ISSN 2707-1898 (print)

Український журнал інформаційних технологій
Ukrainian Journal of Information Technology
http://science.lpnu.ua/uk/ujit
https://doi.org/10.23939/ujit2023.01.009
Article received 28.04.2023 p.
Article accepted 02.05.2023 p.
UDK 004.[05+42]

Correspondence author
M. M. Seniv
maksym.m.seniv@lpnu.ua

В. Я. Лахай, О. М. Кузьмич, М. М. Сенів
Національний університет "Львівська політехніка", м. Львів, Україна

УДОСКОНАЛЕНИЙ МЕТОД ПІДВИЩЕННЯ ЗРУЧНОСТІ СУПРОВОДУ ЗА УМОВ ЗАСТОСУВАННЯ БЕЗСЕРВЕРНОЇ АРХІТЕКТУРИ

З'ясовано, що досягнення високих показників якості програмного забезпечення (ПЗ), зокрема зручності його супроводу (англ. maintainability [1]), з кожним робом стає дедалі більшою проблемою через появу нових технологій, зокрема хмарних технологій та безсерверної архітектури. Попри це, вимоги до рівня зручності супроводу ПЗ тільки зростають, а дана характеристика його якості отримує все більше уваги. Раніше розроблені науковими та інженерними спільнотами численні інструменти, методології, підходи та методи розробки ПЗ для його підвищення якості є недостатньо ефективними за наявних умов. Було підтверджено, що застосування удосконаленого методу підвищення зручності супроводу ПЗ за умов застосування безсерверної архітектури є нагальною необхідністю. Було проаналізовано поточний стан підходів до підвищення зручності супроводу ПЗ. Внаслідок цього була обрана так звана "чиста архітектура" (з англ. Clean Architecture) як найкращий наявний підхід із причини того, що він успадковує основні переваги конкурентних підходів, надає чіткіші настанови та охоплює ширший спектр процесу розроблення ПЗ. Незважаючи на те, що він забезпечує істотне підвищення основних характеристик зручності супроводу ПЗ, таких як модульність (англ. modularity) і зручність перевикористання (англ. resusability), все ще є інші характеристики, які потрібно розглянути, такі як аналізованість (англ. analyzability), модифікованість (англ. modifiability) і тестування (англ. testability). Було визначено вимоги та розроблено удосконалений метод підвищення зручності супроводу ПЗ за умов застосування безсерверної архітектури, який задовольняє всі попередньо сформульовані вимоги. Зокрема, цей підхід покращує роботу з залежностями, надає базову структуру для компонент, підвищує згуртованість та зменшує зчепленість функціоналу. Для оцінювання ефективності створеного методу було розроблено дві реалізації одного й того ж проекту: на підставі Clean Architecture та з використанням удосконаленого методу. Для них були розраховані такі метрики оцінювання зручності супроводу ПЗ: показник зручності супроводу ПЗ (англ. Maintainability Index, MI) та його варіація, за якої до уваги беруться тільки файли з показником менше 100, яку будемо називати відфільтрований показник зручності супроводу ПЗ (англ. Filtered Maintainability Index, FMI). Проаналізувавши отримані результати, було виявлено, що застосування удосконаленого методу підвищує значення першої метрики на 2,1 % та другої на 8,3 %. Отже, було доведено ефективність розробленого вдосконаленого методу.

Ключові слова: чиста архітектура, інверсія залежностей, ін'єкція залежностей, безсерверність.

Вступ / Introduction

У відповідь на зростаючу потребу в обчислювальних потужностях багато організацій створили власні центри оброблення даних. Основним недоліком цього підходу є потреба створення та управління як програмними, так і апаратними компонентами інформаційних систем. Оскільки потреба в обробці та зберіганні даних тільки зростає, це призводить до значних труднощів, пов'язаних із масштабуванням таких систем. Вирішенням проблеми масштабування для апаратної складової є використання хмарних технологій. Для програмної складової, враховуючи вимоги до високої гнучкості та надійності, рішенням стала поява нових типів архітектур розроблення ПЗ, зокрема безсерверних і мікросервісних.

Незважаючи на те, що хмарні обчислення та нові архітектури вирішували попередні проблеми, їх застосування має сильний зв'язок із зниженням зручності супроводу ПЗ, однієї з найважливіших характеристик якості програмного забезпечення. Основною причиною її зниження є те, що ці технології є більш досконалими, а їх правильна адаптація складніша. Окрім цього, минуло не так багато часу з моменту їх впровадження, і, як наслідок, нові підходи та методи розроблення ПЗ для їх ефективного використання не достатньо поширені. Раніше для підтримки зручності супроводу ПЗ на прийнятному рівні науковими та інженерними спільнотами були розроблені численні інструменти, методології, підходи та методи розроблення ПЗ. Найпопулярнішими з них є Clean Architecture, SOLID, Test Driven Development (від англ. Керована тестами розроблення, TDD) і Domain Driven Design (від англ. Предметно-орієнтоване проектування, DDD). Через привнесену хмарними технологіями та новими архітектурами складність розроблення ПЗ, дані методи не дають змогу досягати такого ж рівня якості програмного забезпечення, як і раніше. Отже, актуальною науково-прикладною задачею є розроблення нових і удосконалення наявних методів підвищення якості, зокрема підвищення такої її характеристики як зручність супроводу ПЗ.

Об'єкт дослідження – процес підвищення якості програмного забезпечення.

Предмет дослідження – методи, засоби і підходи для підвищення зручності супроводу ПЗ за умов застосування безсерверної архітектури.

Мета роботи – створення удосконаленого методу підвищення зручності супроводу ПЗ за умов застосування безсерверної архітектури.

Для досягнення зазначеної мети визначено такі *основні завдання дослідження:*

- провести аналіз наявних підходів до підвищення зручності супроводу ПЗ;
- обрати підхід-основу та визначити характеристики, які потрібно удосконалити;
- на підставі удосконалення згаданих вище характеристик запропонувати власний метод, який, на відміну від наявних, дасть змогу підвищувати зручність супроводу ПЗ за умов застосування безсерверної архітектури;
- розробити демонстраційне ПЗ та верифікувати ефективність запропонованого методу.

Аналіз останніх досліджень та публікацій. Недостатнє висвітлення проблеми підвищення зручності супроводу ПЗ зумовлено її більш прикладною, ніж науковою спрямованістю, та емпіричним типом даних, хоча є ряд робіт, які висвітлюють цю тему [2–5]. Наприклад, у роботі [2] стверджується, що розробники повинні прагнути до мінімальної складності, оскільки підвищена складність створює ризики безпеці програмного забезпечення, що, водночас, значно знижує надійність програмного забезпечення. Окрім цього, нездатність підтримувати низьку складність і простий структурний дизайн негативно вплине на гнучкість програмного забезпечення, яка є одним із найважливіших показників якості програмного забезпечення. Також варто зазначити, що підвищення складності програмного забезпечення погано впливає не тільки на програмне забезпечення, а й на бізнес [2].

Зручність супроводу ПЗ складається з таких характеристик [1]: модульність; зручність перевикористання; аналізованість; зручність внесення змін; зручність тестування.

Зручність тестування, зручність внесення змін та аналізованість мають вискоий рівень залежнсоті від їх спільної характеристики – структурованості. Структурованість складається з таких характеристик: згуртованість (англ. cohesion), зчеплення (англ. coupling) та розмір. Хорошою структурованість вважається за умови високої згуртованості та низького зчеплення. Це вимагає організації подібного функціоналу разом в одному класі або модулі [3].

У роботі [4] автор розглядає концепцію архітектури програмного забезпечення в різні періоди часу, а саме минуле, теперішнє та майбутнє. Стверджується, що найбільш релевантними архітектурами на даний момент є безсерверні та мікросервісні архітектури. Причиною їх появи є збільшення проектів, функціональності та зростання вимог до якості програмного забезпечення, особливо до зручності супроводу ПЗ. Перехід до цих архітектур ще більше ускладнив розробку програмного забезпечення.

Зі збільшенням складності погоджуються автори роботи [5]. Також у цій роботі зазначено, що підходи, які мали зменшувати цю складність, втратили свою актуальність через перехід до нових типів архітектур.

Основні причини програмних збоїв розглянуто у роботі [6]. Автори стверджують, що програма переважно дає збій двома способами: логічна помилка та технічна помилка (виняток). Технічна помилка – це збій, викликаний технічною неможливістю виконання визначеного завдання. Зазвичай виникнення таких помилок пов'язане з недостатнім прогнозуванням можливих помилок у роботі програми та їх обробкою, що водночас виникає внаслідок нерозуміння використовуваних технологій. Логічна помилка – це помилка, при якій програма дає несподівані результати за відсутності технічних помилок. Найбільш частою причиною їх появи є нерозуміння предметної області та взаємодій між компонентами розробленої системи. Отже, підвищення зручності супроводу ПЗ може допомогти покращити інші характеристики програмного забезпечення. Окрім цього, у роботі [3] стверджується, що подолання проблеми зручності супроводу ПЗ можливе за допомогою хороших підходів до розроблення ПЗ.

Автор дослідження [6] розглядає можливість збільшення якості програмного забезпечення за допомогою підходу TDD, оскільки багато досвідчених розробників програмного забезпечення, включаючи Фаулера, вважають, що якість виграє від написання тестів перед створенням коду. Хоча зручність тестування є важливою частиною зручності супроводу ПЗ, TDD зосереджується на невеликій частині процесу розроблення ПЗ, і її переваги можна відтворити за допомогою інших підходів. Перевагами є підвищення зручності тестування, а також те, що тести можуть відіграти роль документації [7]. З іншого боку, тести не гарантують коректну роботу написаного коду. Такий підхід вимагає більше часу на розробку, а також ознайомлення всіх інженерів із правильним написанням тестів. Окрім цього, це вимагає зміни тестів щоразу, коли виникає потреба змінити функціональність [7].

З висновків у роботі [8], які були отримані на практиці, стає зрозумілим, що використання підходів до розроблення ПЗ, а саме DDD, дає змогу підвищити зручність супроводу ПЗ. Його перевагами є зосередження основної уваги на предметній області та створення програмних моделей, які відображають глибоке розуміння предметної області, а також підвищення ефективності спілкування між інженерами, що створюють програмне забезпечення, і фахівцями в предметній області. З іншого боку, DDD вимагає від інженерів глибоких знань предметної області та більше часу на розробку, що, водночас, збільшує вартість розроблення. Його використання невиправдано в проектах із відносно простою предметною областю [9].

Згідно із підходом Clean Architecture, сучасне програмне забезпечення має занадто багато залежностей від речей, які не є справжньою бізнес-логікою та часто змінюються: баз даних, інтерфейсу користувача, вебсервісів, пристроїв тощо. Цей підхід дає можливість зменшити кількість залежностей бізнес-логіки із зовнішніми сервісами та покращити зручність супроводу ПЗ, оскільки принцип інверсії залежностей (англ. Depdendency Inversion Principle) є основним використовуваним принципом. Як наслідок, це допомагає підвищити модульність, аналізованість та можливість повто-

рного використання, а також тестування на рівні одиниці. З іншого боку, введення шаблонного коду та відсутність суворих інструкцій вважаються основними недоліками. З боку зручності супроводу ПЗ, Clean Architecture недостатньо сприяє підвищенню здатності до аналізу, модифікації та тестування. Звичайні практики можуть навіть призвести до зниження цих характеристик. Також важливо зазначити, що чиста архітектура має більш абстрактний характер, аніж прикладний та не надає чітких інструкцій для застосування [10].

Визначимо метрики, якими можна виміряти зручність супроводу ПЗ. Існує не так багато показників для оцінювання зручності супроводу ПЗ. Основними з них є цикломатична складність, міри складності Холстеда та показник зручності супроводу ПЗ. Оскільки показник зручності супроводу ПЗ містить цикломатичну складність та міру Холстеда, саме ця метрика буде використовуватись [11]. Дана метрика розраховується за формулою 1.

$$MI = \max\left[0{,}100\frac{171-5{,}2\ln(V)-0{,}23G-16{,}2\ln(L)+50\sin(\sqrt{2{,}4C})}{171}\right]. \quad (1)$$

З наведеного вище можна підсумувати, що проблема недостатньої зручності супроводу ПЗ є надзвичайно актуальною на тлі переходу галузі на нові архітектури розроблення ПЗ. Однак ця проблема недостатньо широко висвітлена науковцями у зв'язку з більш прикладними, ніж науковими питаннями. З наукових праць, які описують цю проблему, стає зрозуміло, що вирішення її полягає у створенні удосконаленого методу за умов застосування безсерверної архітектури для підвищення зручності супроводу ПЗ.

Результати дослідження та їх обговорення / Research results and their discussion

Формування основи вдосконаленого методу

Розглянувши результати проведеного аналізу, було визначено, що підхід Clean Architecture є оптимальним для розроблення вдосконаленого методу на його основі. Він успадковує основні переваги DDD та TDD, надає чіткіші настанови та охоплює ширший спектр процесу розроблення ПЗ. Незважаючи на те, що він уже забезпечує істотне підвищення основних характеристик зручності супроводу ПЗ, таких як модульність і зручність перевикористання, все ще є інші характеристики, які потрібно розглянути, такі як аналізованість, зручність внесення змін та зручність тестування. Отже, вдосконалений метод повинен базуватись на підході Clean Architecture та пропонувати шляхи підвищення зручності тестування, зручності внесення змін та аналізованості.

Зручність тестування має високий рівень залежності від організації роботи із залежностями в коді. Попри те, що Clean Architecture базується на DIP, подальше вдосконалення роботи із залежностями є необхідністю.

Оскільки, як було визначено під час аналізу, зручність тестування, зручність внесення змін та аналізованість мають високий рівень залежності від структурованості, їх підвищення можна досягти саме через підвищення цієї характеристики. Це можливо шляхом підвищення згуртованості та зменшення зчепленості функціоналу. Іншим способом підвищення структурованості є формування базової структури компонент.

Отже, розроблення вдосконаленого методу складається з таких кроків:

Крок 1: Покращення роботи із залежностями.

Крок 2: Формування базової структури компонента.

Крок 3: Підвищення згуртованості та зменшення зчепленості функціоналу.

Крок покращення роботи із залежностями

DIP дає змогу бізнес-логіці залежати від абстракцій, а не від деталей реалізації. Проте даний принцип не надає інструкцій як ці залежності варто ініціалізувати та надавати у використання логіці сервісів. Окрім цього, залежностей може бути багато, і їх створення та логіка передачі можуть ускладнювати код. Із боку можливості тестування, важливо мати логіку передачі залежностей якомога більш простою. І для цього є рішення – ін'єкція залежностей.

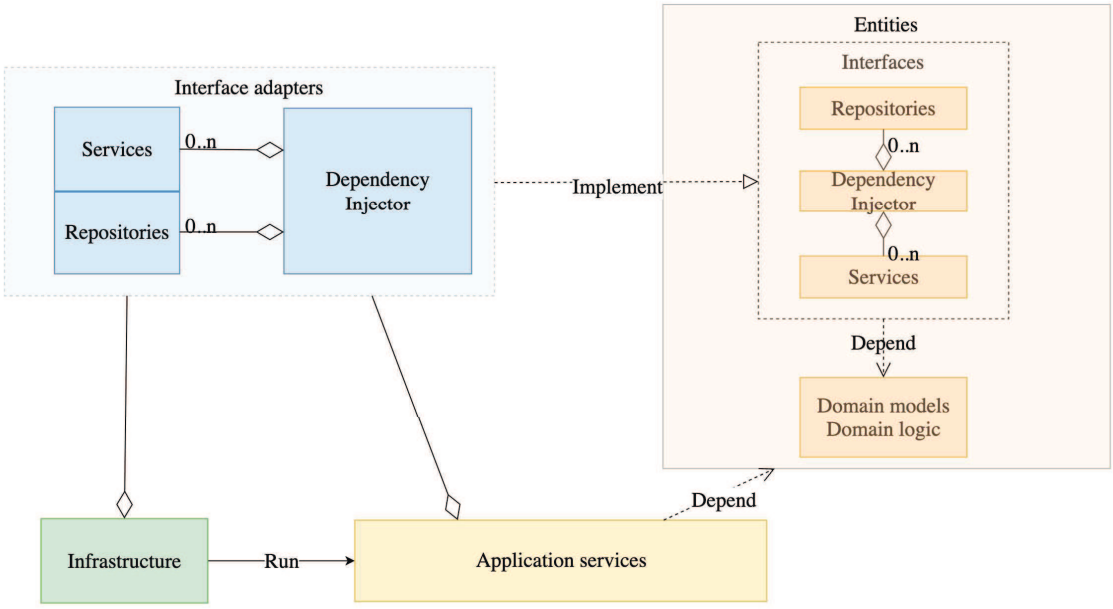

Рис. 1. UML діаграма вдосконаленого методу з доданим інжектором залежності / The UML class diagram of the improved metod with added dependency injector

Бізнес-логіка повинна знати тільки те, як використовувати залежності, а не те, як їх ініціювати. Інтерфейс інжектора залежностей має бути частиною рівня сутностей, як наведено на рис. 1, із конкретними реалізаціями на рівні адаптерів інтерфейсів. Відповідальним за ініціювання має бути рівень інфраструктури, який і передає ініційований інжектор до сутностей рівня сервісів додатку. Інжектор залежностей має містити інші об'єкти, які мають подібний підхід, наприклад, репозиторії.

Найкращий спосіб заповнити бізнес-логіку залежностями – це розпакувати об'єкти інжектора залежностей у конструкторах класів рівня сервісів додатку. Окрім цього, немає сенсу створювати кілька інжекторів залежностей, достатньо одного для кожного домену. Цей об'єкт повинен знати, як ініціалізувати інші адаптери інтерфейсів із реальних і тестових даних.

Для ініціалізації інжектора залежностей необхідна інформація, наприклад облікові дані для баз даних, ARN (імена ресурсів Amazon) для SNS ресурсів, змінні параметри тощо. Переважно на реальній інфраструктурі така інформація може бути надана через змінні середовища, але у більш складних випадках можуть бути звернення до баз даних та інших сервісів. Отже, у реальних середовищах інжектор залежностей має збирати дані зі змінних середовища та інших служб. Для тестування спрощений та уніфікований механізм встановлення залежностей є необхідністю. Коли справа доходить до баз даних, найкращим способом є використання бібліотек емуляції. Для сервісів доцільніше створити тестову версію адаптера інтерфейсу з фіктивними відповідями та деяким внутрішнім станом. Для цілей тестування часто достатньо мати один шаблон ініціалізації інжектора залежностей з усіма адаптерами інтерфейсів та повторно використовувати його в усіх тестах.

Крок формування базової структури компонента

Взаємопов'язані компоненти варто розкласти на оптимальну кількість невеликих модулів за однаковою схемою. Застосування однакового шаблону для всіх компонент іноді може бути неможливим через їхню різну природу, залежності тощо. Тому важливо не дотримуватися цього правила суворо. Загальний шаблон повинен існувати, проте може бути кілька шаблонів, які розширюють його, але не замінюють.

Стандартний компонент у безсерверній архітектурі представляє функціонал, відповідний одній лямбда функції в системі. Введення базової структури вимагає поділу цієї функціональності на модулі з низьким зчепленням.

Найкращий спосіб зробити це – дотримуватися SRP (Single Responsibility Principle, від англ. Принцип Єдиної Відповідальності). Компоненти мають такі обов'язки: отримувати події, виконувати деякі операції над витягнутими з них даними та надсилати події у відповідь. Отже, взаємодія з подіями є незалежною відповідальністю, і її можна вилучити з бізнес-логіки. Внаслідок цього ми можемо розділити компоненти принаймні на три модулі: lambda_hadler, event_data, processor.

Модуль event_data містить класи подій, які знають про структури подій, щоб вони могли інкапсулювати логіку вилучення з них корисної інформації. Також там можна виконувати перевірку подій відповідно до схем і бізнес-правил, а також стандартні операції реєстрації. Внаслідок цього бізнес-логіка не залежить від зміни структури подій, вона використовує суворо визначений інтерфейс для взаємодії з даними.

Модуль lambda_handler відповідає за отримання всіх необхідних даних із зовнішнього світу для ініціалізації залежностей та ініціалізації класів подій. Після ініціалізації він надає їх бізнес-логіці.

Модуль processor – це чиста бізнес-логіка, і всі його залежності від зовнішнього світу ретельно інкапсульовані в інших класах і модулях: ін'єктори залежностей, сховища, служби, класи подій.

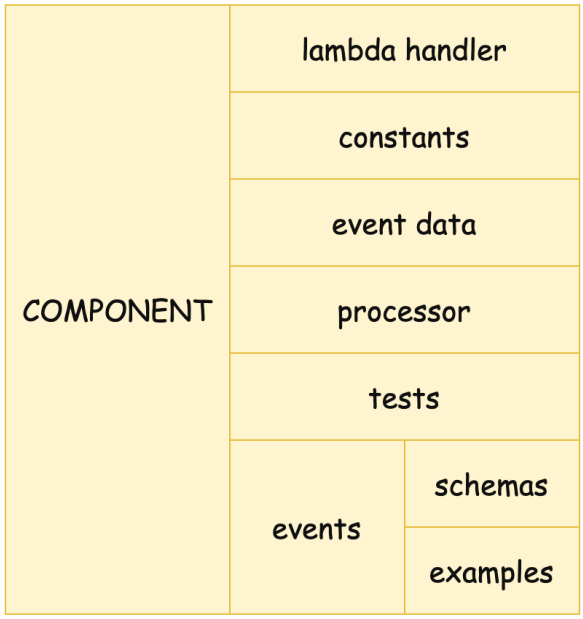

Рис. 2. Файлова структура вдосконаленого методу / The file structure of the improved metod

Окрім цього, розділення констант, класів помилок та інших спільних речей покращує структуру. Найкраще місце для тестів – у тій же самій папці, їх не варто переміщувати кудись інше. Зберігати схеми подій і приклади разом із кодом, який має їх обробляти, вважається хорошою практикою. Внаслідок цього вийде базова структура компоненту, як зображено на рис. 2.

Крок підвищення згуртованості та зменшення зчепленості функціоналу

Щоб підвищити згуртованість та зменшити зчеплення, дуже пов'язаний функціонал варто зберігати разом, а не розділяти на рівні сервісів додатку. Наприклад, ми можемо мати 10 сервісів додатку для маніпулювання даними користувачів. Їх логіка дуже пов'язана, а її організація в кількох модулях робить водночас їх пов'язаними. Отже, функціонально пов'язана логіка повинна бути виведена з прикладних служб. Але ця логіка може бути частиною рівня сутностей тільки в найпростіших випадках, коли немає залежності від зовнішнього світу. У всіх інших ситуаціях таку логіку недоцільно розташовувати на рівні сутностей. Отже, рівень із ним має бути між рівнем сервісів додатку і рівнем сутностей і може бути названий як сервіси домену.

Сервіси домену – це рівень, який повинен містити пов'язану функціональність, щоб служби додатків могли мати високу згуртованість та низьку зчепленість.

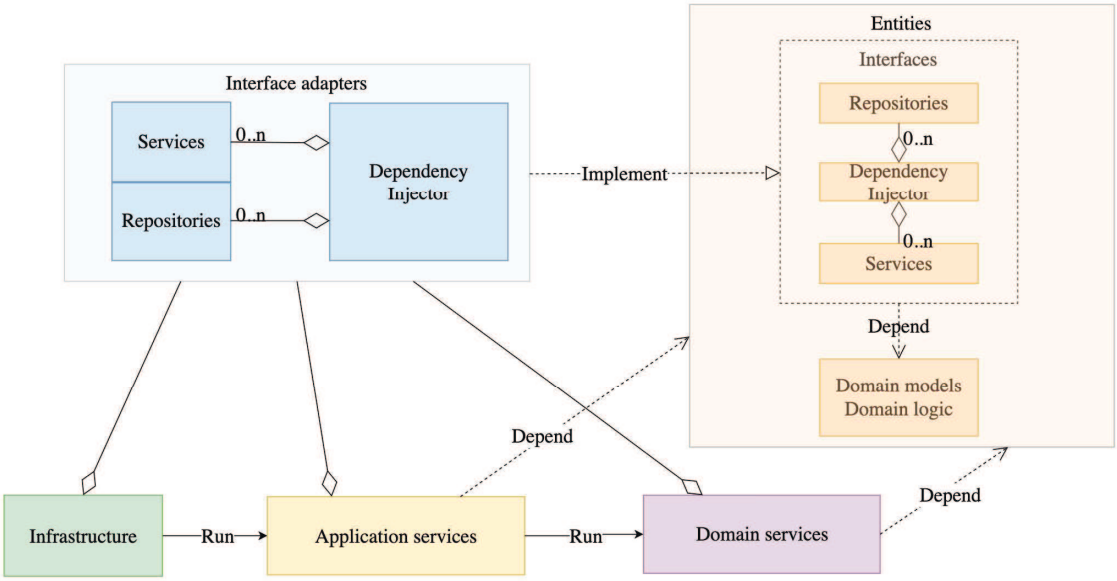

Рис. 3. Фінальна UML діаграма класів вдосконаленого методу / The final UML class diagram of the improved metod

***Приклад* 3**. Потрібно обчислити значення частинних

Опис розробленого вдосконаленого методу

На рис. 3 наведено діаграму класів сформованого методу. Відповідно до нього, архітектура повинна складатися з таких рівнів: сутностей, сервісів домену, сервісів додатків, адаптерів інтерфейсів та інфраструктури. Основне вдосконалення тут стосується впровадження рівня сервісів домену, який має зменшити зчеплення між компонентами та, як наслідок, покращити структурованість і всі три цільові характеристики.

Іншим важливим аспектом розробленого методу є використання техніки ін'єкції залежностей для покращення оброблення залежностей. Дане вдосконалення дає змогу підвищити зручність тестування.

Також було сформовано базову структуру компонента для безсерверної керованої подіями архітектури. Це має однаковий позитивний вплив на всі три цільові характеристики.

Внаслідок цього було запропоновано шляхи підвищення зручності тестування, зручності внесення змін та аналізованості.

Розроблення ПЗ із використанням удосконаленого методу

Запланований проект є відносно невеликим, тому буде багато простих файлів із найвищими показниками показника зручності супроводу ПЗ і не так багато із показниками нижче. Оскільки розроблений метод має покращити зручність супроводу ПЗ, його основна увага зосереджена саме на тих файлах, які мають нижчі оцінки показника зручності супроводу ПЗ. Отже, щоб оцінити реальне покращення зручності супроводу ПЗ, буде корисно підраховувати середнє значення показника зручності супроводу ПЗ тільки для тих файлів, чиє значення показника зручності супроводу ПЗ нижче 100. Такий показник можна назвати відфільтрованим показником зручності супроводу ПЗ.

Тому для оцінювання показника покращення зручності супроводу ПЗ було обрано дві метрики: показник зручності супроводу ПЗ та відфільтрований показник зручності супроводу ПЗ.

Є кілька вимог до демонстраційного проекту ПЗ:

1. Він повинен мати принаймні дві залежності: одну, до якої можна отримати доступ за допомогою репозиторію, а іншу – за допомогою служби.
2. Він повинен мати принаймні п'ять сервісів додатку.

Всі ці вимоги можуть бути задоволені проектом "Студентське управління". Цей проект повинен мати наступний функціонал:

- Створення профілю студента.
- Видалення профілю студента.
- Зміна прізвища в профілі студента.
- Зміна номера телефону в профілі студента.
- Зміна назви групи для всіх профілів студентів.
- Повідомляти державну службу про зміни студентів, які її цікавлять.

Для реалізації даного функціоналу програмне забезпечення повинно мати доступ до таблиці Student, щоб можна було реалізувати репозиторій. Також функція взаємодії з державною службою може бути реалізована за допомогою сервісу. Проект повинен бути покритий тестами.

Основні технології, які будуть використовуватися в обох реалізаціях: Python, AWS Lambda, Amazon DynamoDB, Amazon SNS.

Реалізація проекту на підставі Clean Architecture

Було розроблено демонстраційне програмне забезпечення на підставі Clean Architecture (рис. 4).

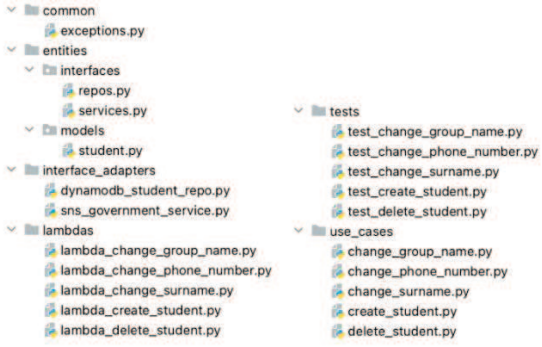

Рис. 4. Файлова структура реалізації проекту на підставі Clean Architecture / The file structure of the project implementation based on Clean Architecture

Як видно зі структури, всі рівні мають власні папки. У use_cases у нас є 5 файлів для кожної частини функціоналу та відповідних їм 5 файлів у обрробниках лямбда-функцій і тестах (рис. 5).

```
logger = logging.getLogger('createNewStudentLogger')
dynamodb = boto3.resource('dynamodb', region_name='eu-central-1')
student_table = dynamodb.Table(os.environ.get('ENV_STUDENT_TABLE_NAME'))
student_repo = StudentRepo(student_table)
sns = boto3.client("sns", region_name="eu-central-1")
government_service = GovernmentService(sns=sns, topic=os.environ.get('ENV_GOVERNMENT_SNS_TOPIC_ARN'))


Vladyslav Lakhai *
def handler(event, context):
    return create_student(logger=logger, event=event,
                          student_repo=student_repo, government_service=government_service)
```

Рис. 5. Приклад lambda_handler у реалізації проекту на підставі Clean Architecture / An example of lambda_handler in the implementation of a project based on Clean Architecture

Реалізація проекту на підставі розробленого методу

Було розроблено демонстраційне програмне забезпечення на підставі удосконаленого методу (рис. 6).

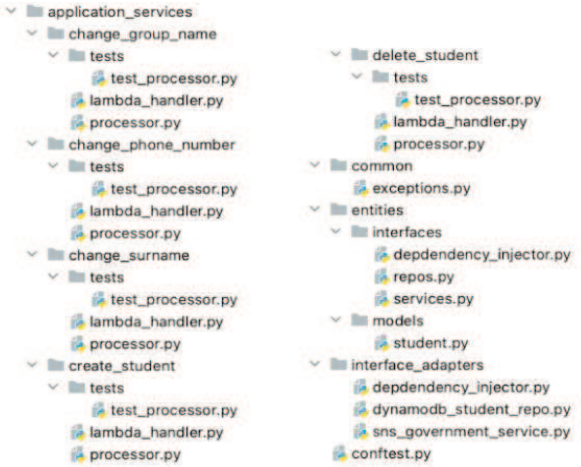

Рис. 6. Файлова структура реалізації проекту на підставі розробленого методу / File structure of project implementation based on the developed metod

Як видно зі структури, рівень сервісів домену відсутній. Раніше згадувалося, що цей рівень необов'язковий і його можна опустити у випадках, коли бізнес-логіка не надто складна. У цьому випадку 5 сервісів додатку є недостатніми, щоб виправдати впровадження цього рівня.

У цій реалізації було представлено інжектор залежностей, який допоміг значно спростити роботу із залежностями, особливо в обробниках лямбда-функцій, тестах і файлах сервісів додатку (рис. 7).

```
di = DependencyInjector.initialize_dependencies()


Vladyslav Lakhai
def handler(event, context):
    return create_student(di=di, event=event)
```

Рис. 7. Приклад lambda_handler у реалізації проекту на підставі розробленого методу / An example of lambda_handler in project implementation based on the developed metod

Окрім цього, не всі рекомендації щодо структури компонент, викладені в описі методу, були дотримані через малий розмір проекту. Наприклад, константи, event_data та папки подій були пропущені.

Було підготовлено реалізації підхіду Clean Architecture та удосконаленого методу. Щоб оцінити метрики, було зібрано інформацію про MI за допомогою інструменту Radon та запущено скрипт обчислення (табл. 1, 2).

Табл. 1. Результати обчислення Maintainability Index / The results of the Maintainability Index calculation

Clean Architecture	87,52705156479156
Improved Approach	89,65277751454305

Табл. 2. Результати обчислення Filtered Maintainability Index / The results of the Filtered Maintainability Index calculation

Clean Architecture	56,34468047677044
Improved Approach	64,52380862129044

З отриманих результатів чітко видно збільшення показників MI та FMI. MI зріс на 2,1 % із 87,5 до 89,6, а FMI піднявся на 8,3 % із 56,3 до 64,5. Отже, було доведено ефективність покращеного методу.

Для кожної реалізації було оцінено визначені метрики. Підвищення показника зручності супроводу ПЗ на 2,1 % та відфільтрованого показника зручності супроводу ПЗ на 8,3 % підтвердили ефективність створеного методу.

Обговорення результатів дослідження. Упродовж останніх років питання підходу для підвищення зручності супроводу програмного забезпечення набуло загострення через використання нових технологій, зокрема хмарних обчислень та безсерверної архітектури. До розгляду цього питання долучилась низка авторів у роботах [2–5]. Кожен із них частково описав причини та наслідки змін, які відбулись із розробкою програмного забезпечення при переході до безсерверної архітектури. Існують підходи для підвищення зручності супроводу програмного забезпечення, розроблені науковими та інженерними спільнотами [6–10]. Окремої уваги заслуговує підхід Clean Architecture [10], оскільки він успадковує основні переваги підходів DDD [8] та TDD [7], надає чіткіші інструкції та охоплює ширший спектр процесу розробки. Незважаючи на те, що він уже забезпечує суттєве підвищення основних характеристик зручності супроводу, таких як модульність і зручність багаторазового використання, все ще є інші характеристики, які потрібно розглянути, такі як аналізованість, зручність внеснення змін і зручність тесту-

вання. Запропонований удосконалений підхід покликаний усунути ці недоліки.

Отже, за результатами роботи можна сформулювати наукову новизну і практичну значущість результатів дослідження.

Наукова новизна отриманих результатів дослідження – вперше розроблено метод підвищення зручності супроводу ПЗ за умов застосування безсерверної архітектури.

Практична значущість результатів дослідження – розроблений метод підвищення зручності супроводу ПЗ за умов застосування безсерверної архітектури можна використати на практиці під час розроблення будь якого безсерверного ПЗ.

Висновки / Conclusions

Розроблено вдосконалений метод до підвищення зручності супроводу ПЗ за умов застосування безсерверної архітектури. За результатами дослідження можна зробити такі основні висновки:

1. Розглянуто передумови значного попиту на хмарні технології та безсерверну архітектуру, а також підтверджено проблему зниження зручності супроводу ПЗ у цих умовах. Визначено, що цю проблему можна подолати за допомогою підходів і методологій розроблення ПЗ. Також визначено, що проблема недостатньої зручності супроводу ПЗ є надзвичайно актуальною і що вирішення цієї проблеми полягає у створенні нового вдосконаленого методу до розроблення ПЗ на підставі наявних з акцентом на особливі вимоги безсерверної архітектури.
2. Проаналізовано сучасний стан підходів та методів до підвищення зручності супроводу ПЗ. Виявлено, що "чиста архітектура" успадковує основні переваги підходів DDD та TDD, а також надає більш чіткі настанови та охоплює ширший обсяг процесу розроблення ПЗ, тому вважається найкращою основою для покращеного методу. Сформульовано вимоги до вдосконаленого методу.
3. Розроблено вдосконалений метод до підвищення зручності супроводу ПЗ за умов застосування безсерверної архітектури, що задовольняє всі раніше сформульовані вимоги.

Було розроблено реалізації як на підставі чистої архітектури, так і вдосконаленого методу, а також наведено приклади основних покращень. Для даних реалізацій обраховані та проаналізовані метрики, а саме підвищення показника зручності супроводу ПЗ на 2,1 % та відфільтрованого показника зручності супроводу ПЗ на 8,3 % для вдосконаленого методу. Отже, отримані результати підтверджують ефективність створеного методу.

References

[1] ISO25010/57. (2022). Maintainability. https://iso25000.com/index.php/en/iso-25000-standards/iso-25010/57-maintainability

[2] Alenezi, M., & Zarour, M. (2020). On the relationship between software complexity and security. International Journal of Software Engineering & Applications (IJSEA), 11(1), 51–60. https://doi.org/10.5121/ijsea.2020.11104

[3] Abeyrathne, T. K. L., Gayal, B. A. I., Jayathilake, R. D. C. K., Weththasinghe, W. V. S. A. (2021). Increase Maintainability of a Software Product using Structuredness. University of Kelaniya. https://www.researchgate.net/publication/354363138_Increase_Maintainability_of_a_Software_Product_using_Structuredness

[4] Hasselbring, W. (2018). Software Architecture: Past, Present, Future. Gruhn, V., Striemer, R. (eds.). The Essence of Software Engineering. Springer, Cham. https://doi.org/10.1007/978-3-319-73897-0_10

[5] Lakhai, V., Kuzmych, O., Seniv, M. (2022). An improved approach to the development of software with increased requirements for flexibility and reliability in terms of creating small and medium-sized projects. IEEE 17th International Conference on Computer Sciences and Information Technologies (CSIT). Lviv. https://doi.org/10.1109/csit56902.2022.10000787

[6] Bondyopadhyay, A., & Dr. Mandal, A. C. (2022). Improvement of Software Reliability using Data Mining Technique. International Journal of Scientific and Research Publications, 12(6), 183–187. http://dx.doi.org/10.29322/IJSRP.12.06.2022.p126XX

[7] Anwer, F., Aftab, S., Waheed, U., Muhammad, S. S. (2017). Agile Software Development Models TDD, FDD, DSDM, and Crystal Methods: A Survey. International Journal of Multidisciplinary Sciences and Engineering, 246–270. https://www.researchgate.net/publication/316273992_Agile_Software_Development_Models_TDD_FDD_DSDM_and_Crystal_Methods_A_Survey

[8] Landre, E., Wesenberg, H., Olmheim, J. (2007). Agile enterprise software development using domain-driven design and test first. 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2007. https://doi.org/10.1145/1297846.1297967

[9] Uludag, O., Hauder, M., Kleehaus, M., Schimpfle C., Matthes, F. (2018). Supporting Large-Scale Agile Development with Domain-Driven Design. International Conference on Agile Software Development, 232–247. http://dx.doi.org/10.1007/978-3-319-91602-6_16

[10] Martin, R. C. (2017). Clean Architecture: A Craftsman's Guide To Software Structure And Design, London: Pearson [in England].

[11] Radon. (2020). Code Metrics. https://radon.readthedocs.io/en/latest/intro.html

V. Y. Lakhai, O. M. Kuzmych, M. M. Seniv

Lviv Polytechnic National University, Lviv, Ukraine

AN IMPROVED METHOD FOR INCREASING MAINTAINABILITY IN TERMS OF SERVERLESS ARCHITECTURE APPLICATION

It has been found that achieving high quality indicators, in particular maintainability [1], is becoming an increasing problem due to the emergence of new technologies, in particular cloud technologies and serverless architecture. Despite this, requirements for the level of ease of support are only growing, and this characteristic of software quality is receiving more and more attention. The numerous tools, methodologies, approaches and methods of software development previously developed by the scientific and engineering communities for its enhancement are insufficiently effective un-

der the existing conditions. It was confirmed that the use of an improved method to increase the convenience of maintenance under the conditions of using a serverless architecture is an urgent necessity. The current state of approaches to increasing the convenience of support was analyzed. As a result, Clean Architecture was chosen as the best existing approach because it inherits the main advantages of competing approaches, provides clearer instructions, and covers a wider range of the development process. Although it provides significant improvements in key maintainability characteristics such as modularity and reusability, there are still other characteristics that need to be considered, such as analyzability. modifiability and testability. The requirements for the improved method were determined and an improved method was developed to increase the convenience of support in the conditions of using a serverless architecture, which satisfies all the previously formulated requirements. In particular, this approach improves work with dependencies, provides a basic structure for components, increases cohesion and reduces coupling of functionality. To evaluate the effectiveness of the created method, two implementations of the same project were developed: based on Clean Architecture and using the improved method. The following maintainability assessment metrics were calculated for them: the Maintainability Index (MI) and its variation, in which only files with an index of less than 100 are taken into account, which we will call the Filtered Maintainability Index FMI. Analyzing the obtained results, it was found that the application of the improved method increases the value of the first metric by 2.1 % and the second by 8.3 %. In this way, the effectiveness of the developed improved method was proven.

Keywords: clean architecture, dependencies inversion, dependency injection, serverless.

Інформація про авторів:

Лахай Владислав Ярославович, здобувач, кафедра програмного забезпечення.
Email: vlad.luckhi@gmail.com; https://orcid.org/0000-0003-1346-5731

Кузьмич Олег Миколайович, здобувач, кафедра програмного забезпечення.
Email: oleg.kuzmuch@gmail.com; https://orcid.org/0000-0002-9749-6385

Сенів Максим Михайлович, канд. техн. наук, доцент, кафедра програмного забезпечення.
Email: maksym.m.seniv@lpnu.ua; https://orcid.org/0000-0003-1044-4628

Цитування за ДСТУ: Лахай В. Я., Кузьмич О. М., Сенів М. М. Удосконалений метод підвищення зручності супроводу за умов застосування безсерверної архітектури. *Український журнал інформаційних технологій.* 2023. Т. 5, № 1. С. 09–16.

Citation APA: Lakhai, V. Y., Kuzmych, O. M., Seniv, M. M. (2023). An improved method for increasing maintainability in terms of serverless architecture application. *Ukrainian Journal of Information Technology*, 5(1), 09–16. https://doi.org/10.23939/ujit2023.01.009