DOI: https://doi.org/10.26642/ten-2026-1(97)-234-240
УДК 004.4

І.Д. Лімінович, аспірант
Д.Д. Плечистий, к.т.н., доц.
Т.М. Локтікова, ст. викладач
Н.О. Кушнір, ст. викладач
Державний університет «Житомирська політехніка»

Огляд сучасних підходів до декомпозиції прикладних програмних систем: від моноліту до модульного моноліту та мікросервісів

У статті узагальнено усталені підходи до декомпозиції прикладних програмних систем як інструменту керування складністю, еволюцією та ризиками змін упродовж життєвого циклу програмного продукту. Наголошено, що декомпозиція є не одноразовим технічним рішенням, а керованим процесом, який визначає межі відповідальності компонентів, характер їхніх залежностей, можливості паралельної розробки та локалізації дефектів. Розглянуто монолітний підхід як типовий стартовий стан системи, його переваги з погляду простоти розгортання й транзакційної узгодженості, а також обмеження, пов'язані зі збільшенням кількості прихованих зв'язків, ускладненням супроводу та зниженням передбачуваності змін. Описано еволюцію від шарового структурування до компонентного мислення, у межах якого функціональність групується навколо доменних відповідальностей і формалізованих інтерфейсів. Окремо проаналізовано концепцію модульного моноліту як архітектури єдиного розгортання з жорсткими внутрішніми межами, що надає можливість зменшити зчеплення, підвищити керованість модифікацій і зберегти відносно просту операційну модель. Також показано роль Domain-Driven Design у формуванні семантичних кордонів системи, зокрема концепції bounded context як основи для узгоджених доменних моделей і правил у різних частинах застосунку. Проаналізовано підходи до визначення меж декомпозиції за бізнес-можливостями та субдоменами, акцентовано їхню придатність для узгодження архітектурних рішень із тим, як еволюціонує предметна область та організаційні процеси. Окрему увагу приділено еволюційним підходам модернізації Strangler Fig і Anti-Corruption Layer, які забезпечують інкрементальну трансформацію системи, зменшення ризиків і можливість поступового відокремлення функціональності без сценарію «великого переписування». Сформульовано висновки щодо вибору підходу залежно від вимог до автономності компонентів, швидкості внесення змін, узгодженості даних та допустимого рівня операційної складності, а також окреслено типові компроміси, які виникають при переході від монолітних до більш дрібнозернистих архітектур.

Ключові слова: *програмна система; декомпозиція; модульний моноліт; мікросервіс; патерн декомпозиції; еволюційна модернізація; bounded context; Domain-Driven Design; Strangler Fig; Anti-Corruption Layer.*

Актуальність теми. Підвищення ефективності розробки та супроводу прикладних програмних систем значною мірою залежить від обраної архітектурної декомпозиції та стабільності меж між компонентами. Основними чинниками, які визначають керованість еволюції системи, є рівень зчеплення модулів, ступінь їхньої відповідальності та передбачуваність впливу змін на суміжні підсистеми. Порушення або нечіткість архітектурних меж призводить до накопичення прихованих залежностей, ускладнення тестування й локалізації дефектів, зростання ризику регресій та збільшення часу впровадження змін. На практиці це проявляється у зниженні темпу розвитку продукту та підвищенні вартості його підтримки, особливо в умовах постійного розширення функціональності й інтеграції із зовнішніми сервісами. Застосування сучасних підходів до декомпозиції – від модульного структурування монолітів до виділення компонентів за доменними межами та еволюційних стратегій модернізації – дозволяє забезпечити керованість архітектури, зменшити каскадність змін і підвищити відтворюваність інженерних рішень. Завдяки розвитку методологій Domain-Driven Design та практик інкрементальної модернізації (зокрема Strangler Fig і Anti-Corruption Layer) перспективним є впровадження підходів, які поєднують семантично обґрунтовані межі моделі з можливістю поступової трансформації системи без критичних перерв у її експлуатації. Отже, систематизація усталених варіантів декомпозиції та їхніх компромісів є актуальною для підвищення надійності архітектурних рішень і забезпечення сталого розвитку прикладних програмних систем.

Аналіз останніх досліджень та публікацій, на які спираються автори. Проблематика архітектурної декомпозиції програмних систем висвітлена у працях з інженерії програмного забезпечення та сучасних публікаціях, присвячених еволюції монолітних застосунків, мікросервісним підходам і домен-орієнтованому проєктуванню. У роботах М.Фаулера [1, 2] розглянуто монолітний підхід як практично

 © І.Д. Лімінович, Д.Д. Плечистий, Т.М. Локтікова, Н.О. Кушнір, 2026

доцільний стартовий варіант для багатьох продуктів, а також обґрунтовано ризики деградації структури монолітів за відсутності формалізованих меж та дисципліни модульності. Концептуальні засади Domain-Driven Design, які формують семантичні кордони системи через bounded context, узагальнено у роботах Еванса [3] та у подальших інтерпретаціях стратегічного дизайну [4], де підкреслюється значення узгодженої доменної моделі в межах визначеного контексту як умови зменшення прихованого зчеплення.

Частина наукових і прикладних робіт зосереджена на підходах еволюційної модернізації, які передбачають поетапне оновлення архітектури з контролем ризиків і збереженням працездатності системи. Зокрема, підхід Strangler Fig як практична стратегія поступового заміщення функціональності, описаний у рекомендаціях щодо декомпозиції монолітів [8], де наголошено на контролі ризиків і можливості паралельної експлуатації старих та нових компонентів. Питання семантичної ізоляції при інтеграції різних доменних моделей розкрито через патерн Anti-Corruption Layer [9], який забезпечує узгодження взаємодії між підсистемами без «проникнення» обмежень legacy-системи в новий дизайн.

Метою статті є аналіз та узагальнення усталених підходів до декомпозиції прикладних програмних систем, визначення їхніх ключових принципів, переваг і обмежень, а також формування обґрунтованих висновків щодо вибору архітектурної стратегії з урахуванням компромісів між керованістю змін, автономністю компонентів і операційною складністю.

Викладення основного матеріалу. Декомпозиція програмної системи в сучасній інженерії програмного забезпечення розглядається як спосіб зробити складність програми керованою, а її масштабування – передбачуваним. Питання «як поділити систему» фактично означає визначити межі відповідальності, правила формування залежностей, стабільні інтерфейси та механізми внесення змін без широкого узгодження. Якщо такі межі визначено невдало або вони існують лише формально, то система швидко накопичує приховані зв'язки: локальна зміна логіки породжує правки в багатьох місцях, зростає вартість тестування, а регресії стають важко пояснюваними. Тому декомпозицію коректно трактувати не як декоративне структурування коду, а як архітектурний інструмент, який задає довгострокові обмеження та дозволяє локалізувати зміни в межах підсистем.

На практиці найчастіше відправною точкою є моноліт, тобто застосунок, який розгортається як єдиний проєкт. Моноліт має очевидні переваги на ранніх етапах розробки: простіше впровадження змін, відсутність мережевих накладних витрат між компонентами, більш передбачуваний обмін інформацією між модулями та нижча операційна складність. Водночас «моноліт» не є синонімом «хаосу»: якість такого рішення визначається тим, чи існує внутрішня модульність і дисципліна залежностей. Стратегія «monolith first» підкреслює, що моноліт може бути раціональним вибором, якщо архітектурні межі та практики масштабування системи закладено заздалегідь [1]. Інакше моноліт швидко перетворюється на важкокеровану та заплутану систему, де оптимізація чи модернізація потребує непропорційних зусиль.

Структуру застосунку з монолітною архітектурою зображено на рисунку 1.

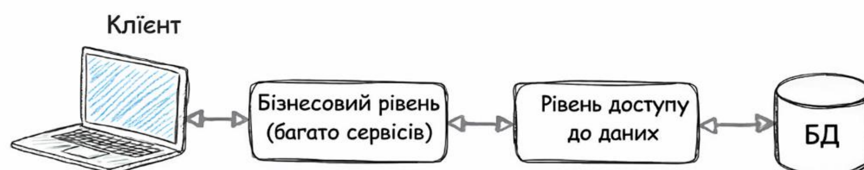

Рис. 1. Архітектура монолітного застосунку

Традиційне шарове структурування, яке передбачає відокремлення рівнів представлення, бізнес-логіки та доступу до даних, тривалий час залишалося базовою формою внутрішньої організації великих монолітних застосунків. Такий підхід насправді знижує локальну структурну хаотичність, забезпечуючи зрозумілу ієрархію абстракцій та певну уніфікацію реалізаційних рішень. Водночас, зі збільшенням системи, проявляється характерне обмеження: наскрізні бізнес-сценарії виявляються «розподіленими» між шарами, а зміни, які мають суттєве доменне значення, потребують узгоджених модифікацій на кількох рівнях одночасно. Наслідком цього є підвищення координаційних витрат під час внесення змін, збільшення ймовірності регресій та формування непрозорих ланцюжків взаємодії між компонентами системи. Додатково, шарова модель нерідко стимулює накопичення «спільних» сервісів і утилітарних підсистем, які поступово перетворюються на центри концентрації залежностей, унаслідок чого зростає зчеплення та погіршується локалізація впливу змін. У таких умовах архітектурні межі втрачають здатність виконувати стримувальну функцію, а якість і продуктивність системи деградують через неочевидні побічні ефекти та складність відтворення причинно-наслідкових зв'язків. Зазначені фактори обумовили

поширення підходів, які орієнтують декомпозицію на «вертикальні» зрізи – навколо функціональних можливостей або доменних відповідальностей, а не навколо технічних рівнів реалізації. У межах такого підходу межі визначаються передусім логікою предметної області, що підсилює узгодженість проєктних рішень із характером змін у бізнесі та спрощує контроль цілісності сценаріїв.

Проміжним і водночас самодостатнім варіантом між класичним монолітом та мікросервісними архітектурами стала концепція модульного моноліту. Її суть полягає у збереженні єдиного артефакту розгортання за умови внутрішньої структурної сегментації системи на модулі з чітко визначеними межами, власними інтерфейсами та контрольованими залежностями. У відповідних працях наголошується, що модуль у цьому контексті не зводиться до організаційної одиниці на рівні файлової структури, а розглядається як компонент, який інкапсулює пов'язану функціональність і надає доступ до неї через формалізований інтерфейс [2]. Отже, модульний моноліт потребує впровадження правил, які роблять межі операційно значущими, а не декларативними: прямий доступ до внутрішніх деталей інших модулів має бути обмежений, а міжмодульна взаємодія забезпечена прозорими контрактами та механізмами контролю залежностей. Практично це передбачає розмежування публічних і внутрішніх частин модуля, визначення стабільних точок інтеграції та усунення «обхідних» залежностей, які знижують автономність. Важливо, що модульний моноліт допускає інкрементальне впровадження модульності в наявні монолітні системи: декомпозиція може виконуватися поступово, без радикальної зміни операційної моделі, що зменшує ризики модернізації та надає можливість зберегти безперервність експлуатації.

Сучасні огляди модульних монолітів підкреслюють їхню цінність як архітектурного підходу, здатного скорочувати цикл впровадження змін завдяки формалізації внутрішніх меж і обмеженню неконтрольованих залежностей, зберігаючи при цьому відносно просту модель експлуатації порівняно з повністю розподіленими рішеннями. Зокрема, описані патерни модульного моноліту орієнтують на зменшення зв'язності між модулями та проєктування малих і стабільних API, які інкапсулюють значно більшу внутрішню реалізацію [3]. Як наслідок, підвищується передбачуваність впливу змін: модифікації доменної логіки можуть здійснюватися в межах конкретного модуля без непропорційного розповсюдження ефектів на інші частини системи. Це, в свою чергу, створює сприятливі умови для локальної оптимізації продуктивності та цілеспрямованого керування нефункціональними характеристиками, оскільки «критичні» сценарії мають чітко визначених власників і межі відповідальності. Додатково, модульні межі можуть слугувати природними одиницями стандартизації архітектурних рішень щодо кешування, транзакційності, обробки помилок і асинхронних взаємодій, що підвищує узгодженість технічної політики в межах системи. Зрештою, за умови стабільних внутрішніх контрактів і низької зв'язності, модульний моноліт формує передумови для подальшої еволюції архітектури: окремі модулі потенційно можуть бути виокремлені у самостійні сервіси без необхідності «великого перепроєктування» решти системи, якщо така потреба виникне на наступних етапах розвитку.

Водночас питання «як саме провести межу» не зводиться до технічної модульності; воно потребує семантичної основи. Таку основу пропонує Domain-Driven Design (DDD), у межах якого стратегічний дизайн розглядає великі домени як мережу контекстів. Ключовим поняттям є bounded context – обмежений контекст, у якому доменна модель має узгоджений сенс, а взаємодія між контекстами повинна бути явною [4]. Практична важливість bounded context полягає у тому, що він дозволяє розділити систему за сенсом, а не за технологіями. Як наслідок, модульний моноліт може бути реалізований як набір bounded context-модулів, де кожен модуль інкапсулює власну модель і правила, а інтеграція відбувається через контракти або події.

Поряд із DDD у джерелах інформації з мікросервісної архітектури сформувалися два популярні патерни встановлення меж: декомпозиція за бізнес-можливостями та декомпозиція за субдоменами. Перший підхід пропонує формувати компоненти відповідно до бізнес-можливостей організації, тобто до того, що бізнес робить для створення цінності, а не до внутрішніх технічних підсистем [5]. Другий підхід орієнтує на виділення компонентів відповідно до субдоменів DDD, які відображають різні частини предметної області з правилами та поняттями, що відрізняються [6]. Обидва варіанти корисні тим, що зменшують спокусу «порізати» систему механічно, наприклад, за таблицями бази даних або за набором CRUD-ендпоінтів, що рідко відповідає реальним незалежностям змін у бізнесі.

Якщо вимоги до автономності компонентів і незалежності їхнього життєвого циклу стають критичними, то організації переходять до розгляду мікросервісної декомпозиції. Мікросервісний підхід передбачає побудову системи як сукупності окремих сервісів, які розгортаються незалежно, взаємодіють через мережеві інтерфейси та спираються на формалізовані контракти [7]. У такій моделі архітектурні межі набувають не лише внутрішнього, а й операційного змісту: окремий сервіс стає самостійною одиницею розгортання, масштабування та моніторингу, а також зоною відповідальності команди, яка може планувати релізи без жорсткої синхронізації з іншими компонентами. Практично це відкриває можливість паралельного розвитку функціональності, диференційованого масштабування та локалізації інцидентів, однак водночас підвищує вимоги до узгодженості інтерфейсів і дисципліни взаємодії між сервісами.

Однак доцільно підкреслити, що мікросервісна архітектура не зводиться до поділу моноліту на кілька репозиторіїв або до контейнеризації компонентів. Формальне «розбиття» без зміни характеру залежностей часто призводить до утворення розподіленого моноліту, коли витрати на мережеву взаємодію та експлуатацію розподіленої інфраструктури зростають, тоді як автономність змін залишається обмеженою. Реальна автономність зазвичай потребує автономності даних (або принаймні чіткого розмежування відповідальності за моделі і сховища), що актуалізує питання узгодженості у розподіленому середовищі та зумовлює ширше застосування асинхронних взаємодій, ретраїв, ідемпотентності й механізмів компенсації. Крім того, мікросервісний підхід підвищує вимоги до спостережуваності, реліз-менеджменту та тестування інтеграцій, оскільки коректність і надійність системи визначаються не лише внутрішньою логікою сервісів, а й стабільністю їхньої взаємодії. Саме тому у багатьох командах мікросервіси стають виправданими лише за умови, що організаційні переваги незалежних життєвих циклів і масштабування суттєво переважають над зростанням операційної складності.

Структуру сайту інтернет-магазину з мікросервісною архітектурою зображено на рисунку 2.

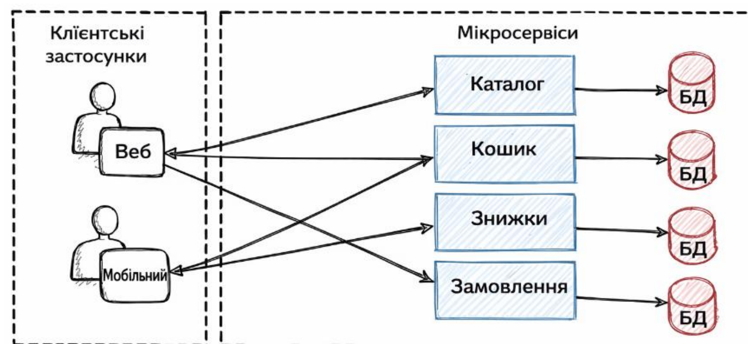

Рис. 2. Архітектура мікросервісного вебзастосунку

Однією з найбільш поширених архітектурних помилок під час переходу до мікросервісної архітектури є прагнення відтворити її зовнішні атрибути без досягнення ключової передумови – фактичного послаблення зв'язків між частинами системи. Наслідком такого підходу стає феномен «розподіленого моноліту», за якого система формально складається з окремих сервісів, однак зберігає властивості монолітної взаємозалежності. У подібній конфігурації зміни часто набувають синхронного характеру: реліз одного сервісу потребує одночасного оновлення інших через жорстку зв'язаність моделей, спільні бібліотеки, неявні припущення щодо форматів даних або нестабільні інтеграційні контракти. Окрім цього, мережеві виклики в межах критичних сценаріїв створюють додаткову латентність і підвищують чутливість системи до часткових відмов, що проявляється у каскадних збоях, деградації продуктивності та складності діагностики інцидентів.

Причини формування «розподіленого моноліту» зазвичай пов'язані з некоректним визначенням меж і невдалим вибором характеру взаємодії. Зокрема, проблемним є встановлення меж за технічними шарами (окремо «сервіс даних», «сервіс бізнес-логіки», «сервіс представлення») замість доменно-семантичних кордонів, оскільки це відтворює шарову фрагментацію моноліту вже у розподіленому середовищі. Іншим типовим чинником є надмірно гранульована синхронна взаємодія між сервісами, коли доменна операція реалізується як довгий ланцюжок послідовних RPC/HTTP-викликів. Така організація збільшує кількість точок відмови, ускладнює забезпечення узгодженості та провокує появу транзакцій, які фактично «розтягуються» між сервісами без адекватних механізмів компенсації. У результаті система втрачає здатність до незалежної еволюції: зміни, які мали би бути локальними, розповсюджуються через контракти й дані на суміжні компоненти, а межі сервісів перестають виконувати функцію ізоляції.

З огляду на це, коректна декомпозиція вимагає не лише структурного поділу коду або фізичного виділення компонентів у самостійні процеси, а й переосмислення контрактів, моделі володіння даними та механізмів взаємодії між частинами системи. Практично це означає формалізацію меж відповідальності на рівні доменної моделі, мінімізацію спільних залежностей, забезпечення стабільності інтерфейсів та проєктування інтеграцій так, щоби зменшити потребу в синхронній координації. Важливою передумовою автономності є також узгоджена стратегія управління даними, оскільки спільні сховища або «перехресні» запити до чужих моделей часто відновлюють сильну зв'язаність навіть при формальному поділі сервісів. У розподілених архітектурах суттєвого значення набувають асинхронні комунікації, ідемпотентність, механізми повторів і компенсацій, які дозволяють забезпечувати коректність бізнес-процесів у присутності часткових відмов і тимчасової неузгодженості станів.

Зважаючи на високу вартість та ризики сценарію «великого переписування», у практиці модернізації особливе місце посідають еволюційні підходи переходу між архітектурними станами. Strangler Fig

описують як спосіб інкрементальної трансформації моноліту у набір компонентів або сервісів шляхом поетапного заміщення окремої функціональності новою реалізацією, за умов тимчасового співіснування старого й нового рішень [8]. Концептуально цей підхід переносить акцент із одномоментної перебудови на керований процес серії малих змін, кожна з яких має вимірюваний результат і допускає зворотність. Практична цінність Strangler Fig полягає в можливості контролю ризиків через поступове перенаправлення трафіку, перевірку гіпотез під реальним навантаженням, збереження безперервності експлуатації та зниження ймовірності масштабних регресій за рахунок обмеження площі змін.

Водночас важливо зазначити, що застосування Strangler Fig не обмежується переходом саме до мікросервісів. У межах модульного моноліту цей підхід може виконувати роль механізму уточнення і закріплення кордонів: окремі модулі можуть поступово відокремлюватися у більш автономні компоненти, змінювати характер інтеграції (наприклад, від прямих викликів – до подій або черг), або виноситися у зовнішні сервіси тоді, коли їхня операційна та організаційна автономність стає економічно виправданою. Таким чином, Strangler Fig доцільно розглядати як загальну еволюційну стратегію керованого переходу між архітектурними формами, що дозволяє поєднати вимоги до стабільності експлуатації з необхідністю довготривалої архітектурної трансформації.

Якщо інтеграція між різними частинами системи неминуча, то надважливою стає проблема узгодження моделей. У цьому контексті Anti-Corruption Layer розглядається як прошарок між двома доменними моделями і не дозволяє одній частині «заражати» іншу своїми припущеннями [9]. З позиції декомпозиції це означає, що навіть за наявності legacy-компонентів або зовнішніх систем інтеграція може бути організованою так, щоби межа була семантичною: новий модуль або сервіс зберігає власну модель і правила, а не копіює структуру даних і терміни іншої підсистеми. Для довготривалого масштабування проєкту це важливо, бо саме неконтрольоване змішування моделей часто стає джерелом технічних проблем та ускладнює подальше розділення на модулі.

Вибір варіанту декомпозиції у прикладних системах визначається не лише технікою, але й соціально-організаційними чинниками. Навіть найкраще спроєктована модульність деградує, якщо немає механізмів підтримки меж: угод про публічні API модулів, правил залежностей, практик оглядів коду та інструментів автоматичної перевірки. У цьому сенсі модульний моноліт є вимогливою архітектурою: він дозволяє уникнути розподілених накладних витрат, але вимагає внутрішньої дисципліни та постійного контролю архітектурної цілісності. Натомість мікросервіси забезпечують жорсткі межі на рівні розгортання, але переносять частину складності в площину операцій: керування мережевими відмовами, спостережуваність, контроль версій, безпечні релізи та узгодженість даних.

На практиці доцільно розглядати декомпозицію як кероване наближення до автономності, де ключовим є баланс. Для багатьох прикладних систем раціональною є траєкторія, у якій система починається як моноліт, потім набуває рис модульного моноліту через доменну структуру та обмеження залежностей, а далі виносить лише ті частини, для яких вигоди автономності переважають вартість розподіленості. Такий підхід узгоджується з тезою про те, що архітектурний стиль має відповідати реальним потребам змін і масштабу, а не абстрактним «модним» вимогам. У результаті основне завдання декомпозиції полягає у тому, щоб створити такі межі, які одночасно є семантично осмисленими, технологічно здійсненними та підтримуваними процесами розробки.

Таким чином, сучасні варіанти декомпозиції утворюють не набір взаємовиключних підходів, а інструментарій для масштабування та розвитку проєктів. Моноліт, модульний моноліт і мікросервіси різняться не стільки кількістю файлів або контейнерів, скільки силою меж та вартістю їхньої підтримки. DDD із bounded context, декомпозиція за бізнес-можливостями та субдоменами, а також еволюційні патерни модернізації на кшталт Strangler Fig і Anti-Corruption Layer формують методичну базу для вибору і поступової зміни архітектури. Відтак, ефективна декомпозиція має оцінюватися через призму керованості змін, здатності локалізувати ризики, стабільності контрактів і реальної вартості експлуатації, а не через формальну відповідність певному тренду.

Висновки та перспективи подальших досліджень. Проведені огляд й аналіз свідчать про те, що архітектурна декомпозиція є одним із ключових інструментів керування складністю, еволюцією та ризиками змін у прикладних програмних системах. Встановлено, що традиційне шарове структурування, попри переваги впорядкування реалізації, у великих монолітах часто призводить до «розтікання» бізнес-сценаріїв між рівнями, зростання координаційних витрат і погіршення локалізації впливу змін. Це обґрунтовує доцільність переходу до доменно-орієнтованих підходів, у яких межі визначаються семантикою предметної області та відповідальністю за бізнес-результат, а не технічними шарами реалізації.

Показано, що модульний моноліт є самодостатнім архітектурним варіантом, який поєднує переваги єдиного розгортання з можливістю формалізації внутрішніх меж, інтерфейсів і правил залежностей. За умови реального контролю міжмодульної взаємодії та мінімізації прихованих зв’язків такий підхід підвищує передбачуваність змін, сприяє локалізації оптимізацій продуктивності та створює основу для подальшої еволюції системи. Водночас обґрунтовано, що мікросервісна декомпозиція є доцільною насамперед у ситуаціях, коли потреба в незалежних життєвих циклах компонентів і команд переважає над

зростанням операційної складності; при цьому ключовою умовою успішності виступає не формальний поділ коду, а автономність доменної відповідальності, контрактів і, зазвичай, даних. Наголошено, що ігнорування цих передумов призводить до феномену «розподіленого моноліту», який поєднує інфраструктурні витрати розподіленої системи з обмеженою автономністю змін.

Окремо визначено, що еволюційні підходи модернізації, зокрема Strangler Fig та Anti-Corruption Layer, є практично обґрунтованими механізмами переходу між архітектурними станами без сценарію «великого переписування». Їхнє застосування дозволяє знижувати ризики трансформації за рахунок поетапності, керованого перенаправлення трафіку, зворотності змін і семантичної ізоляції моделей під час інтеграції з legacy-компонентами.

Перспективи подальших досліджень доцільно пов’язувати з формалізацією критеріїв вибору між монолітною, модульно-монолітною та мікросервісною декомпозицією з урахуванням типових профілів навантаження, вимог до узгодженості даних та організаційних обмежень. Окремий інтерес становить розробка кількісних моделей оцінювання «вартості меж» (coupling cost) і прогнозування наслідків декомпозиційних рішень для темпу змін, ризику регресій та операційної складності. Також перспективним є дослідження практик автоматизованого контролю архітектурних меж (architecture fitness functions, CI-гейти) та методів емпіричної валідації декомпозиції на основі метрик продуктивності, надійності й керованості еволюції програмних систем.

Список використаної літератури:

1. *Fowler M.* Monolith First / *M.Fowler* [Electronic resource]. – Access mode : https://martinfowler.com/bliki/MonolithFirst.html.
2. *Brown S.* Modular monoliths / *S.Brown* [Electronic resource]. – Access mode : https://static.simonbrown.je/modular-monoliths.pdf.
3. *Richardson C.* Architectural patterns for modular monoliths that enable fast flow / *C.Richardson* [Electronic resource]. – Access mode : https://microservices.io/post/architecture/2024/09/09/modular-monolith-patterns-for-fast-flow.html.
4. *Fowler M.* Bounded Context / *M.Fowler* [Electronic resource]. – Access mode : https://martinfowler.com/bliki/BoundedContext.html.
5. *Richardson C.* Pattern: Decompose by business capability / *C.Richardson* [Electronic resource]. – Access mode : https://microservices.io/patterns/decomposition/decompose-by-business-capability.html.
6. *Richardson C.* Pattern: Decompose by subdomain / *C.Richardson* [Electronic resource]. – Access mode : https://microservices.io/patterns/decomposition/decompose-by-subdomain.html.
7. *Richardson C.* Pattern: Microservice architecture / *C.Richardson* [Electronic resource]. – Access mode : https://microservices.io/patterns/microservices.html.
8. Strangler fig pattern / AWS Prescriptive Guidance [Electronic resource]. – Access mode : https://docs.aws.amazon.com/prescriptive-guidance/latest/modernization-decomposing-monoliths/strangler-fig.html.
9. *Richardson C.* Pattern: Anti-corruption layer / *C.Richardson* [Electronic resource]. – Access mode : https://microservices.io/patterns/refactoring/anti-corruption-layer.html.
10. Онлайн-сервіс проходження курсів української мови / *О.С. Свістельник, Т.М. Локтікова, А.В. Морозов та інші* // Технічна інженерія. – 2023. – № 2 (92). – С. 137–145. DOI: 10.26642/ten-2023-2(92)-137-145.
11. *Товма О.* Основні типи архітектури програмного забезпечення / *О.Товма* [Електронний ресурс] – Режим доступу : https://www.artofba.com/uk/post/main-types-of-software-architecture.
12. Що таке мікросервісна архітектура: значення, складові, переваги [Електронний ресурс]. – Режим доступу : https://wezom.com.ua/ua/blog/scho-take-mikroservisna-arhitektura-znachennya-skladovi-perevagi.

References:

1. Fowler, M., «Monolith First», [Online], available at: https://martinfowler.com/bliki/MonolithFirst.html
2. Brown, S., «Modular monoliths», [Online], available at: https://static.simonbrown.je/modular-monoliths.pdf
3. Richardson, C., «Architectural patterns for modular monoliths that enable fast flow», [Online], available at: https://microservices.io/post/architecture/2024/09/09/modular-monolith-patterns-for-fast-flow.html
4. Fowler, M., «Bounded Context», [Online], available at: https://martinfowler.com/bliki/BoundedContext.html
5. Richardson, C., «Pattern: Decompose by business capability», [Online], available at: https://microservices.io/patterns/decomposition/decompose-by-business-capability.html
6. Richardson, C., «Pattern: Decompose by subdomain», [Online], available at: https://microservices.io/patterns/decomposition/decompose-by-subdomain.html
7. Richardson, C., «Pattern: Microservice architecture», [Online], available at: https://microservices.io/patterns/microservices.html
8. AWS Prescriptive Guidance, «Strangler fig pattern», [Online], available at: https://docs.aws.amazon.com/prescriptive-guidance/latest/modernization-decomposing-monoliths/strangler-fig.html
9. Richardson, C., «Pattern: Anti-corruption layer», [Online], available at: https://microservices.io/patterns/refactoring/anti-corruption-layer.html
10. Svistelnyk, O.S., Loktikova, T.M., Morozov, A.V. et al. (2023), «Onlain-servis prokhodzhennia kursiv ukrainskoi movy», *Tekhnichna inzheneriia*, No. 2 (92), pp. 137–145, doi: 10.26642/ten-2023-2(92)-137-145.

11. Tovma, O., «Osnovni typy arkhitektury prohramnoho zabezpechennia», [Online], available at: https://www.artofba.com/uk/post/main-types-of-software-architecture
12. *Shcho take mikroservisna arkhitektura: znachennia, skladovi, perevahy*, [Online], available at: https://wezom.com.ua/ua/blog/scho-take-mikroservisna-arhitektura-znachennya-skladovi-perevagi

Лімінович Іван Дмитрович – аспірант, асистент кафедри інженерії програмного забезпечення Державного університету «Житомирська політехніка».
https://orcid.org/0000-0003-2196-4186.
Наукові інтереси:
- веброзробка;
- оптимізація великих вебдодатків;
- інформаційні системи та технології.

E-mail: ivanliminovych@gmail.com.

Плечистий Дмитро Дмитрович – кандидат технічних наук, доцент, доцент кафедри комп’ютерних наук Державного університету «Житомирська політехніка».
https://orcid.org/0000-0002-4803-159X.
Наукові інтереси:
- комбінаторна оптимізація;
- інформаційні технології.

E-mail: kkn_pdd@ztu.edu.ua.

Локтікова Тамара Миколаївна – старший викладач кафедри інженерії програмного забезпечення Державного університету «Житомирська політехніка».
https://orcid.org/0000-0002-3525-0179.
Наукові інтереси:
- цифрова обробка зображень;
- інформаційні системи та технології.

E-mail: tamlokt@ukr.net.

Кушнір Надія Олександрівна – старший викладач кафедри інженерії програмного забезпечення Державного університету «Житомирська політехніка».
https://orcid.org/0000-0002-0797-3687.
Наукові інтереси:
- комбінаторна оптимізація;
- інформаційні технології.

E-mail: kipz_kno@ztu.edu.ua.

Liminovych I.D., Plechystyy D.D., Loktikova T.M., Kushnir N.O.

Review of modern approaches to decomposing applied software systems: from monoliths to modular monoliths and microservices

The article summarizes established approaches to decomposing applied software systems as a means of managing complexity, evolution, and change-related risks throughout the software product life cycle. It emphasizes that decomposition is not a one-time technical decision but a controlled process that defines component responsibilities, the nature of their dependencies, opportunities for parallel development, and defect localization. The monolithic approach is considered as a typical initial state of a system, highlighting its advantages in terms of deployment simplicity and transactional consistency, as well as its limitations caused by the accumulation of hidden couplings, increased maintenance complexity, and reduced predictability of change. The paper describes the evolution from layered structuring to component-based thinking, in which functionality is organized around domain responsibilities and formalized interfaces. Particular attention is given to the concept of a modular monolith as a single-deployment architecture with strict internal boundaries that reduce coupling, improve change management, and preserve a relatively simple operational model. The article also outlines the role of Domain-Driven Design in forming semantic system boundaries, in particular the concept of bounded context as a foundation for consistent domain models and rules across different parts of an application. Approaches to defining decomposition boundaries by business capabilities and subdomains are analyzed, with an emphasis on their usefulness for aligning architectural decisions with the evolution of the domain and organizational processes. Special attention is paid to evolutionary modernization approaches Strangler Fig and Anti-Corruption Layer which enable incremental architectural transformation, risk reduction, and gradual functional separation without a «big rewrite» scenario. Conclusions are formulated regarding the selection of an approach depending on requirements for component autonomy, change velocity, data consistency, and the acceptable level of operational complexity, and typical trade-offs that arise when moving from monolithic to more fine-grained architectures are outlined.

Keywords: software system; decomposition; modular monolith; microservice; decomposition pattern; evolutionary modernization; bounded context; Domain-Driven Design; Strangler Fig; Anti-Corruption Layer.

Стаття надійшла до редакції 30.12.2025.