-impedance-mismatch/
22. Fowler, M. (2005, December 12). *Retroactive Event*. https://martinfowler.com/eaaDev/RetroactiveEvent.html
23. Fowler, M. (2005, December 12). *Parallel Model*. https://martinfowler.com/eaaDev/ParallelModel.html
24. Lima, S., Correia, J., Araujo, F., & Cardoso, J. (2021). Improving observability in Event Sourcing systems. *Journal of Systems and Software*, *181*, 111015. https://doi.org/10.1016/j.jss.2021.111015
25. Overeem, M., Spoor, M., Jansen, S., & Brinkkemper, S. (2021). An empirical characterization of event sourced systems and their schema evolution—Lessons from industry. *Journal of Systems and Software*, *178*, 110970. https://doi.org/10.1016/j.jss.2021.110970

*Дата першого надходження рукопису до видання: 24.08.2025*
*Дата прийнятого до друку рукопису після рецензування: 19.09.2025*
*Дата публікації: 31.12.2025*