

Міністерство освіти і науки України
Державний заклад
«Луганський національний університет імені Тараса Шевченка»

Навчально-науковий інститут математики та інформаційних технологій

Кафедра інформаційних технологій та систем

Васильченко Станіслав Вячеславович

Вплив парадигм функціонального програмування на проєктування та
архітектуру сучасного програмного забезпечення

кваліфікаційна робота

здобувача вищої освіти другого (магістерського) рівня

освітньої програми «Мультимедійні системи»

за спеціальністю F2 «Інженерія програмного забезпечення»

Особистий підпис



Науковий керівник



Микола СЕМЕНОВ,
кандидат педагогічних наук, доцент
кафедри інформаційних технологій
та систем

Завідувач кафедри



Микола СЕМЕНОВ,
кандидат педагогічних наук, доцент
кафедри інформаційних технологій
та систем

Полтава – 2025

АНОТАЦІЯ

Тема: Вплив парадигм функціонального програмування на проєктування та архітектуру сучасного програмного забезпечення.

Спеціальність: F2«Інженерія програмного забезпечення».

Установа: ЛНУ імені Тараса Шевченка, 2026 р.

Магістерська робота містить: 95 с., 14 рис., 6 табл., 35 джерел.

Об'єкт дослідження – процеси проєктування та архітектурного формування сучасного програмного забезпечення.

Предмет дослідження – вплив парадигм функціонального програмування на архітектуру та проєктні рішення під час розроблення сучасного програмного забезпечення.

Мета дослідження – визначення закономірностей і механізмів впливу парадигм функціонального програмування на принципи проєктування та побудову архітектури сучасного програмного забезпечення.

Результати дослідження – було запропоновано концептуальну модель застосування функціональних принципів під час побудови архітектури, яка пояснює, як окремі елементи функціонального стилю змінюють структуру системи, механізми обробки даних та організацію модулів.

Ключові слова: ФУНКЦІОНАЛЬНЕ ПРОГРАМУВАННЯ, ПРОЄКТУВАННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ, АРХІТЕКТУРА ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ, ФУНКЦІОНАЛЬНА АРХІТЕКТУРА ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ, MVU

ABSTRACT

Topic: The Influence of Functional Programming Paradigms on the Design and Architecture of Modern Software.

Specialty: F2 "Software Engineering".

Institution: Taras Shevchenko National University of Luhansk, 2026.

Master's thesis contains: 95 pages, 14 figures, 6 tables, 35 sources.

Object of Study: The processes of design and architectural formation of modern software.

Subject of Study: The influence of functional programming paradigms on the architecture and design decisions during the development of modern software.

Result of Study: To determine the patterns and mechanisms of the influence of functional programming paradigms on the design principles and the construction of the architecture of modern software.

Research Results: A conceptual model for applying functional principles during architecture construction was proposed, which explains how individual elements of the functional style change the system's structure, data processing mechanisms, and module organization.

Key Words: FUNCTIONAL PROGRAMMING, SOFTWARE DESIGN, SOFTWARE ARCHITECTURE, FUNCTIONAL SOFTWARE ARCHITECTURE, MVU

ЗМІСТ

ВСТУП	4
РОЗДІЛ 1. ТЕОРЕТИЧНІ ТА МЕТОДОЛОГІЧНІ ОСНОВИ ФУНКЦІОНАЛЬНОГО ПРОГРАМУВАННЯ	10
1.1. Еволюція парадигм програмування та роль функціонального підходу	10
1.2. Ключові принципи та концепції функціонального програмування	16
1.3. Вплив концепцій функціонального програмування на критичні вимоги до програмного забезпечення	24
1.4. Висновки за розділом 1	32
РОЗДІЛ 2. АРХІТЕКТУРНИЙ АНАЛІЗ ЗАСТОСУВАННЯ ФУНКЦІОНАЛЬНИХ ПАРАДИГМ	33
2.1. Імперативний, об'єктно-орієнтований та функціональний підходи: їхні концептуальні відмінності	33
2.2. Інтеграція функціональних концепцій у гібридні мови програмування	41
2.3. Вплив парадигми функціонального програмування на сучасні архітектурні шаблони.	46
Висновки до розділу 2	56
РОЗДІЛ 3. РОЗРОБКА РЕКОМЕНДАЦІЙ ТА АПРОБАЦІЯ ЕФЕКТИВНОСТІ ВПРОВАДЖЕННЯ ФУНКЦІОНАЛЬНОГО СТИЛЮ	58
3.1. Критерії та метрики оцінки архітектурної якості функціонального програмного забезпечення	58
3.2. Розробка рекомендацій щодо доцільного впровадження функціонального програмування	64
3.3. Апробація запропонованих рекомендацій та оцінка результатів	66
Висновки до розділу 3	69
ВИСНОВКИ	71
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ	75
Додаток А	79
Додаток Б	82
Додаток В	84

ВСТУП

Посилення вимог до якості, надійності й масштабованості сучасних інформаційних систем підвищує значення функціонального програмування в інженерії програмного забезпечення. Упродовж останніх років функціональні парадигми дедалі частіше інтегруються у популярні мови програмування та практики проєктування, що свідчить про їхню практичну цінність і потребу в глибшому науковому аналізі. Незважаючи на історично академічний статус функціонального підходу, нині він формує нові тенденції у створенні архітектур, особливо в системах зі складною логікою, вимогами до безпечної паралельності та високим навантаженням. Обрана тема є актуальною, оскільки дозволяє дослідити, як концепції, що ще донедавна вважалися обмеженими, модифікують прийоми проєктування та є фундаментом для нових архітектурних стилів.

Проблематика функціонального програмування активно розроблялася ще у середині XX століття в роботах дослідників, які закладали основи формальних моделей обчислень. Значний внесок зробив Аланзо Чьорч (Alonzo Church), який створив λ -числення як теоретичну основу для абстракцій функцій і обчислень без змінного стану [7]. Подальший розвиток функціональної парадигми забезпечили Джон Маккарті (John McCarthy), автор мови LISP [23; 24], Пітер Лендін (Peter J. Landin), який сформулював поняття семантики мов програмування та «функціональний стиль» [19], а також Робін Мілнер (Arthur John Robin Gorell Milner), що створив мову ML та вніс фундаментальний вклад у типові системи й формальні методи [26]. У сучасну епоху функціональні ідеї активно популяризували Ерік Мейєр (Erik Meijer) [25], Саймон Пейтон Джонс (Simon Peyton Jones), Кріс Окезан (Chris Okasaki) [29] та автори екосистеми мов Haskell, Scala і Clojure.

Проєктування та архітектуру програмного забезпечення досліджували насамперед автори, пов'язані з розвитком структурного, об'єктно-орієнтованого та компонентного підходів. Значний вплив на формування

архітектурних концепцій справили Фредерік Брукс (Frederick P. Brooks) [6], Девід Парнас (David L. Parnas) [30; 31], Баррі Бем (Barry Boehm), Едсгер Дейкстра (Edsger W. Dijkstra) [12], Том ДеМарко (Tom DeMarco) [11], Івар Якобсон (Ivar Jacobson), Гленфорд Майерс (Glenford Myers) [27].

Подальший розвиток архітектурного проектування та значний практичний внесок забезпечили Мартін Фаулер (Martin Fowler) Лен Басс (Len Bass), Пол Клементс (Paul Clements), Рік Казман (Rick Kazman). Роберт Мартін (Robert C. Martin, “Uncle Bob”) – автор концепцій «чистої архітектури» та принципів SOLID [21; 22]. Їхні праці сформували широку методологічну базу для побудови масштабованих і підтримуваних програмних систем.

Одночасно проблеми функціонального підходу та архітектурного проектування поєднували дослідники, які намагалися інтегрувати функціональні концепції у практики інженерії програмного забезпечення. Саймон Пейтон Джонс (Simon Peyton Jones) і його колеги аналізували застосування функціональних моделей у великих системах та побудові паралельних архітектур. Пол Чіусано (Paul Chiusano) та Рунар Б'ярнасон (Runar Bjarnason) у роботах про Scala й концепції «функціонального проектування» описали, як функціональні абстракції впливають на архітектурні структури. Брюс Тейт (Bruce Tate) і Річ Хікі (Rich Hickey) у контексті Clojure досліджували, як імутабельність і функціональний стиль змінюють підходи до керування станом та модульності систем.

Завдяки цим авторам з'явилося міждисциплінарне бачення, яке поєднує функціональні принципи та архітектурні практики сучасної розробки. На основі аналізу існуючих досліджень констатуємо недостатню визначеність впливу функціональних парадигм на сучасні методи інженерії програмного забезпечення. Хоча в багатьох прикладних мовах активно використовуються ідеї чистих функцій, імутабельності даних, функцій вищого порядку чи декларативного опису поведінки, розуміння того, як ці концепти трансформують архітектурні рішення і підходи до побудови складних систем, залишається фрагментарним. Часто розробники впроваджують окремі

елементи функціональності інтуїтивно, без усвідомлення їхнього системного впливу на архітектуру застосунку. Це створює методологічний розрив між теоретичними перевагами функціонального підходу та реальними архітектурними моделями, що потребує додаткового дослідження. Це обумовило вибір теми магістерської роботи.

Об'єкт дослідження – процеси проєктування та архітектурного формування сучасного програмного забезпечення.

Предмет дослідження – вплив парадигм функціонального програмування на архітектуру та проєктні рішення під час розроблення сучасного програмного забезпечення.

Мета дослідження – визначення закономірностей і механізмів впливу парадигм функціонального програмування на принципи проєктування та побудову архітектури сучасного програмного забезпечення.

Основна увага зосереджувалася на виявленні того, як функціональний стиль змінює процес прийняття архітектурних рішень, сприяє підвищенню надійності й передбачуваності програмних систем, а також підсилює можливості масштабування та підтримуваності коду.

Для досягнення мети сформульовано такі **завдання**:

- 1) проаналізувати сучасний стан функціонального програмування, окреслити ключові принципи та властивості цієї парадигми й дослідити їхній зв'язок із вимогами, що висуваються до сучасного програмного забезпечення;
- 2) порівняння функціонального та імперативного/ об'єктно-орієнтованого підходів у контексті архітектурного проєктування, включаючи аспекти керування станом, організації модулів та забезпечення паралельності.
- 3) дослідити практичні приклади використання функціональних концепцій у сучасних мовах програмування, зокрема у гібридних підходах, а також виявити їхній вплив на архітектурні шаблони.

- 4) розробити формалізовані критерії та метрики для оцінки архітектурної якості (зв'язність, зчеплення, зрозумілість) програмного забезпечення, що розроблене із застосуванням принципів функціонального програмування.
- 5) розробити рекомендації щодо ефективного впровадження функціонального стилю у процес інженерії програмного забезпечення з урахуванням специфіки різних проєктних контекстів.
- 6) апробувати запропоновані рекомендації на фрагменті реального програмного проєкту (навчальному прикладі) та оцінити отриманий практичний ефект щодо поліпшення масштабованості та підтримуваності коду.

Наукова новизна магістерської роботи полягає у систематизації та узагальненні закономірностей впливу функціонального програмування на архітектурні та проєктні рішення в інженерії програмного забезпечення, що раніше були реалізовані переважно фрагментарно. У дослідженні було запропоновано концептуальну модель застосування функціональних принципів під час побудови архітектури, яка пояснює, як окремі елементи функціонального стилю змінюють структуру системи, механізми обробки даних та організацію модулів. Новизна також проявляється у виявленні переваг і ризиків інтеграції функціональних підходів у гібридні архітектури, що широко використовуються у промисловій розробці, та у формулюванні практичних рекомендацій щодо оптимізації архітектур із урахуванням властивостей функціонального стилю.

Методи дослідження: аналіз і синтез наукових джерел, порівняльний аналіз різних парадигм програмування, системний підхід до оцінювання архітектурних моделей, а також методи індукції та дедукції для узагальнення властивостей функціональних концепцій у контексті їхнього впливу на проєктування систем. Крім того, використовувалися елементи моделювання для побудови концептуальної архітектурної схеми, що враховує використання функціональних принципів, та методи практичного аналізу, спрямовані на

розгляд реальних прикладів впровадження функціонального стилю в сучасних мовах програмування.

У першому розділі здійснено огляд еволюції парадигм програмування та визначено роль функціонального підходу в сучасних системах. Розглянуто ключові принципи та концепції функціональної парадигми, а також проаналізовано їхній вплив на критичні вимоги до програмного забезпечення.

У другому розділі описано архітектурний аналіз застосування функціональних концепцій. Розглянуто концептуальні відмінності між імперативним, об'єктно-орієнтованим та функціональним підходами, досліджено інтеграцію функціональних принципів у гібридні мови програмування та їхній вплив на сучасні архітектурні шаблони.

У третьому розділі проведено розробку рекомендацій щодо доцільного впровадження функціонального стилю програмування. Представлено критерії та метрики оцінки архітектурної якості програмного забезпечення, виконано апробацію запропонованих рекомендацій та здійснено оцінку отриманих результатів.

РОЗДІЛ 1. ТЕОРЕТИЧНІ ТА МЕТОДОЛОГІЧНІ ОСНОВИ ФУНКЦІОНАЛЬНОГО ПРОГРАМУВАННЯ

1.1. Еволюція парадигм програмування та роль функціонального підходу

Парадигми програмування являють собою фундаментальні філософські підходи та набори концепцій, які визначають спосіб структурування коду, вирішення обчислювальних завдань та організацію взаємодії компонентів програмного забезпечення (ПЗ). Історичний розвиток програмування є процесом постійної абстракції та пошуку методів управління зростаючою складністю систем.

Еволюцію парадигм можна умовно розділити на кілька ключових етапів.

Імперативні та процедурні парадигми (1950-1970-ті). Цей початковий етап характеризувався домінуванням імперативного стилю, де програміст чітко вказує комп'ютеру послідовність інструкцій, які змінюють глобальний стан програми.

Процедурне програмування (ПП): (FORTRAN, C, Pascal) впровадило концепцію підпрограм (функцій/процедур) для організації коду та уникнення дублювання, що стало першим кроком до структурного управління. Однак, ПП має обмеження у великих системах через тісний зв'язок між даними та процедурами, які їх обробляють, а також складність контролю мутабельного (змінного) стану.

Об'єктно-орієнтована парадигма (1980-ті – по сьогодні). Об'єктно-орієнтоване програмування (ООП): (Smalltalk, C++, Java, C) виникло як відповідь на необхідність кращого управління складністю через моделювання реального світу. Ключові принципи: інкапсуляція (поєднання даних та методів), успадкування (ієрархія класів) та поліморфізм (здатність об'єктів приймати різні форми).

ООП стало домінуючою парадигмою для проєктування великих корпоративних та графічних систем, забезпечуючи високу повторюваність коду та модульність.

Декларативні парадигми та відродження функціонального програмування (1990-ті – по сьогодні). Цей етап фокусується не на тому, як виконувати обчислення (послідовність кроків), а на тому, що потрібно обчислити (бажаний результат).

Функціональне програмування (ФП), що базується на лямбда-численні (розроблене А. Чорчем у 1930-х), є класичною декларативною парадигмою. Хоча воно існувало давно (Lisp, Haskell), його сучасне відродження безпосередньо пов'язане з потребами архітектури сучасного ПЗ.

Основні причини посилення ролі функціонального програмування:

1. Проблема багатопотоковості та паралелізму: З поширенням багатоядерних процесорів виникла потреба ефективного використання паралельних обчислень. Концепція незмінності даних (Immutability) та чистих функцій виключає побічні ефекти та гонки даних (race conditions), спрощуючи безпечне виконання коду на паралельних потоках.

2. Управління станом (State Management): У великих розподілених системах (особливо у фронтенді та базах даних) керування змінним станом стало основним джерелом помилок та непередбачуваності. Використання чистих функцій гарантує, що при однакових вхідних даних завжди буде однаковий результат, роблячи поведінку системи передбачуваною та легко верифікованою.

3. Підвищення рівня абстракції та модульності: функціональне програмування сприяє створенню висококомпонованого коду. Використання функцій як членів першого класу та композиції функцій дозволяє створювати складну логіку шляхом комбінування простих, незалежних, перевірених компонентів.

Сьогодні функціональне програмування рідко використовується у "чистому" вигляді (крім Haskell або Erlang). Його найбільша роль полягає у трансформації гібридних мов (Scala, Kotlin, JavaScript, C, Python) та визначенні сучасних архітектурних шаблонів:

- Реактивне програмування (Reactive Programming): Базується на незмінності та потоках даних.
- Архітектура, керована подіями (Event Sourcing): Використовує ідею незмінної послідовності подій (Immutable Log).
- Front-end фреймворки (наприклад, Redux): Принцип єдиного, незмінного стану (Single Immutable State Tree) походить безпосередньо з функціонального програмування.

Сучасний стан розвитку функціонального програмування характеризується суттєвим розширенням його впливу на методи та практики створення програмного забезпечення. Хоча функціональна парадигма має тривалу історію, упродовж останніх двох десятиліть вона вийшла за межі суто академічної сфери та стала одним із ключових чинників еволюції підходів до програмування й архітектурного проектування. Зростання масштабів програмних систем, потреба в ефективній обробці великих обсягів даних і забезпеченні безпечної паралельності сприяли переосмисленню ролі функціонального підходу. Багато концепцій, що донедавна вважалися специфічними або нішевими, сьогодні інтегруються в популярні промислові мови та прикладні фреймворки.

Однією з ознак сучасного етапу розвитку функціонального програмування є його поширення у гібридних мовах загального призначення. Мови Scala, Kotlin, JavaScript, Python, а також нові версії Java та C# включають механізми, що дозволяють програмістам використовувати функції вищого порядку, лямбда-вирази, імутабельні структури даних та декларативні моделі обробки інформації. Це свідчить про поступове зближення парадигм та утвердження функціональних підходів як основного інструмента для побудови надійних і передбачуваних програмних систем. У той же час спеціалізовані функціональні мови, такі як Haskell, F#, OCaml та Erlang/Elixir, зберігають важливу роль у розробці високонадійних систем, застосовуваних у фінансових платформах, телекомунікаційних рішеннях та інфраструктурі розподілених обчислень.

Суттєвий вплив на розвиток функціонального програмування справляють зміни в апаратних архітектурах. Перехід до багатоядерних процесорів і масового використання розподілених систем висуває нові вимоги до моделей обчислень. Традиційний імперативний підхід, орієнтований на змінний стан і послідовне виконання, виявляється менш ефективним за умов високого рівня паралельності. Натомість функціональний стиль - завдяки імутабельності, чистим функціям і можливості природного розпаралелювання - демонструє конкурентні переваги, що стимулює його інтеграцію у фреймворки для реактивного програмування, обробки поточкових даних та побудови fault-tolerant систем.

Окремою тенденцією є зміцнення позицій функціонального підходу в галузі аналізу даних і машинного навчання. Популярні інструменти обробки даних, зокрема Apache Spark, побудовані на концепціях функцій вищого порядку та імутабельних структур. Це забезпечує передбачуваність виконання та спрощує оптимізацію обчислювальних процесів у розподіленому середовищі. У наукових підходах до моделювання складних систем також спостерігається активне застосування функціональних абстракцій через їхню формальну строгість і можливість математичного доведення властивостей програм.

Ще однією важливою характеристикою сучасного стану розвитку функціонального програмування є зміна парадигми в освітній сфері. Функціональний підхід дедалі частіше включають до навчальних програм провідних університетів як елемент базової інженерної підготовки. Це зумовлено потребою у підготовці фахівців, здатних працювати з декларативними моделями, складними типами даних та формальними методами валідації програмних рішень. Водночас зростає кількість прикладних курсів, що орієнтовані на поєднання функціонального програмування з промисловими технологіями, такими як мікросервіси, хмарні платформи та засоби побудови реактивних систем.

Таким чином, сучасний стан функціонального програмування характеризується інтеграцією його концепцій у широкий спектр мов і технологій, зростанням практичної значущості в умовах паралельних та розподілених обчислень і поступовим формуванням нового рівня архітектурної культури. Функціональний підхід уже не виступає альтернативою традиційним парадигмам, а стає їхнім доповненням і каталізатором розвитку методів інженерії програмного забезпечення.

Наприкінці параграфа приведемо приклади (Листинг 1.1 – імперативний підхід, Листинг 1.2 – функціональний підхід), які демонструють при функціональному підході чисті функції та незмінність даних (Immutability) – для підвищення надійності та безпеки паралельності. А також забезпечують відсутність побічних ефектів – для передбачуваності та тестованості.

Листинг 1.1

Імперативний підхід

```
using System;
using System.Collections.Generic;

public class Employee
{
    public string Name { get; set; }
    public int Salary { get; set; }
}

public class ImperativeExample
{
    public static List<string> FilterAndFormat(List<Employee> employees, int
minSalary)
    {
        // Мутабельний список для збору результатів
        List<string> result = new List<string>();

        foreach (var emp in employees)
        {
            // Фільтрація
            if (emp.Salary >= minSalary)
            {
                // Трансформація
                result.Add($"{emp.Name.ToUpper()} ({emp.Salary})");
            }
        }
        return result;
    }
}
```

Функціональний підхід

```
using System.Linq;
// ... клас Employee залишається той самий

public class FunctionalExample
{
    // Використовуємо IReadOnlyList<T> для підкреслення незмінності
    public static IReadOnlyList<string> FilterAndFormat(IReadOnlyList<Employee>
employees, int minSalary)
    {
        // Функціональний підхід: Композиція
        var result = employees
            .Where(e => e.Salary >= minSalary)    // 1. Фільтрація (Чиста функція)
            .Select(e => $"{e.Name.ToUpper()} ({e.Salary})") // 2. Трансформація
            (Чиста функція)
            .ToList(); // 3. Збір результату (створення нової, незмінної колекції)

        return result;
    }
}

// Приклад використання
// var employees = new List<Employee> { ... };
// var highEarnings = FunctionalExample.FilterAndFormat(employees, 50000);
```

У Листингу 1.2 приведено функціональний підхід (композиція функцій за допомогою LINQ). Використано композицію трьох чистих функціональних операцій (Where, Select, ToList), які виконуються послідовно, не змінюючи вихідну колекцію. Код стає декларативним (описує, що робити, а не як), лаконічним та легко читається (послідовність операцій). Отже, функціональні концепції (чистота та композиція) поліпшують проектування архітектури даних. Функціональний стиль зміщує фокус з "що змінюється" на "що обчислюється", що є ключовим для створення надійних, масштабованих і легко підтримуваних архітектур сучасного ПЗ.

1.2. Ключові принципи та концепції функціонального програмування

Функціональне програмування є декларативною парадигмою, яка базується на ідеї побудови програм як композиції чистих математичних функцій, уникаючи мутабельного стану та побічних ефектів. Фундаментальні основи FP були закладені Алонзо Черчем у 1930-х роках через розробку λ -числення, яке стало формальним обчислювальним базисом, що передувало сучасним комп'ютерам [7]. Сучасний науковий інтерес до функціонального програмування був відроджений Джоном Бекусом у його знаковій праці [2] (1978), де він критикував імперативний стиль і запропонував функціональне програмування як більш абстрактну та композиційну альтернативу.

Опишемо це математично. λ -числення є мінімалістичною формальною системою, що служить теоретичною основою для функціональних мов. Вона описує обчислення через застосування та абстракцію функцій.

λ -вираз e визначається рекурсивною граматикою (1.1), що містить лише три форми:

$$e ::= x \mid (\lambda x. e) \mid (e_1 e_2), \quad (1.1)$$

де:

- **змінна (x):** Ідентифікатор.
- **λ -абстракція ($\lambda x. e$):** Визначення анонімної функції, де x є параметром, а e – тілом функції.
- **застосування Функції ($e_1 e_2$):** Виклик функції e_1 з аргументом e_2 .

Обчислення в λ -численні зводиться до **β -редукції** (1.2), яка є правилом заміщення, еквівалентним виклику функції:

$$(\lambda x. e_1) e_2 \rightarrow [e_2/x] e_1, \quad (1.2)$$

де $[e_2/x] e_1$ означає тіло функції e_1 після заміни всіх вільних входжень змінної x на аргумент e_2 .

Приклад 1 (функція ідентичності):

$$id = (\lambda x. x)$$
$$(\lambda x. x) y \rightarrow [y/x]x = y$$

Приклад 2 (функція додавання двох):

$$add = (\lambda x. \lambda y. x + y)$$
$$add\ 5\ implies(\lambda y. 5 + y)$$
$$add\ 5\ 3\ implies\ 5 + 3 = 8$$

Ключові ідеї λ -числення:

- функції як основні об'єкти (у λ -численні все представлено у вигляді функцій, навіть дані можна моделювати як функції);
- анонімні функції (функції не потребують імен, що дозволяє створювати компактні вирази), застосування функцій (обчислення виконується шляхом застосування функції до аргументу);
- редукція (основна операція - підстановка аргументу в тіло функції для отримання результату);
- абстракція та композиція (можна створювати функції, які приймають інші функції, що є основою концепції функцій вищого порядку).

Лямбда-числення забезпечує чистоту обчислень, без побічних ефектів. Воно передбачає незмінність даних, що робить програми більш передбачуваними. Підтримує функції вищого порядку, які широко використовуються в сучасних мовах програмування. Розширення лямбда-числення з типами стало основою для систем типів у таких мовах, як Haskell та OCaml.

Першим фундаментальним принципом функціонального програмування є функції як об'єкти першого класу (first-class functions). Це означає, що функції можуть передаватися як аргументи іншим функціям, повертатися як результати, присвоюватися змінним чи включатися до структур даних [26].

Завдяки цьому з'являються функції вищого порядку (higher-order functions), що приймають або повертають функції, - це дає високу абстракцію, модульність та можливість комбінування функціональних блоків [27].

Таким чином, програмна логіка формується не як послідовність команд, а як композиція функцій, що значно підвищує гнучкість і повторне використання коду.

Однією з потужних технік, яка впливає з концепції функцій першого класу, є карринг (англ. currying), названий на честь математика Гаскелла Каррі (Haskell Curry).

Карринг - це процес перетворення функції (1.3), яка приймає кілька аргументів (наприклад, $f(a, b, c)$), на послідовність функцій, кожна з яких приймає лише один аргумент і повертає нову функцію, доки не будуть надані всі необхідні аргументи. Тобто, $f(a, b, c)$ перетворюється на $f(a)(b)(c)$ [8].

$$f(a, b) \rightarrow g(a) \rightarrow g_a(b) \quad (1.3)$$

Важливість каррингу для архітектури функціонального програмування:

1. Створення спеціалізованих функцій (Partial Application): карринг дозволяє легко реалізувати часткове застосування (partial application). Це процес фіксації одного або кількох перших аргументів функції, створюючи нову, більш спеціалізовану функцію.

2. Підвищення Композиційності: спеціалізовані функції, отримані через часткове застосування, стають ідеальними будівельними блоками для композиції. Вони мають фіксований, передбачуваний інтерфейс (зазвичай, приймають один аргумент), що значно спрощує їх послідовне об'єднання.

3. Спрощення впровадження HOF: карринг природно інтегрується з функціями вищих порядків (HOF), оскільки багато HOF (таких як *map* або *filter*) очікують функцію, яка приймає один елемент колекції. Часткове застосування дозволяє легко адаптувати складнішу бізнес-логіку до цього інтерфейсу.

Наведемо приклад (концептуальний):

Якщо існує функція *calculateDiscount(rate, price)*, карринг дозволяє створити спеціалізовану функцію *applyTaxRate* (1.4), зафіксувавши ставку:

$$\text{applyTaxRate} = \text{calculateDiscount}(\text{rate} = 0.15) \quad (1.4)$$

Нова функція *applyTaxRate(price)* тепер є чистою функцією, готовою до композиції з іншими функціями обробки цін. Це забезпечує гнучкість, повторне використання та чітке відокремлення даних від логіки.

Важливим принципом є чисті функції (pure functions) та відсутність побічних ефектів (no side effects). Чиста функція повертає результат лише на основі своїх вхідних аргументів і не змінює стан зовнішнього середовища (наприклад, глобальних змінних, файлів, баз даних) та не залежить від зовнішнього стану [26].

Детермінованість (determinism) - за тих же аргументів функція завжди поверне однаковий результат - сприяє передбачуваності, тестованості та формальній верифікації коду.

Побічні ефекти ж часто призводять до ускладнень в модульності, паралельності та відлагодженні, тому уникнення їх є ключем до високоякісної архітектури програмного забезпечення.

Третім критичним принципом є імітабельність даних (immutability) та некоректування стану (no mutable state). У ФП дані після створення не змінюються - замість цього створюються нові структури за потреби.

Імутабельність спрощує розуміння програми, оскільки змін стану, який потрібно відстежувати, немає. Це особливо суттєво в умовах багатопотоковості та паралелізму: відсутність змінного стану знижує ризик гонок даних (data races) і дає можливість безпечного розпаралелювання.

Четвертим важливим елементом є декларативний стиль програмування (declarative programming) замість імперативного. У функціональному

програмуванні програміст описує що треба зробити (наприклад, трансформацію даних) замість як це виконати крок за кроком.

Такий підхід підвищує читабельність, відокремлює логіку від побічних ефектів та дає кращу основу для формальної оптимізації. У поєднанні з композицією функцій це дозволяє створювати складні обчислення через ланцюжки функціональних перетворень.

П'ятим концептуальним елементом є композиція функцій (function composition) і рекурсія (recursion) замість традиційних циклів. Композиція дозволяє комбінувати прості функції в більш складні, забезпечуючи модульність і повторне використання.

Рекурсія часто використовується там, де в імперативному стилі застосовувались цикли, і вимагає належної підтримки від мови програмування (наприклад, хвостова рекурсія). У функціональному програмуванні композиція і рекурсія працюють разом як ключові абстракції обчислень.

Крім цих основних принципів, функціональне програмування включає такі концепції, як ледаче (відкладене) обчислення (lazy evaluation) - коли обчислення відкладається до моменту використання результату - що дозволяє працювати з потенційно нескінченними структурами даних та оптимізувати виконання.

Також важливим є поняття монад (monads) - абстракцій, що дозволяють узгоджено управляти побічними ефектами, вводити обчислювальні контексти (наприклад, обробка помилок, асинхронність) в чистому функціональному стилі.

Ці концепції відіграють ключову роль в архітектурі сучасних функціональних систем.

Підсумуємо наведені інформацію через призму архітектурної переваги функціонального підходу, яка походить від суворого дотримання кількох ключових принципів [4]:

1. **Чистота (Pure Functions)** є наріжним каменем функціонального програмування. Функція вважається чистою, якщо вона задовольняє дві умови:
2. **Детермінізм:** при однакових вхідних аргументах функція завжди повертає однаковий результат.
3. **Відсутність побічних ефектів (No Side Effects):** функція не змінює зовнішній стан системи, включаючи глобальні змінні, аргументи, не виконує операцій введення/виведення (I/O) чи мутацію об'єктів поза своїм локальним обчислювальним простором.

Функціональне програмування вимагає, щоб дані, після їх створення, **не могли бути змінені (Immutability)**. Якщо необхідна модифікація структури даних, створюється **нова копія** з внесеними змінами, а оригінальна структура залишається недоторканою. Цей принцип критично важливий для:

- **Спрощення паралелізму:** у багатопотоковому середовищі відсутність спільного мутабельного стану унеможливорює виникнення гонок даних (race conditions) та значно спрощує архітектуру конкурентних систем.
- **Підвищення надійності:** незмінність спрощує логічне міркування про стан програми (reasoning about state) та сприяє прозорості посилань (referential transparency).

У функціональному програмуванні функції розглядаються як будь-які інші значення (змінні). Це означає, що функція може бути:

- присвоєно змінній.
- передана як аргумент іншій функції (Функції Вищих Порядків).
- повернута як результат іншої функції (використання замикань, Closures).

Функції Вищих Порядків (Higher-Order Functions, HOF), такі як:

$\text{map}, \text{filter}, \text{reduce}(\text{fold}),$

є основними інструментами для композиції та абстракції над потоками даних, замінюючи традиційні імперативні цикли [1].

FP заохочує створення складної бізнес-логіки шляхом послідовного застосування простих чистих функцій: $h(x) = f(g(x))$. Цей підхід сприяє високій модульності та повторному використанню коду.

Для керування складністю та ізоляції необхідних побічних ефектів (наприклад, I/O, винятки, асинхронність), функціональне програмування використовує концепції, запозичені з **теорії категорій** [17]:

- **Функтори (Functors):** типи, які знають, як застосовувати функції до значень, "обгорнутих" у певний контекст (наприклад, *Option/ Maybe, List*).
- **Монади (Monads):** це потужніша абстракція, яка дозволяє **компонувати** обчислення, що мають побічні ефекти, **декларативним та безпечним способом**, водночас зберігаючи чистоту решти коду. Вони слугують архітектурним шаблоном для ізоляції "брудних" частин програми [Hutton, Meijer (LINQ)].

FP-концепції активно інтегруються в гібридні мови (Scala, C#, JavaScript) [Fogus, Meijer] і впливають на ключові архітектурні шаблони:

- **Реактивне програмування (Reactive Programming):** використовує функції вищих порядків та незмінні потоки даних (Observable streams).
- **LINQ (C#):** безпосередня реалізація HOF (*Select* \equiv *map*, *Where* \equiv *filter*) для декларативних запитів до даних.
- **Мікросервісна Архітектура:** чисті функції ідеально підходять для безстанових (stateless) мікросервісів, які легко масштабуються.

Узагальнюючи, ключові принципи функціонального програмування - функції першого класу, чистота функцій та відсутність побічних ефектів, імутабельність даних, декларативність, композиція і рекурсія - утворюють чітку парадигму, орієнтовану на підвищення прозорості, передбачуваності, підтримуваності та паралелізованості програмного забезпечення. Розуміння й правильне застосування цих принципів лежить в основі ефективного

використання функціонального стилю у сучасній інженерії програмного забезпечення.

1.3. Вплив концепцій функціонального програмування на критичні вимоги до програмного забезпечення

Функціональне програмування істотно впливає на задоволення критичних вимог до сучасного програмного забезпечення, зокрема надійності, безпеки, передбачуваності, масштабованості та зрозумілості коду. Центральне місце у цьому процесі займає принцип чистих функцій, згідно з яким результат обчислення визначається виключно переданими аргументами, а сама функція не змінює глобальний стан і не породжує побічних ефектів. Такий підхід значно спрощує процеси тестування, формальної верифікації та моделювання поведінки системи, оскільки кожна функція може розглядатися як математичне відображення. Детермінованість чистих функцій дає змогу точно прогнозувати результати обчислень за будь-яких умов, що зменшує кількість помилок і робить програмний код більш стійким до некоректної взаємодії компонентів.

Впровадження ключових концепцій функціонального програмування (FP) - чистоти функцій та незмінності даних (immutability) - суттєво підвищує відповідність програмного забезпечення (ПЗ) критичним архітектурним вимогам, таким як надійність, безпека та зрозумілість коду.

Надійність та верифікація через чистоту

Основний принцип чистоти функцій (відсутність побічних ефектів і детермінізм) прямо впливає на надійність програмної системи. Чиста функція залежить виключно від своїх вхідних аргументів і завжди повертає однаковий результат для однакових вхідних даних. Це забезпечує прозорість посилань (referential transparency): будь-який виклик чистої функції може бути замінений її результатом без зміни поведінки програми. Ця властивість значно спрощує верифікацію коду, оскільки тестування зводиться до перевірки відповідності вихідних даних вхідним, ізолюючи функцію від складності зовнішнього стану. Таким чином, чисті функції знижують ймовірність появи

непередбачуваних помилок, що є критичним для створення надійних систем [4].

Безпека та багатопотоковість завдяки незмінності

Вимога незмінності даних є фундаментальною для забезпечення безпеки в сучасних багатопотокових та розподілених системах. Принцип незмінності гарантує, що після створення об'єкта його стан не може бути змінений. Це автоматично усуває проблему гонок даних (race conditions), які виникають, коли два або більше потоків одночасно намагаються змінити спільну змінну. У функціональних архітектурах потоки можуть безпечно читати спільні дані, оскільки вони гарантовано не будуть мутовані. Це дозволяє створювати високопродуктивні, паралельні системи без необхідності використання складних і часто неефективних механізмів синхронізації (наприклад, блокувань locks), спрощуючи архітектуру конкурентного ПЗ.

Підвищення зрозумілості та підтримуваності

Дотримання чистоти та незмінності даних також сприяє зрозумілості та підтримуваності коду. Оскільки чисті функції не мають прихованих залежностей і побічних ефектів, програміст, читаючи код, може зосередитися виключно на логіці функції, не відстежуючи, як її виконання може змінити стан інших частин системи. Це полегшує рефакторинг і локалізацію помилок. Композиція невеликих, чистих, добре ізольованих функцій створює високомодульну структуру, де кожен компонент має чітко визначену відповідальність, що є основою для довготривалої підтримуваності складних програмних архітектур.

Схема на рис. 1.1 демонструє систематичний вплив ключових концепцій функціонального програмування (ФП) на відповідність програмного забезпечення (ПЗ) критичним архітектурним вимогам.

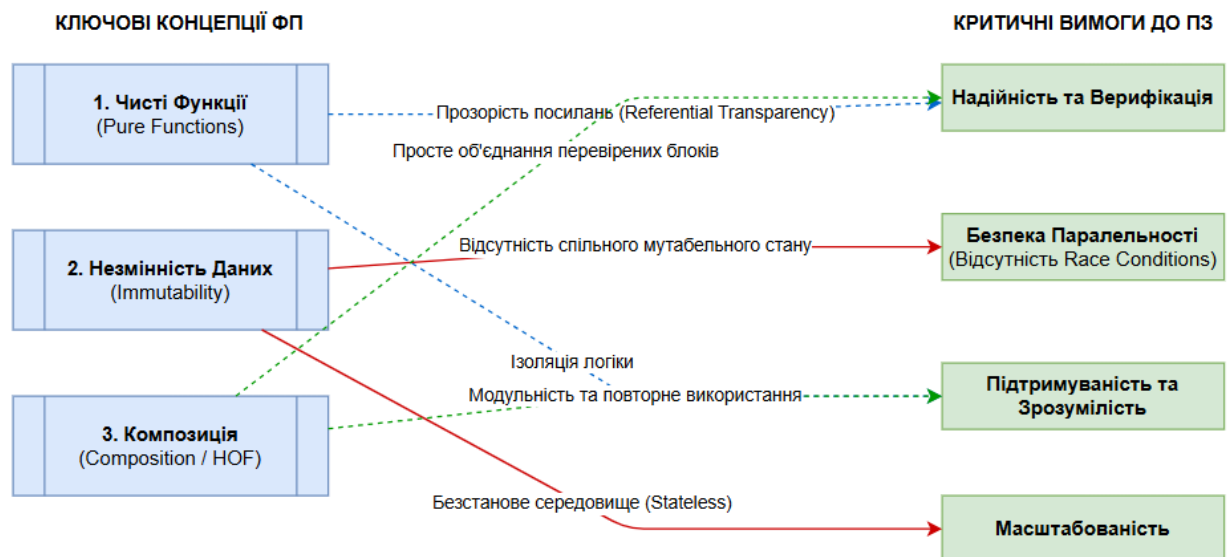


Рис. 1.1 Вплив ключових концепцій функціонального програмування на критичні вимоги до програмного забезпечення

Діаграма (рис 1.1) розділена на дві основні групи. Зліва розташовані «Ключові Концепції» функціонального програмування: «Чисті Функції» (Pure Functions), «Незмінність Даних» (Immutability) та «Композиція» (використання функцій вищих порядків). Ці концепції є вихідними принципами функціональної парадигми, які виступають каталізаторами для підвищення якості архітектури.

Справа розташовані «Критичні Вимоги до ПЗ», які є цільовими метриками якості: «Надійність та Верифікація», «Безпека Паралельності», «Підтримуваність та Зрозумілість», а також «Масштабованість». Зв'язки (стрілки) показують прямий причинно-наслідковий зв'язок між концепціями функціонального програмування та цими вимогами, забезпечуючи архітектурну перевагу функціонального стилю.

«Чисті функції» підвищують «Надійність» через «Прозорість» посилань, спрощуючи тестування та верифікацію. У поєднанні з «Композицією» (яка забезпечує «Модульність»), вони також суттєво поліпшують «Підтримуваність та Зрозумілість» коду, оскільки логіка стає ізольованою та легко об'єднується. «Незмінність Даних» є ключовою концепцією для сучасних багатопотокових систем. «Незмінність» гарантує «Безпеку

Паралельності», оскільки усуває гонки даних завдяки відсутності спільного мутабельного стану. Це, у свою чергу, на пряму сприяє «Масштабованості» системи, оскільки дозволяє легко створювати безстанові середовища (Stateless), які легко горизонтально масштабуються.

Обговоримо інші аспекти впливу концепцій функціонального програмування на вимоги до програмного забезпечення. Принцип незмінності (іммутабельності) даних передбачає відсутність модифікації існуючих структур та створення нових значень у разі потреби зміни стану. Цей підхід суттєво підвищує рівень безпеки системи, оскільки усуває можливість несанкціонованої або неконтрольованої зміни даних, що характерно для імперативних програм із розгалуженою логікою. У багатопотокових та розподілених середовищах іммутабельність ліквідує одну з найпоширеніших причин програмних збоїв - гонки даних, які виникають через конкурентний доступ до спільного змінного стану. За умов використання незмінних структур потреба у складних механізмах синхронізації зменшується, що суттєво спрощує проектування паралельних алгоритмів і підвищує стабільність роботи системи під навантаженням.

Функціональні концепції мають прямий вплив на зрозумілість і підтримуваність програмного коду. Логіка, побудована на послідовності чистих функцій та композиції перетворень, є більш прозорою, оскільки позбавлена прихованих побічних ефектів і не потребує стеження за численними змінами стану. Такий код легше аналізувати, рефакторити та адаптувати до нових вимог, що особливо важливо в умовах довготривалих корпоративних проєктів. Використання декларативних конструкцій дозволяє зосередитися на сутності обчислення, а не на процедурних деталях його реалізації. Це сприяє підвищенню якості документації, полегшує колективну роботу над проєктом і прискорює процеси онбордингу нових розробників.

Таким чином, функціональне програмування на рівні концепцій та принципів забезпечує суттєві переваги у задоволенні критичних вимог до

програмного забезпечення, формуючи фундамент для побудови надійних, безпечних, зрозумілих і масштабованих програмних систем.

Крім підвищення надійності та зрозумілості, застосування функціональних концепцій позитивно впливає й на масштабованість програмного забезпечення. Завдяки імутабельності даних та чистоті функцій компоненти системи стають слабо зв'язаними й можуть виконуватися незалежно, що є критично важливим у мікросервісних, реактивних та хмарно орієнтованих архітектурах. У таких середовищах часто використовуються моделі, які потребують високого рівня паралелізму та розподілення навантаження між вузлами. Функціональний стиль, орієнтований на відсутність глобального стану, природно підтримує ці вимоги, оскільки окремі обчислювальні фрагменти можна без ризику побічних ефектів виконувати у різних потоках чи навіть на різних серверах. Це зменшує потребу в складних балансувальних механізмах та забезпечує стабільність продуктивності навіть у динамічно масштабованих системах.

Функціональні концепції також сприяють підвищенню модульності архітектури, що є однією з ключових вимог до якісного програмного забезпечення. Завдяки композиції та чіткому розмежуванню відповідальностей логіку системи можна поділити на ізольовані компоненти, які легко тестувати, змінювати та розширювати. Це робить архітектурні рішення більш стійкими до змін вимог, зменшує ризики інтеграційних помилок і забезпечує високу гнучкість системи в умовах швидкої еволюції проєкту. Особливо важливо те, що модульність, досягнута за допомогою функціонального підходу, не лише спрощує структуру, але й закладає основу для автоматичного тестування й застосування інструментів формального аналізу.

Важливим аспектом впливу функціонального програмування на критичні вимоги є підтримка формальних методів, оскільки чисті функції та імутабельні моделі даних легко описуються математичними структурами та алгебраїчними рівностями. Це дає можливість застосовувати формальні

докази коректності для окремих компонентів системи або навіть для абстрактних рівнів архітектури. Використання таких підходів підвищує рівень довіри до системи, що є особливо актуальним у сферах, де помилка може мати критичні наслідки - у фінансових застосунках, медичних системах, інфраструктурних рішеннях та авіоніці. Таким чином, функціональне програмування розширює стандартні підходи до верифікації, забезпечуючи нові інструменти для гарантування надійності систем.

Загалом, вплив концепцій функціонального програмування на задоволення критичних вимог до програмного забезпечення має комплексний характер. Чистота функцій сприяє передбачуваності й тестованості; імутабельність усуває ризики конкурентного доступу та підвищує безпеку; декларативність та композиція функцій забезпечують прозорість структури й підтримуваність архітектури; а формальна строгість підсилює можливості верифікації та доведення коректності системи. Сукупність цих властивостей створює міцний теоретичний і практичний фундамент для розроблення сучасних архітектур, що відповідають найвищим стандартам якості та надійності.

Ще одним важливим наслідком застосування функціональних принципів є підвищення стійкості програмних систем до помилок та небажаної поведінки. Оскільки чисті функції не взаємодіють із глобальним станом і не мають прихованих побічних ефектів, коло потенційних місць виникнення помилок значно звужується. Це допомагає локалізувати проблеми, спрощує процес діагностики та скорочує час на виправлення. Розробники можуть аналізувати функціональні компоненти окремо, не враховуючи складні контексти їх виконання, що підсилює можливості статичного аналізу та автоматизованих інструментів пошуку дефектів. У сукупності це створює більш передбачувану й стабільну поведінку системи в умовах реальної експлуатації.

З точки зору промислових процесів розроблення програмного забезпечення, функціональний стиль сприяє

формуванню архітектури з меншою кількістю залежностей між компонентами. Коли програмні модулі будуються як композиції функцій, які не змінюють стан, залежності набувають напрямку «дані → обробка», а не «модуль → модуль». Це суттєво знижує зчеплення системи і дає змогу вільніше масштабувати та розвивати окремі частини без ризику порушення роботи всієї архітектури. Така структурна гнучкість є важливою для великих і розподілених команд розробників, а також для проєктів, що еволюціонують упродовж багатьох років.

Окремо варто наголосити на ролі функціональних концепцій для підвищення рівня безпеки програмного забезпечення. Імутабельність не лише усуває ризики гонок даних, але й спрощує створення захищених моделей передачі інформації між компонентами. Оскільки дані не можуть бути змінені після створення, зломисник втрачає можливість ін'єкції небезпечних змін через неконтрольовані модифікації об'єктів. Декларативний стиль також зменшує кількість небезпечних побічних дій, які можуть виникати внаслідок неправильного використання ресурсів чи доступу до системних API. У поєднанні з суворими типізованими моделями - як у Haskell чи F# - функціональний підхід забезпечує високий рівень формального контролю над даними та потоками виконання, що робить системи більш стійкими до помилок і атак.

У довгостроковій перспективі використання функціонального програмування в архітектурному проєктуванні позитивно впливає на життєвий цикл програмного продукту. Підтримуваність і зрозумілість коду, зменшення логічних залежностей, прозорість структури та можливість легкої перевірки коректності - це ті характеристики, які визначають успішність програмних рішень у високо конкурентному середовищі цифрових індустрій.

Функціональний стиль дає змогу створювати системи, що швидше адаптуються до змін, легко розширюються, демонструють стабільні показники продуктивності та менше схильні до деградації якості з часом.

Отже, концепції функціонального програмування мають глибокий та системний вплив на критичні вимоги до сучасного програмного забезпечення. Вони забезпечують підвищення надійності, безпеки, масштабованості та прозорості архітектури, водночас роблячи програмні системи більш ефективними у розробці, супроводженні та еволюції. Це визначає їхню значущість як у наукових дослідженнях, так і в промислових програмних практиках, де функціональний підхід дедалі частіше стає основою для створення високоякісних програмних продуктів.

1.4. Висновки за розділом 1

У першому розділі здійснено систематизований аналіз еволюції програмних парадигм із виокремленням місця та значення функціонального програмування в загальному контексті розвитку теорії та практики програмної інженерії. Аргументовано, що на сучасному етапі функціональна парадигма рідко застосовується як домінуюча модель при створенні програмних систем, проте її концептуальні засади справляють істотний вплив на формування новітніх архітектурних шаблонів. Відзначено, що низка інструментів для обробки даних ґрунтується на використанні функцій вищого порядку та ідеї імутабельних структур.

Окрему увагу приділено історико-методологічному огляду становлення математичних передумов функціонального програмування. Розкрито фундаментальні положення λ -числення, яке виступає теоретичною основою функціональних мов. Серед ключових принципів окреслено: чистоту функцій, детермінованість обчислень, незмінність даних, декларативний стиль програмування, композиційність функцій, рекурсивні механізми та концепцію монад. Показано, що зазначені принципи стали визначальними для розвитку реактивного програмування, технології LINQ та мікросервісної архітектури.

Концептуальний апарат функціонального програмування справляє глибокий і системний вплив на формування критичних вимог до сучасних програмних систем. Його застосування забезпечує підвищення рівня надійності, безпеки, масштабованості та прозорості архітектурних рішень, водночас оптимізуючі процеси розробки, супроводу та еволюційного розвитку програмного забезпечення.

РОЗДІЛ 2. АРХІТЕКТУРНИЙ АНАЛІЗ ЗАСТОСУВАННЯ ФУНКЦІОНАЛЬНИХ ПАРАДИГМ

2.1. Імперативний, об'єктно-орієнтований та функціональний підходи: їхні концептуальні відмінності

У сучасному програмуванні можна виокремити три основні парадигми – імперативну, об'єктно-орієнтовану та функціональну, кожна з яких пропонує власний спосіб моделювання процесів і систем. Імперативний підхід ґрунтується на описі послідовності команд, що змінюють стан програми, використовуючи змінні, цикли та умовні конструкції. Він є простим для розуміння на низькому рівні та ефективним для алгоритмічних задач, однак ускладнює масштабування великих систем і контроль побічних ефектів. Об'єктно-орієнтований підхід, навпаки, моделює систему через об'єкти, які поєднують стан і поведінку, використовуючи принципи інкапсуляції, наслідування та поліморфізму. Це робить його зручним для розробки великих і складних систем, полегшує підтримку та розширення, проте може бути надмірно складним для простих задач і вимагати більше ресурсів.

Функціональний підхід пропонує декларативний стиль програмування, де акцент робиться на тому, що потрібно обчислити, а не на тому, як саме це зробити. Він базується на використанні чистих функцій, що працюють з незмінними даними, та рекурсії замість циклів. Такий підхід мінімізує побічні ефекти, полегшує тестування та паралелізацію, проте має високий поріг входу і може бути менш ефективним для певних типів задач. Головна відмінність між цими парадигмами полягає у способі мислення: **імперативний підхід орієнтується на процедури та зміну стану, об'єктно-орієнтований - на взаємодію об'єктів, а функціональний - на чисті функції та незмінні дані.** Таким чином, кожна парадигма має власні переваги й обмеження, що визначають її застосування залежно від контексту та цілей програмної системи.

Імперативний, об'єктно-орієнтований та функціональний підходи до програмування формують різні уявлення про організацію обчислень і структуру програмних систем. Імперативний стиль ґрунтується на послідовному виконанні команд, які змінюють стан програми в часі. Основою такого підходу є керування потоком виконання та мутабельні змінні, що забезпечують можливість покрокового оновлення стану. На противагу цьому, об'єктно-орієнтований підхід вводить концепції інкапсуляції, класів і об'єктів, кожен із яких поєднує стан і поведінку. Тут зміна стану розглядається як взаємодія об'єктів через методи, а структура програмного забезпечення визначається ієрархіями класифікацій та механізмами спадкування. Функціональне програмування, натомість, базується на моделі обчислень, що розглядає програму як композицію чистих функцій, де результат залежить лише від переданих аргументів. Зміна стану осмислюється не як модифікація існуючих значень, а як створення нових, що відображає математичну природу функціональної парадигми.

Ключовою концептуальною відмінністю цих підходів є спосіб керування станом. Імперативні та об'єктно-орієнтовані системи використовують мутабельні структури даних, що передбачають зміну внутрішнього стану об'єкта протягом усього життєвого циклу. Такий підхід є природним для моделювання реальних процесів, однак створює складність у відстеженні змін, особливо у великих системах із численними взаємодіями. У функціональному ж програмуванні стан зазвичай незмінний: після створення значення воно не може бути змінене. Це усуває цілу низку проблем, пов'язаних із неконтрольованими змінами даних, спрощує тестування, робить поведінку системи передбачуваною та знижує ризики, пов'язані з синхронізацією в багатопотоковому середовищі.

Відмінності простежуються і в організації модульності систем. Об'єктно-орієнтований підхід базується на класах, об'єктах і механізмах спадкування, що формують ієрархічну модель організації коду. Модулі в ООП об'єднують стан і функціональність, визначають межі відповідальності та

утворюють деревоподібні структури. Водночас функціональний стиль пропонує модульність, що ґрунтується на композиції функцій: функції виступають первинними будівельними блоками, які об'єднуються в нові функції без зміни існуючих. Це сприяє більшій гнучкості та повторному використанню, оскільки модулі не залежать від внутрішнього стану та поведуть себе однаково за будь-яких умов. Функціональна композиція дозволяє вибудовувати складні обчислення шляхом об'єднання простих функцій, що дає змогу зменшити зчеплення та спростити логіку

Відповідно до теми та завдань магістерської роботи розглянемо ці відмінності в контексті архітектурного проектування ПЗ. На рис. 2.1 представлено порівняльну схему парадигм програмування.

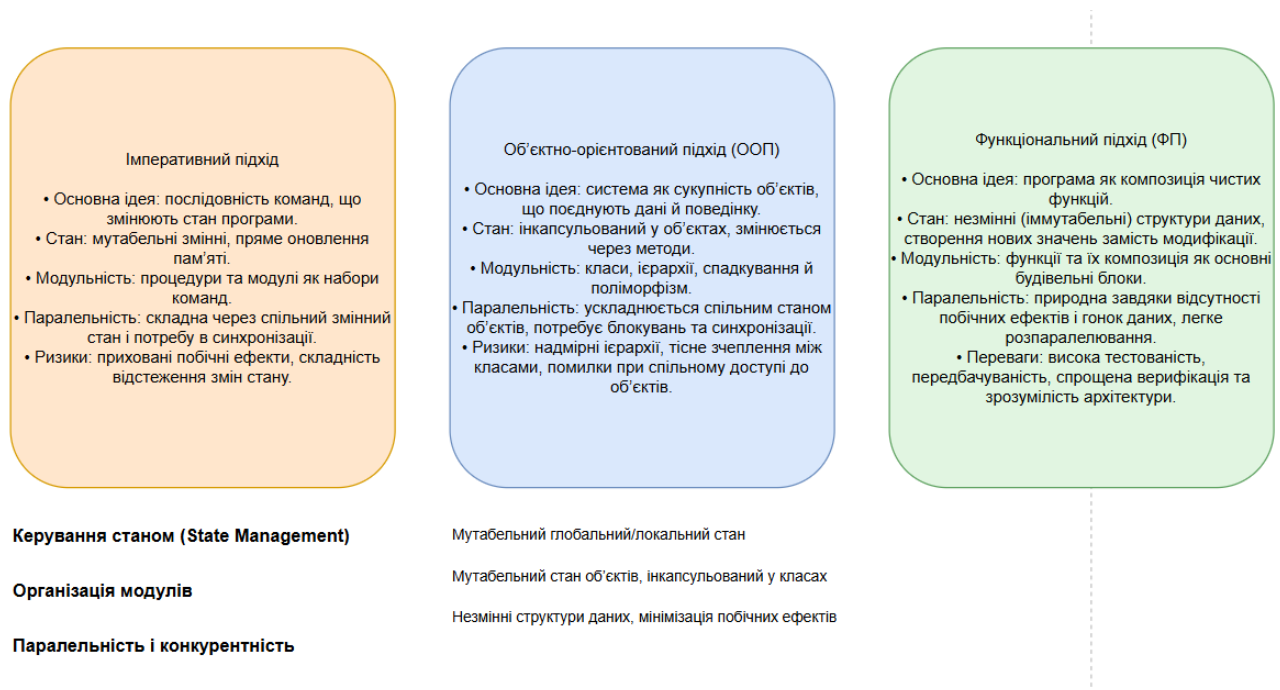


Рис. 2.1. Порівняння парадигм програмування в контексті архітектурного проектування.

Архітектурні аспекти забезпечення паралельності та конкурентності демонструють особливо значні контрасти між парадигмами. В ООП кожен об'єкт може мати власний стан, який змінюється внаслідок виконання методів, що створює потенційні конфлікти при доступі з кількох потоків. Це вимагає

складних механізмів синхронізації: блокувань, м'ютексів, семафорів або управління чергами повідомлень. У функціональному програмуванні паралельність отримує суттєві переваги завдяки незмінності даних та чистоті функцій: відсутність побічних ефектів означає, що одна й та сама функція може безпечно виконуватися в декількох потоках над незалежними наборами даних. У розподілених системах це дає можливість досягати високої масштабованості, оскільки обчислення легко розділяються та розпаралелюються.

Обґрунтуємо вибір архітектурних критеріїв: він базується на необхідності комплексного аналізу впливу фундаментальних парадигм програмування на якісні характеристики сучасних програмних систем. Зокрема, такі параметри, як концептуальна основа, організація модулів та модель паралельності, виступають визначальними факторами при формуванні структурного скелета системи, оскільки вони безпосередньо диктують способи декомпозиції складності та стратегії обробки даних у багатопотокових середовищах. Керування станом та контроль побічних ефектів є критичними аспектами для забезпечення детермінованості поведінки програмного забезпечення, що набуває особливої ваги в умовах зростання вимог до надійності та передбачуваності функціонування розподілених систем, де мінімізація мутабельності стає ключем до масштабованості.

Водночас, вибір критеріїв повторного використання, тестованості та зрозумілості коду зумовлений потребою в оцінці довгострокової ефективності розробки та супроводу програмних продуктів. Архітектурна гнучкість у поєднанні з аналізом схильності до помилок дозволяє не лише прогнозувати адаптивність системи до змін у бізнес-вимогах, але й ідентифікувати потенційні вразливості на ранніх етапах проектування. Таким чином, сукупність цих критеріїв формує цілісну метричну базу для порівняльного аналізу, дозволяючи об'єктивно оцінити, наскільки обраний архітектурний підхід сприяє досягненню високих показників якості, зниженню технічного

боргу та забезпеченню стійкості системи в умовах еволюції технологічного ландшафту.

На основі цих критеріїв у табл. 2.1-2.3 наведено порівняння підходів програмування в контексті архітектурного проектування.

Таблиця 2.1

Порівняльна таблиця підходів програмування в контексті архітектурного проектування (концептуальна модель та організація коду)

Критерій	Імперативний підхід	Об’єктно-орієнтований підхід (ООП)	Функціональний підхід (ФП)
Концептуальна основа	Послідовність команд, що змінюють стан	Моделювання системи через об’єкти, що поєднують дані й поведінку	Композиція чистих функцій, математична модель обчислень
Організація модулів	Модулі - набори процедур	Модулі - класи/об’єкти, ієрархії, спадкування	Модулі - функції та композиції функцій
Повторне використання	Використання процедур і бібліотек	Спадкування, інтерфейси, поліморфізм	Компонування функцій, вищі функції, безсторонність (statelessness)

Табл. 1.2 демонструє, як кожна парадигма формує архітектурне мислення: від процедурної лінійності до об’єктної взаємодії та декларативної чистоти, що має суттєвий вплив на масштабованість, тестованість і підтримуваність програмних систем

Таблиця 2.2

Порівняльна таблиця підходів програмування в контексті архітектурного проєктування (Керування станом і модель виконання)

Критерій	Імперативний підхід	Об'єктно-орієнтований підхід (ООП)	Функціональний підхід (ФП)
Керування станом	Активно змінюваний, глобальний або локальний мутабельний стан	Стан інкапсульовано в об'єктах, змінюється через методи	Нездатність до зміни стану (імутабельність), створення нових значень
Побічні ефекти	Поширені; зміна змінних, I/O	Присутні; зміна внутрішнього стану об'єктів	Мінімізовані; чисті функції без побічних ефектів
Модель паралельності	Складна: конфлікти станів, потреба в синхронізації	Проблемна через спільний стан об'єктів; потребує блокувань	Природна: відсутність гонок даних завдяки незмінності, легке розпаралелювання
Придатність для розподілених систем	Обмежена	Обмежена через складність синхронізації	Дуже висока: природна для реактивних та мікросервісних архітектур

Табл. 2.2 підкреслює, що функціональний стиль забезпечує кращу модульність, тестованість і масштабованість у сучасних архітектурних рішеннях.

Порівняльна таблиця підходів програмування в контексті архітектурного проектування (якість, надійність і супровід)

Критерій	Імперативний підхід	Об'єктно-орієнтований підхід (ООП)	Функціональний підхід (ФП)
Тестованість	Ускладнюється побічними ефектами	Середня; залежить від кількості взаємодій між об'єктами	Висока: чисті функції легко тестувати ізольовано
Зрозумілість коду	Складна у великих системах через змінний стан	Краще за імперативний, але ускладнюється великими ієрархіями	Висока: декларативність і композиція підвищують прозорість
Архітектурна гнучкість	Низька; код часто тісно пов'язаний зі станом	Середня; залежить від дизайну класів	Висока; функції легко комбінуються, замінюються та масштабуються
Схильність до помилок	Вести до помилок через приховані зміни стану	Помилки часто пов'язані зі станом об'єктів	Мінімальна: детермінованість та відсутність мутацій знижують ризики

Архітектурні аспекти забезпечення паралельності та конкурентності демонструють особливо значні контрасти між парадигмами. В ООП кожен об'єкт може мати власний стан, який змінюється внаслідок виконання методів, що створює потенційні конфлікти при доступі з кількох потоків. Це вимагає складних механізмів синхронізації: блокувань, м'ютексів, семафорів або управління чергами повідомлень. У функціональному програмуванні паралельність отримує суттєві переваги завдяки незмінності даних та чистоті функцій: відсутність побічних ефектів означає, що одна й та сама функція

може безпечно виконуватися в декількох потоках над незалежними наборами даних. У розподілених системах це дає можливість досягати високої масштабованості, оскільки обчислення легко розділяються та розпаралелюються.

Таким чином, концептуальні відмінності між імперативним, об'єктно-орієнтованим і функціональним підходами полягають у різному способі сприйняття стану, структури модулів і обчислювальних моделей. Функціональна парадигма пропонує рішення, що усувають низку обмежень традиційних підходів, зокрема в контексті паралельності, тестованості та передбачуваності системи, що робить її цінним інструментом для створення сучасних програмних архітектур.

2.2. Інтеграція функціональних концепцій у гібридні мови програмування

Застосування можливостей функціонального програмування у сучасних мовах Scala, Kotlin, JavaScript та C# демонструє тенденцію інтеграції функціональних концепцій у широковживані екосистеми, орієнтовані на промислове використання. Scala є однією з найбільш виражено гібридних мов, яка поєднує об'єктно-орієнтовану модель із потужними функціональними абстракціями. Її механізми незмінності даних, патерни типу `*map-filter-reduce*`, вищі функції, функціональні колекції та розгалужена система типів надають можливість створювати архітектурно стійкі рішення з формальною передбачуваністю. Широке застосування Scala у фреймворках на кшталт Akka та Spark підтверджує її ефективність у побудові розподілених, реактивних і високонавантажених систем, де імутабельність і чистота функцій стають основою безпечної конкурентності.

Kotlin, хоча й позиціонується як об'єктно-орієнтована мова для JVM, активно впроваджує функціональні можливості, роблячи їх частиною повсякденної розробки. Лямбда-вирази, функції вищого порядку, оператори колекцій та параметри з замиканнями сприяють декларативному стилю програмування. У Kotlin ключовим є акцент на безпечній роботі зі станом через властивість `val` та підтримку незмінних структур, що підвищує передбачуваність та зменшує кількість помилок часу виконання. Функціональні принципи також відіграють значну роль у корутинному підході до асинхронності, де моделі обчислень без побічних ефектів та відсутність змінного глобального стану дозволяють реалізовувати масштабовані неблокуючі операції.

У JavaScript функціональний стиль став домінантним у розробленні сучасних фронтенд-архітектур. React реалізує декларативну модель інтерфейсу як чисту функцію від стану, що формує UI-компоненти без побічних ефектів та забезпечує передбачуваність роботи. Redux, у свою чергу, суворо дотримується принципів незмінності стану та детермінованих

«ред'юсерів», що моделюють чисті функції трансформації даних. Такий підхід полегшує відстеження змін, спрощує тестування й гарантує стабільність поведінки навіть у складних інтерфейсах. Широкий набір функціональних утиліт у бібліотеках `Lodash`, `Ramda` чи `RxJS` підсилює можливості декларативної обробки даних, сприяючи побудові реактивних та асинхронних систем через «потoki подій» та композицію операторів.

Мова `C#` у сучасних версіях також демонструє глибоку інтеграцію функціональних конструкцій, найпоказовішим проявом яких є технологія `LINQ`. `LINQ` оперує моделлю функцій вищого порядку над колекціями, застосовуючи ідеї композиції, трансформацій та фільтрації без необхідності ручного керування ітераторами чи мутабельним станом. Завдяки цьому дані обробляються декларативно, а код стає більш компактним, зрозумілим і передбачуваним. Додаткові можливості, такі як `*immutable collections*` у `.NET`, лямбда-вирази, замикання й `async/await`, створюють підґрунтя для побудови моделей асинхронності, що мінімізують побічні ефекти та полегшують конкурентне виконання.

Отже, аналіз застосування функціональних можливостей у `Scala`, `Kotlin`, `JavaScript` та `C#` свідчить про їхній системний вплив на сучасні архітектурні практики. Незмінність стану, чисті функції, декларативні моделі обчислення та формальна передбачуваність стають ключовими інструментами підвищення надійності й масштабованості програмного забезпечення, незалежно від вихідної парадигми мови. Це підтверджує універсальність функціонального підходу та його здатність гармонійно інтегруватися в різні технологічні платформи.

Окремим аспектом аналізу є оцінка ступеня «функціональності» гібридних мов та їхнього впливу на підтримку архітектурних рішень. Гібридні мови, такі як `Scala`, `Kotlin`, `JavaScript` та `C#`, реалізують функціональні можливості в різному обсязі, що визначає їхню здатність формувати архітектури, орієнтовані на незмінність, декларативність та високу модульність. Ступінь функціональності можна оцінювати за критеріями

підтримки незмінних структур даних, наявності чистих функцій як повноцінних об'єктів, наявності механізмів обробки даних у стилі `map-filter-reduce`, а також можливостей безпечної паралелізації обчислень без складної синхронізації. Scala демонструє найбільшу наближеність до чистої функціональної моделі завдяки глибокій інтеграції імутабельності, потужної системи типів і розвинутим механізмам композиції, що дозволяє будувати архітектури з високим ступенем формальної строгості та передбачуваності.

Kotlin і C# пропонують середній рівень функціональності, зберігаючи об'єктно-орієнтовану основу, але інтенсивно використовуючи функціональні інструменти для підвищення архітектурної гнучкості. Kotlin застосовує функціональні моделі переважно у сфері роботи з колекціями, асинхронністю та декларативними DSL, що полегшує створення модульних і добре структурованих систем. C#, завдяки LINQ та підтримці незмінних колекцій, наближається до декларативного стилю у задачах обробки даних, але все ще зберігає значну залежність від об'єктного стану в загальній архітектурі. У JavaScript переважає прагматичний рівень функціональності: хоча мова не накладає суворих обмежень щодо імутабельності чи чистоти функцій, широке використання цих концепцій у React, Redux та функціональних бібліотеках створює умови для архітектур, орієнтованих на односторонній потік даних та детерміновані трансформації.

У цілому гібридні мови демонструють різний рівень відповідності функціональній парадигмі, що впливає на типи архітектурних рішень, які вони можуть підтримувати. Мови з високим рівнем функціональності полегшують побудову систем із чітким розділенням відповідальностей, стійкою моделлю стану та спрощеною паралельністю, тоді як мови з частковою підтримкою функціонального стилю забезпечують компроміс між гнучкістю об'єктно-орієнтованих підходів і перевагами декларативного моделювання. Це дозволяє адаптувати архітектурні стратегії до специфічних потреб проєкту, поєднуючи різні парадигми для досягнення оптимального балансу між продуктивністю, масштабованістю та підтримуваністю програмних систем.

Для порівняння мов сформулюємо низбку критеріїв. Функціональне програмування ґрунтується на низці принципів, які визначають його відмінність від імперативних чи об'єктно-орієнтованих підходів. Тому при порівнянні мов програмування важливо враховувати ступінь наближення до чистої функціональної парадигми: наскільки мова дозволяє працювати без змінного стану, з функціями вищого порядку та рекурсією як основним механізмом управління потоком. Це дає змогу оцінити, чи можна в конкретній мові писати код у стилі «чистого» ФП, чи вона лише частково підтримує такі ідеї.

Не менш важливим є питання незмінності даних та чистоти функцій, адже саме вони забезпечують передбачуваність і легкість тестування програм. Декларативність і можливість композиції функцій дозволяють створювати складні системи з невеликих модульних блоків, що підвищує зрозумілість і гнучкість коду. Окремо варто виділити критерій безпечної паралелізації: відсутність змінного стану робить функціональні програми природно придатними для багатопотокового та розподіленого виконання, що особливо актуально для сучасних систем.

Нарешті, вплив на архітектурну гнучкість і спрощення дизайну показує практичну цінність ФП у реальних проєктах. Мови, які підтримують функціональні принципи, дозволяють будувати масштабовані та легко модифіковані системи, де модульність і прозорість логіки знижують складність розробки. Сукупність цих критеріїв дає цілісну картину того, наскільки мова програмування сприяє застосуванню функціонального стилю не лише теоретично, а й у практичних сценаріях.

Таким чином набір критеріїв такий:

- ступінь наближення до чистої функціональної парадигми;
- підтримка незмінності (immutability) та чистих функцій;
- рівень декларативності та підтримка композиції;
- безпечна паралелізація та придатність до розподілених систем;
- вплив на архітектурну гнучкість і спрощення дизайну.

За цими критеріями зроблено порівняння мов програмування, результати представлено на рис. 2.2. та в додатку А.



Рис. 2.2 Порівняльний аналіз гібридних мов програмування за критеріями функціональності

2.3. Вплив парадигми функціонального програмування на сучасні архітектурні шаблони.

Вплив функціональної парадигми на еволюцію підходів до побудови архітектури програмного забезпечення можна прослідити на основі аналізу результатів деяких останніх публікацій. У статті [34] досліджується розвиток парадигм функціонального програмування для побудови графічних інтерфейсів, зокрема перехід від класичного підходу Model-View-Controller (MVC) до сучасних функціональних рішень на кшталт Model-View-Update (MVU), а також проблеми модульності та підтримуваності.

Еволюцію функціональних парадигм в статті представлено рис. 1.1. Автори показують, що MVC, започаткований у 1970-х роках, забезпечує розділення моделі та представлення, проте має низку суттєвих недоліків: дублювання логіки при оновленні інтерфейсу, циклічні виклики між моделлю та представленням, а також складність розбиття великих інтерфейсів на незалежні компоненти. Додатково класичні UI-фреймворки, що базуються на нескінченному циклі обробки подій, породжують явище «callback hell» і створюють сильну залежність між потоками управління. Еволюція функціональних підходів демонструється через аналіз ранніх бібліотек: eXene у Standard ML, що використовувала асинхронні повідомлення, але залишалася імперативною; Fudgets у Haskell, де інтерфейс розглядався як потік даних і застосовувалися чисті функції, проте приховувалися імперативні елементи; Fruit, побудована на Functional Reactive Programming, яка вирішувала проблему оновлення, але не забезпечувала належної модульності; Haggis, що поєднувала IO monad із реактивними механізмами.

Окремо розглядається Racket Universe teachpack, який пропонував спрощений функціональний підхід для навчання, здійснюючи повне перемальовування інтерфейсу при кожній зміні стану, що усуває проблему оновлення, але жертвує модульністю.

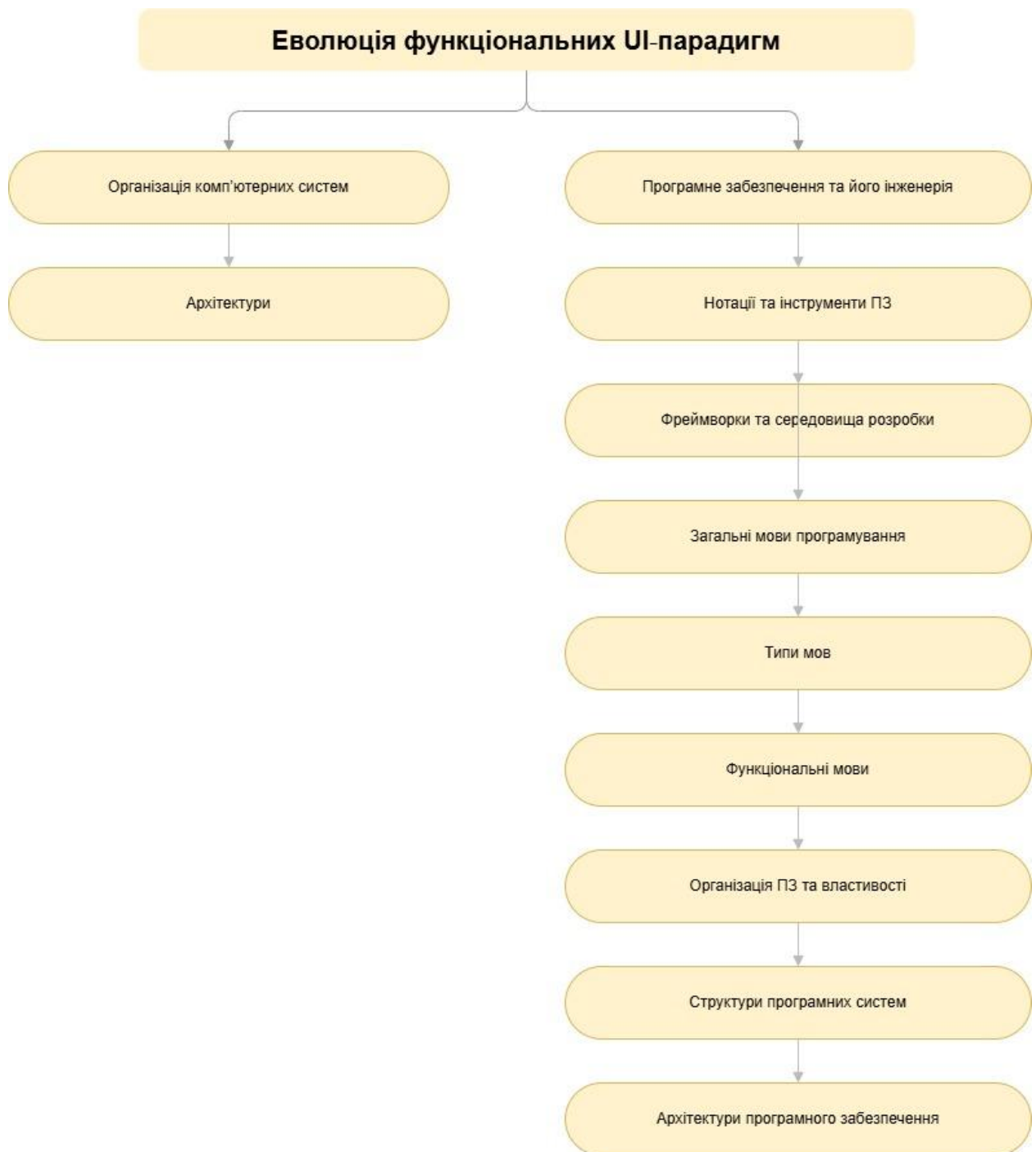


Рис. 2.1 Еволюція функціональних UI парадигм.

Сучасні рішення представлені системами Elm та React. Elm реалізує парадигму MVU через дві чисті функції - view та update, що забезпечує простоту та відсутність циклічних залежностей, проте страждає від слабкої модульності та глобального типу повідомлень. Подальший розвиток ідеї втілено у react-c, а також підтримується ViewModel, аналогічний MVVM, що

дозволяє відокремити доменну модель від стану інтерфейсу. У висновках автори підкреслюють, що MVC добре підходить для модульності, але погано для оновлення, тоді як MVU вирішує проблему оновлення, проте не забезпечує належної модульності. React та react-c є спробою поєднати переваги обох підходів, а використання ViewModel та функціональних комбінаторів сприяє підвищенню тестованості та повторного використання компонентів [34].

Робота [34] вийшла у 2025 р. та індексована *Scopus*, що ще раз підтверджує актуальність обраної теми.



Рис. 3.1. Архітектурна модель Model-View-Update (MVU) [34]

Функціональні підходи відіграють визначальну роль у формуванні архітектурної моделі Model-View-Update (MVU), яка набула популярності завдяки таким фреймворкам, як Elm та React з Redux. Архітектура MVU (рис. 3.1.) ґрунтується на суворому поділі відповідальностей та використанні детермінованих трансформацій стану, що є безпосереднім втіленням принципів функціонального програмування. У цій моделі **model** представляє незмінний стан застосунку, **view** є чистою функцією від стану, яка визначає, що саме відображається користувачеві, а **update** – чистою функцією, яка, отримуючи початковий стан та подію, обчислює новий стан. Такий підхід усуває побічні ефекти у відтворенні інтерфейсу та робить поведінку програми передбачуваною та формально вивірною.

Центральною концепцією MVU є незмінність стану, яка забезпечує простоту reasoning, полегшує відстеження змін і значно зменшує ймовірність виникнення логічних помилок, що часто спостерігаються в імперативних UI-архітектурах. Оскільки кожна зміна стану представлена як новий об'єкт, це створює можливість застосування таких інструментів, як time-travel debugging, автоматичне відтворення користувацьких дій та детермінований rollback. Використання чистих функцій у секціях **view** та **update** спрощує тестування компонентів і забезпечує математичну прозорість поведінки, що є основою високої надійності у розробленні інтерактивних інтерфейсів.

MVU природно поєднується з реактивними та подієвими архітектурами, оскільки обробка подій у цій моделі зводиться до послідовності функціональних перетворень. У поєднанні з потоковими бібліотеками або сигналами (signals) у функціональних фреймворках формується середовище, де UI поводить себе як функція від потоку станів, а логіка обробки подій залишається ізольованою та передбачуваною. Це дозволяє досягти високого рівня масштабованості та адаптивності інтерфейсу – програма легко переносить збільшення складності, а логіка лишається чистою та контрольованою. У підсумку MVU є архітектурним прикладом того, як принципи функціонального програмування – незмінність, чисті функції та декларативне моделювання – можуть забезпечити стабільність, передбачуваність і стійкість до зростання складності в UI-орієнтованих системах.

У роботі [35]) (також індексується *Scopus*) стверджується, що функціональне програмування має потужний набір технік для розробки програмного забезпечення, але бракує систематизованих знань про архітектуру великих систем, що робить його малодоступним для архітекторів і великих проєктів. Автори зазначають, що сьогодні індустрія успішно застосовує функціональні підходи у масштабних рішеннях, проте більшість практик передається як «фольклор» або розпорошена по десятиліттях публікацій ICFP, що ускладнює їх використання. Однією з ключових проблем

є відсутність узгодженого підходу до інтеграції архітектурних принципів у функціональні системи, адже спільнота архітектури ПЗ накопичила значний досвід і методології, які майже не перетинаються з функціональним світом. Мета сучасних досліджень полягає у поєднанні теоретичних основ функціонального програмування з перевіреними архітектурними підходами, щоб забезпечити ефективність і масштабованість сучасних програмних систем [35].

Функціональне програмування справляє суттєвий вплив на формування та еволюцію сучасних архітектурних шаблонів, зокрема в сфері реактивного програмування. Реактивні системи, орієнтовані на асинхронну обробку подій, масштабованість та стійкість до навантаження, природно поєднуються з принципами функціональної парадигми. Ключові концепції функціонального підходу – чисті функції, незмінність даних та декларативне визначення обчислень – забезпечують основу для побудови детермінованих та передбачуваних моделей реактивної взаємодії. В умовах складної потокової обробки даних застосування чистих функцій зменшує кількість прихованих залежностей, полегшує тестування й істотно знижує ризик появи помилок, пов'язаних із мутабельним станом, що є критично важливим для систем зі складною асинхронною логікою.

Принцип незмінності відіграє особливо важливу роль у реактивних архітектурах, оскільки забезпечує безпечне паралельне опрацювання подій та потоків даних. У розподілених і високонавантажених середовищах зміна стану під час конкурентного доступу часто призводить до гонок даних, неконсистентності або складної взаємодії між компонентами. Імутабельні структури даних усувають потребу в блокуваннях і синхронізації, дозволяючи виконувати обчислення у незалежних потоках без ризику порушення цілісності. Це робить реактивні моделі, такі як Publisher–Subscriber або Actor Model, більш надійними та масштабованими. Багато сучасних фреймворків реактивного програмування, на кшталт Akka, RxJava чи Reactor,

безпосередньо опираються на незмінність та чистоту функцій для досягнення високої пропускну здатності та стабільності систем.

Крім того, функціональна декларативність сприяє формуванню зрозумілих та компактних описів потоків даних. У реактивному програмуванні складні асинхронні операції часто описуються композицією операторів трансформації, фільтрації чи агрегації подій. Чисті функції дозволяють будувати такі ланцюги без прихованих побічних ефектів, що полегшує аналіз та передбачення поведінки системи. Декларативний стиль створює можливість оптимізації виконання на рівні фреймворків, забезпечуючи ефективне використання ресурсів та адаптивне масштабування в режимі реального часу. У підсумку застосування функціональних принципів у реактивному програмуванні підсилює структурну дисципліну архітектури, забезпечує стійкість до високого навантаження та сприяє створенню програмних систем з високим рівнем надійності, передбачуваності і підтримуваності.

Концепції функціонального програмування відіграють важливу роль у реалізації архітектурного підходу **Event Sourcing**, оскільки він базується на ідеї фіксації змін стану через послідовність подій, а не через пряме модифікування об'єктів чи базових структур даних. У такій моделі система розглядає стан як похідну від історії подій, що цілком узгоджується з функціональним принципом чистих функцій: функція реконструкції стану з журналу подій є детермінованою та залежить лише від послідовності вхідних подій. Це робить поведінку системи передбачуваною, забезпечує прозору логіку обробки та спрощує формальну верифікацію правильності застосованих змін. У Event Sourcing на прикладі патерну на рис. 3.2 стан перестає бути мутабельним артефактом і натомість є результатом обчислення, що підсилює узгодженість цього підходу з функціональною парадигмою.

Незмінність (immutability) виступає ключовою властивістю, яка забезпечує коректну роботу архітектури, керованої подіями. Події, що фіксуються в журналі, не змінюються після запису, оскільки кожна подія є

незаперечним фактом, який відбувся в системі. Це дозволяє забезпечити високий рівень надійності, оскільки відсутність мутацій унеможливорює порушення консистентності історії системи. Імутабельність спрощує реплікацію, аудит, відлагодження та відтворення стану в будь-який момент часу, що є фундаментальним для побудови стійких до збоїв та розподілених систем. Також вона забезпечує можливість ефективного паралельного доступу до історії подій, оскільки немає ризику конкурентної модифікації даних, а відтак – немає потреби в дорогих механізмах синхронізації.

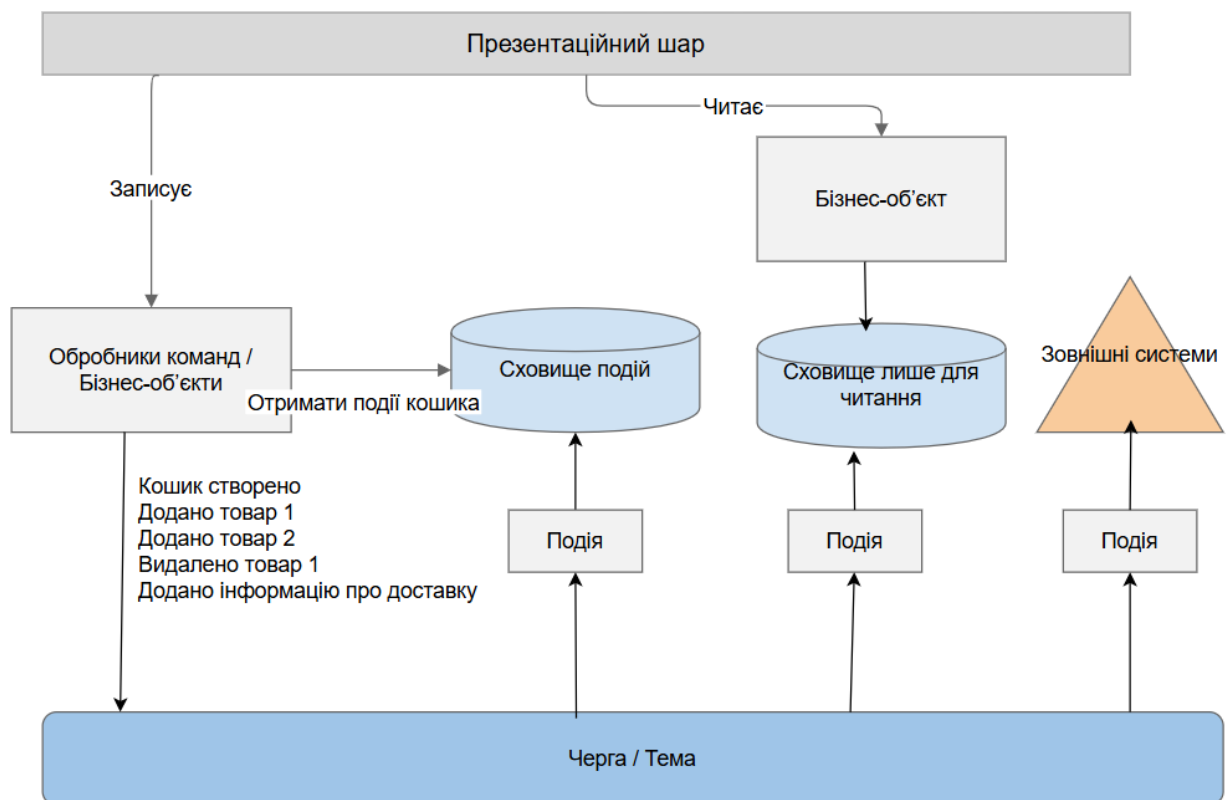


Рис. 3.2. Event Sourcing Pattern [14]

Декларативна природа функціональних підходів також посилює архітектуру Event Sourcing, роблячи логіку трансформації подій у стан більш прозорою й композиційною. Операції з подіями можна визначати як чисті функції трансформації, що полегшує побудову складних обробників подій та їх комбінацію у більш складні моделі проєкцій (projections) або читальних моделей (read models). Це сприяє мінімізації побічних ефектів, оскільки бізнес-

логіка фокусується на перетворенні потоків подій, а не на підтримці розгалуженого змінного стану. Завдяки цьому Event Sourcing легко поєднується з такими архітектурними підходами, як CQRS та EDA (рис. 3.4 та рис. 3.5), де функціональні абстракції дозволяють чітко розділити командну та читальну частини системи.

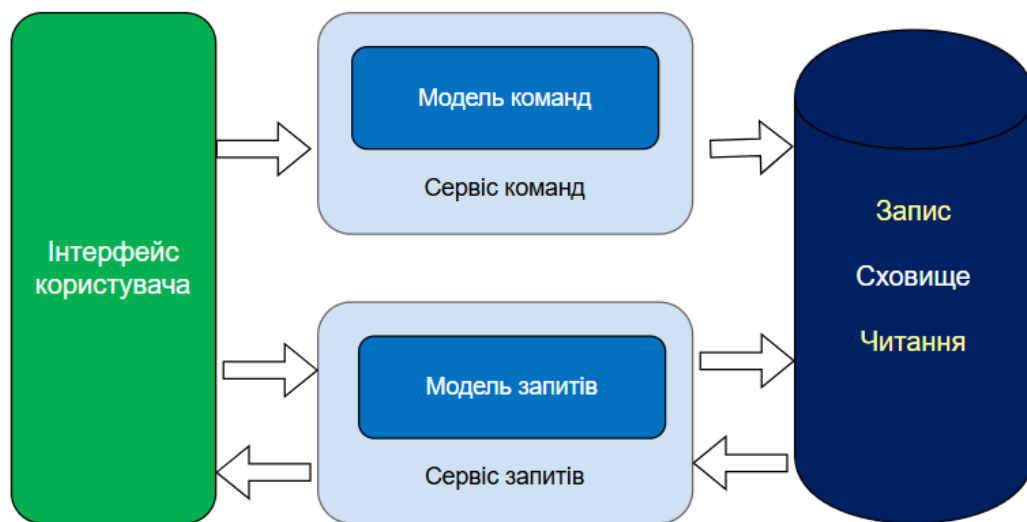


Рис. 3.3. CQRS (Command Query Responsibility Segregation) [18]

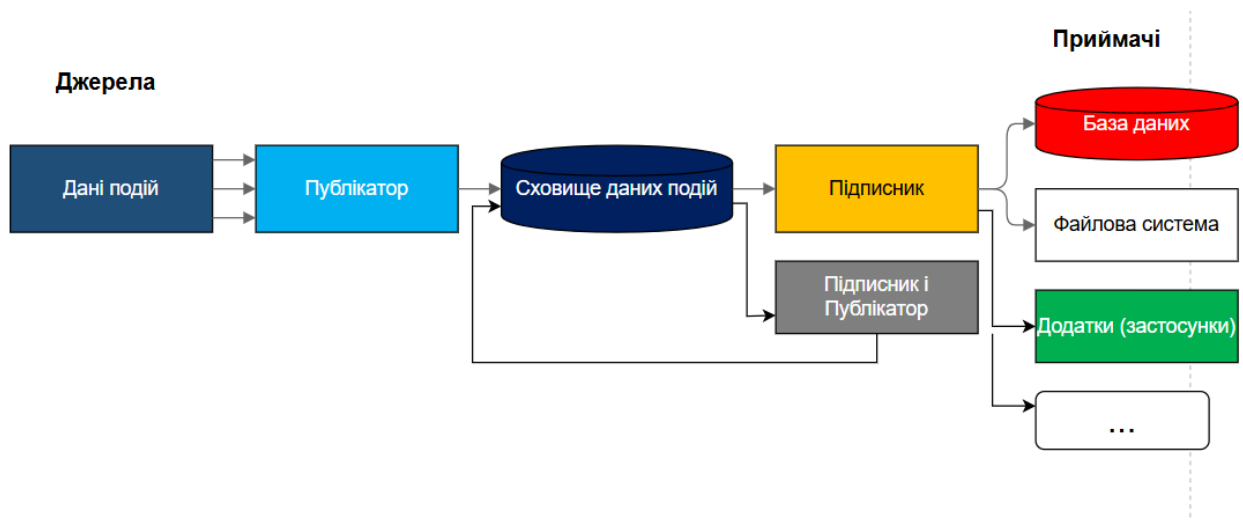


Рис. 3.4. Event-Driven Architecture (EDA) [13]

Таким чином, концепції функціонального програмування – чисті функції, незмінність даних і декларативна композиція – природно інтегруються в архітектуру Event Sourcing і значно підвищують її

ефективність. Вони забезпечують детермінованість, масштабованість, стійкість до збоїв і спрощують супровід логіки системи. Це робить функціональний підхід одним із найпридатніших концептуальних підґрунтів для реалізації архітектур, керованих подіями, особливо в умовах зростання складності й розподіленості сучасних програмних рішень.

Функціональні підходи органічно інтегруються в концепцію Domain-Driven Design (DDD), оскільки вони сприяють чіткому відділенню бізнес-логіки від технічних деталей, підвищуючи прозорість і формальну визначеність предметної області. Одним із ключових принципів DDD є акцент на моделюванні домену через мову предметної області (Ubiquitous Language) та побудова програмних компонентів, що відображають реальні бізнес-правила. Функціональні концепції, зокрема чисті функції та незмінність, дають можливість представляти бізнес-операції як детерміновані перетворення стану, що залежать лише від вхідних даних і не мають побічних ефектів. Такий підхід мінімізує ризики логічних помилок, пов'язаних зі зміною прихованого стану або неконтрольованими взаємодіями між класами, що традиційно ускладнює підтримку великих DDD-систем.

Незмінність (immutability) у контексті DDD виступає важливим механізмом підвищення надійності та прозорості бізнес-логіки. Оскільки агрегати й сутності часто використовуються для моделювання ключових доменних концептів, забезпечення їхньої незмінності або обмеження мутацій за допомогою чітко визначених інваріантів дозволяє гарантувати консистентність доменної моделі. Моделювання стану через імутабельні об'єкти зменшує кількість побічних ефектів і спрощує відтворення логіки при аналізі, тестуванні та аудиті доменних процесів. У великих системах це забезпечує передбачуваність поведінки доменних об'єктів, навіть коли операції виконуються у паралельному або розподіленому середовищі, що є особливо актуальним для складних корпоративних застосунків.

Функціональні підходи також підсилюють тактичні шаблони DDD, зокрема Domain Services, Value Objects та

Aggregates. Value Objects природно відповідають ідеї незмінності, а їхня рівність за значенням робить їх ідеальними для реалізації у функціональному стилі. Domain Services, які інкапсулюють бізнес-операції, зручно представляти у вигляді чистих функцій, що забезпечує високу тестованість та полегшує перевірку бізнес-правил. Навіть агрегати, які за своєю природою містять стан, можуть виграти від функціонального підходу: зміни агрегату можуть моделюватися як функція «поточний стан + команда → новий стан», що відтворює концепцію обчислюваності та детермінізму, характерну для функціональної парадигми.

Завдяки цим властивостям функціональне програмування підвищує здатність до формалізації бізнес-правил, полегшує рефакторинг моделі та робить систему більш стійкою до змін предметної області. Водночас воно забезпечує кращу ізолюваність бізнес-логіки від інфраструктурного шару, що спрощує впровадження таких архітектурних стилів, як Clean Architecture, Hexagonal Architecture або Event Sourcing, які активно використовуються разом із DDD у сучасних програмних системах. У підсумку функціональні підходи не лише узгоджуються з принципами Domain-Driven Design, але й значно підсилюють їх, забезпечуючи більшу прозорість, формальну точність і масштабованість доменної моделі.

Висновки до розділу 2

У другому розділі здійснено порівняльний аналіз трьох провідних парадигм програмування — імперативної, об'єктно-орієнтованої та функціональної. Показано, що імперативний підхід ґрунтується на послідовному виконанні процедур та постійній зміні стану системи, що відображає традиційну модель обчислень, близьку до апаратної реалізації алгоритмів. Об'єктно-орієнтований підхід, у свою чергу, акцентує увагу на моделюванні предметної області через взаємодію об'єктів, які поєднують дані та поведінку, забезпечуючи інкапсуляцію, спадкування та поліморфізм. Функціональна парадигма принципово відрізняється тим, що розглядає програму як композицію чистих функцій, які не мають побічних ефектів та працюють з незмінними структурами даних.

Аналіз інтеграції функціональних можливостей у сучасні мови програмування (Scala, Kotlin, JavaScript, C#) засвідчує системну тенденцію до поєднання традиційних парадигм із концепціями функціонального програмування. Незмінність даних, чисті функції, декларативні моделі обчислень та композиційність виступають універсальними інструментами, що забезпечують підвищення надійності, передбачуваності та масштабованості програмних систем.

Аналіз сучасних архітектурних рішень демонструє, що функціональні концепції стають ключовим інструментом у побудові модульних, передбачуваних та масштабованих систем. Парадигма MVU, реалізована в Elm, забезпечує простоту та відсутність циклічних залежностей завдяки використанню чистих функцій view та update, проте обмежується слабкою модульністю та глобальним типом повідомлень. Подальший розвиток цієї ідеї у react-с та застосування ViewModel (аналогічного MVVM) спрямовані на усунення цих недоліків, поєднуючи переваги MVC та MVU та створюючи умови для підвищення тестованості й повторного використання компонентів.

Функціональні принципи мають визначальний вплив на архітектурні підходи Event Sourcing, CQRS та EDA. Це створює основу для побудови реактивних і розподілених систем, де незмінність та чистота функцій гарантують безпечну конкурентність.

Інтеграція функціональних підходів у концепцію Domain-Driven Design (DDD) підсилює її методологічну цінність, оскільки сприяє чіткому відділенню бізнес-логіки від технічних аспектів та формує прозору модель предметної області.

Таким чином, функціональне програмування постає не лише як альтернативна парадигма, але й як методологічна основа для розробки сучасних архітектурних рішень, що відповідають критичним вимогам до надійності, тестованості та ефективності програмних систем.

РОЗДІЛ 3. РОЗРОБКА РЕКОМЕНДАЦІЙ ТА АПРОБАЦІЯ ЕФЕКТИВНОСТІ ВПРОВАДЖЕННЯ ФУНКЦІОНАЛЬНОГО СТИЛЮ

3.1. Критерії та метрики оцінки архітектурної якості функціонального програмного забезпечення

Одним із ключових аспектів є формалізація метрик зв'язності та зчеплення у функціональному стилі. У традиційних парадигмах зв'язність (cohesion) визначається ступенем взаємопов'язаності елементів всередині класу або модуля, а зчеплення (coupling) – рівнем залежності між різними модулями. У функціональному програмуванні модульність реалізується через композицію функцій, що робить вимірювання цих характеристик більш формальним. Зв'язність у функціональних системах може визначатися ступенем логічної узгодженості набору функцій, які разом реалізують цілісну трансформацію даних. Чим більш однорідною є доменна логіка модуля та чим менше він використовує зовнішні залежності чи контексти, тим вищою є його функціональна зв'язність. Зчеплення, у свою чергу, може оцінюватися кількістю входів і виходів функціонального модуля, необхідністю доступу до зовнішнього стану або використанням непередбачуваних побічних ефектів. Мінімізація залежностей між функціями та забезпечення повної детермінованості є ключовими критеріями зниження зчеплення у функціональних архітектурах.

Другим важливим аспектом є оцінка зрозумілості та прозорості потоку даних, що визначає легкість аналізу поведінки системи та простоту відстеження обчислювальних процесів. У функціональному стилі потік даних розглядається як послідовність або композиція перетворень, що забезпечує високу передбачуваність та зменшує кількість прихованих залежностей. Критеріями зрозумілості можуть бути: лінійність потоку даних (відсутність циклічних або прихованих переходів), мінімізація побічних ефектів, прозорість трансформацій (коли кожна функція виконує одну логічну операцію), а також структурна однорідність функцій. Метрики прозорості

можуть включати глибину композиції функцій, рівень декларативності (співвідношення між описовими та процедурними конструкціями) та кількість точок, де стан змінюється або передається. Чим простіше відстежити весь шлях даних у системі – від початкового введення до фінального результату, – тим вищою є архітектурна зрозумілість.

Окреме значення мають метрики оцінки тестованості та верифікації функціональних модулів. Завдяки детермінованості та відсутності побічних ефектів функціональні програми значно легше тестуються порівняно з імперативними. Метрики тестованості можуть включати: частку чистих функцій у модулі, середню кількість залежностей функції, можливість запуску функції у відриві від середовища виконання, а також обсяг необхідних тестових даних для покриття логічних гілок. Метрики верифікації можуть охоплювати рівень формальної визначеності (наявність типів, що гарантують коректність), можливість математичного доведення властивостей функцій, а також ступінь валідації трансформацій за допомогою property-based testing. Функціональний стиль, орієнтований на чистоту та незмінність, створює умови для автоматизації тестів, зменшує складність моделювання сценаріїв і дозволяє досягти високої точності перевірки логіки.

Базові принципи композиції чистих функцій та незмінності даних (immutability) формують основу для формалізації оцінювання (рис. 3.1).



Рис. 3.1 Формалізація оцінювання якості функціонального ПЗ

На рис 3.1 традиційні метрики зв'язності (Cohesion) і зчеплення (Coupling) трансформуються у функціональному стилі, набуваючи більш математично детермінованого характеру. У функціональному контексті зв'язність оцінює ступінь логічної та доменної узгодженості набору функцій, які агрегуються в модуль. Висока зв'язність корелює з принципом єдиної відповідальності на рівні модуля. Критерій «функціональна однорідність» означає, що модуль повинен реалізовувати цілісну трансформацію даних в межах єдиної доменної області.

Зчеплення вимірює рівень залежності між різними функціональними модулями. У функціональних архітектурах прагнення до низького зчеплення реалізується через мінімізацію неявних залежностей та побічних ефектів. Критерій «Декларативна Залежність» -залежності між модулями повинні бути явно визначені через входи та виходи функцій (аргументи та повернення значень). Зрозумілість (Clarity) та прозорість (Transparency) потоку даних є ключовими для легкості аналізу, відстеження та верифікації обчислювальних процесів. У функціональному стилі потік даних – це композиція перетворень. Критерій прозорості: референційна прозорість (Referential Transparency). Можливість замінити вираз його значенням без зміни поведінки програми. Це прямо пов'язано з використанням чистих функцій. Критерії зрозумілості: лінійність та ациклічність потоку: відсутність прихованих або циклічних залежностей у графі викликів. Дані мають текти передбачуваним шляхом. Структурна однорідність: використання стандартизованих шаблонів функціональної композиції (наприклад, `'map'`, `'filter'`, `'reduce'` або пайплайни даних) для підвищення візуальної зрозумілості.

Функціональний стиль природно забезпечує високий рівень тестованості завдяки детермінованості чистих функцій.

Критерій тестованості: ізольована перевірка. Можливість перевірити поведінку функції у відриві від середовища виконання або будь-якого зовнішнього стану. Критерій верифікації: формальна гарантія коректності. Можливість математичного доведення властивостей функцій, часто за

допомогою системи типів або формальних методів. Більш ретельно критерії описано у додатку Б.

Опишемо зазначені метрики математично.

1. Метрики зв'язності (Cohesion)

Міра чистоти модуля (M_{Pure}) представлено формулою (3.1):

$$M_{Pure}(Module) = \frac{\sum_{i=1}^{N_f} IsPure(f_i)}{N_f}, \quad (3.1)$$

де N_f – загальна кількість функцій у модулі, а $IsPure(f_i)$ дорівнює 1, якщо функція f_i є чистою (детермінованою, без побічних ефектів), і 0, якщо ні. Вище значення – краще.

Ступінь використання внутрішніх типів (SUIT) представлено формулою (3.2):

$$SUIT(Module) = \frac{\sum_{j=1}^{N_{ft}} UsesInternalType(f_j)}{N_{ft}}, \quad (3.2)$$

де N_{ft} – кількість функцій, які приймають аргументи або повертають значення. $UsesInternalType(f_j)$ дорівнює 1, якщо функція f_j взаємодіє із специфічним для домену типом, визначеним у цьому модулі, і 0, якщо використовує лише примітивні/загальні типи. Вище значення – краще.

2. Метрики зчеплення (Coupling)

Індекс нечистого зчеплення (ICI) представлено формулою (3.3):

$$ICI(Module) = \sum_{k=1}^{N_{fn}} \frac{AccessesExternalState(f_k)}{N_{fn}}, \quad (3.3)$$

де N_{fn} – кількість нечистих функцій у модулі. $AccessesExternalState(f_k)$ – кількість точок (змінних, I/O операцій), до яких функція f_k звертається за межами своїх аргументів. Індекс можна нормалізувати на N_{fn} або на загальну кількість функцій. Нижче значення – краще.

Кількість вхідних/вихідних посилок (NIV/NOV) представлено формулою (3.4):

$$\begin{aligned} NIV(Module) &= N_{callers} \\ NOV(Module) &= N_{callees}, \end{aligned} \quad (3.4)$$

$N_{callers}$ – кількість зовнішніх модулів, які викликають функції даного модуля. $N_{callees}$ – кількість зовнішніх модулів, функції яких викликає даний модуль. Чим нижче NIV і NOV, тим менше зчеплення.

3. Метрики прозорості та зрозумілості

Рівень декларативності (Declarative Ratio, DR) представлено формулою (3.5):

$$DR = \frac{N_{Declarative}}{N_{Procedural} + N_{Declarative}}, \quad (3.5)$$

де $N_{Declarative}$ – кількість описових конструкцій (наприклад, map, filter, reduce, композиції функцій) і $N_{Procedural}$ – кількість процедурних конструкцій (наприклад, for цикли, імперативні if/else блоки). Вище значення – краще.

Глибина композиції (Composition Depth, CD) представлено формулою (3.6):

$$CD = \frac{\sum_{i=1}^{N_p} L_i}{N_p}, \quad (3.6)$$

де N_p – загальна кількість пайплайнів або композицій функцій у системі, а L_i – кількість послідовних функцій у i -му ланцюгу (наприклад, а . b . с має глибину 3). Оптимальне значення CD може бути емпіричним, занадто високе може знижувати зрозумілість

4. Метрики тестованості та верифікації

Частка чистих функцій (Pure Function Share, PFS) представлено формулою (3.7):

$$PFS = \frac{N_{Pure}}{N_{Total}}, \quad (3.7)$$

це аналогічно M_{Pure} , але застосовується до всієї кодової бази (N_{Total} – загальна кількість функцій). Високий PFS означає високу ізольовану тестованість.

Покриття Property-Based тестами (PBT Coverage) представлено формулою (3.8):

$$PBT_{Cov} = \frac{N_{Properties\ Tested}}{N_{Critical\ Business\ Properties}}, \quad (3.8)$$

де $N_{Properties\ Tested}$ – кількість фундаментальних властивостей (наприклад, комутативність, ідемпотентність) доменної логіки, для яких існують PBT-тести, а $N_{Critical\ Business\ Properties}$ – загальна кількість таких властивостей. Вище значення – краще.

Коефіцієнт залежності від середовища (Environment Dependency Ratio, EDR) представлено формулою (3.9):

$$EDR = \frac{N_{EnvDependent}}{N_{Total}}, \quad (3.9)$$

де $N_{EnvDependent}$ – кількість функцій, які вимагають ініціалізації зовнішнього середовища (бази даних, мережі, файлової системи) для виконання тесту. Нижче значення – краще.

Отже, критерії та метрики оцінки архітектурної якості функціонального ПЗ ґрунтуються на принципах чистоти, композиційності та незмінності.

Вони забезпечують формальну, вимірювану основу для створення стійких, передбачуваних, легких у супроводі та верифікованих систем, що є ключовим для довгострокової стійкості програмного продукту. Оцінювання архітектурної якості програмного забезпечення, розробленого у функціональному стилі, потребує використання критеріїв і метрик, адаптованих до специфіки функціональної парадигми. На відміну від об'єктно-орієнтованих чи імперативних систем, де модулі часто характеризуються через взаємодії між об'єктами та їхніми станами, функціональні системи базуються на композиції чистих функцій і незмінності даних. Це вимагає перегляду традиційних підходів до вимірювання зв'язності, зчеплення, зрозумілості та тестованості, формуючи нові способи формалізації архітектурної якості.

3.2. Розробка рекомендацій щодо доцільного впровадження функціонального програмування

Доцільне впровадження функціонального програмування у процес інженерії програмного забезпечення потребує врахування особливостей предметної області, архітектурних вимог та організаційних обмежень проєкту. Одним із ключових рекомендацій є поступове впровадження функціональних концепцій у гібридних середовищах, де застосовуються традиційні парадигми, зокрема об'єктно-орієнтована. Оптимально починати з локального використання чистих функцій, незмінних структур даних і декларативних трансформацій у модулях, що обробляють бізнес-логіку або дані, оскільки саме ці сегменти найбільше виграють від детермінованості та зменшення побічних ефектів. Поступова інтеграція дозволяє уникнути різкого переходу архітектури та зменшує навантаження на команду в контексті навчання та адаптації.

Особливу увагу рекомендується приділяти принципам незмінності та контролю за станом системи, оскільки вони найбільш суттєво впливають на масштабованість і надійність програмного забезпечення. Впровадження імутабельних структур даних, уникнення глобального стану та побудова модулів у стилі «вхідні дані → функція → вихідні дані» дозволяють підвищити стійкість системи до конкурентного доступу й забезпечити передбачуваність результатів. Такі підходи особливо актуальні в розподілених і реактивних архітектурах, де складність взаємодії між компонентами зростає із збільшенням навантаження. Водночас важливо забезпечити підтримку ефективних механізмів створення

та копіювання незмінних структур, щоб уникнути зниження продуктивності при їх активному використанні.

Рекомендується також застосовувати функціональний стиль у комбінації з архітектурними патернами, які природно поєднуються з функціональними принципами, такими як Model-View-Update (MVU), CQRS, Event Sourcing та реактивне програмування. Вибір відповідного патерну має ґрунтуватися на вимогах до керування станом, асинхронності та структурної прозорості системи. У складних доменах доцільно використовувати функціональний підхід для моделювання Value Objects та Domain Services, що підвищує формальну строгість та передбачуваність доменної логіки. Успішне впровадження функціонального програмування також передбачає розвиток компетентностей команди, включно з формальною культурою тестування, композиційним мисленням та розумінням типових функціональних абстракцій. У сукупності ці рекомендації дозволяють отримати максимальну користь від функціональної парадигми та забезпечити стійкість, гнучкість і високу якість розроблюваних програмних систем.

3.3. Апробація запропонованих рекомендацій та оцінка результатів

Для перевірки запропонованих рекомендацій розроблено простий тестовий проєкт «Functional Counter».

Його функціональність:

- Є лічильник (ціле число).
- Кнопки: +1, -1, Reset.
- Поруч виводиться історія змін (ListBox), наприклад:

+1 → 1, +1 → 2, -1 → 1, Reset → 0.

- У Label внизу – дата/час останньої операції.

У цьому проєкті:

- використано сучасну функціональну архітектуру **Model-View-Update** в WinForms;
- продемонстровано функціональний стиль: незмінна модель, чиста функція Update, відокремлений View.

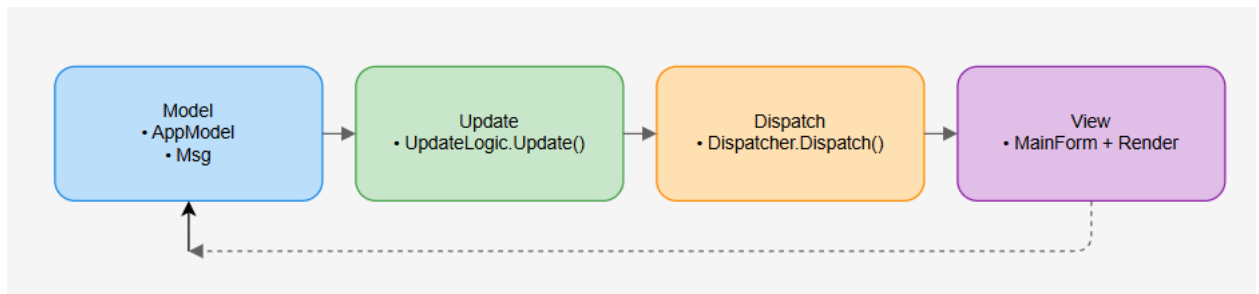


Рис. 3.2. Архітектурна модель додатку «Functional Counter»

На рис. 3.2 зображено архітектурну модель «Functional Counter». Згідно з цією моделлю Update нічого не змінює, але повертає нову модель.

На рис. 3.3 представлено структурну модель «Functional Counter», де Model – містить незмінну AppModel, Msg керує подіями, а ModelFactory містить початкову модель та фабричні методи.

Update містить чисту функцію Update, яка приймає модель та подію Msg і повертає нову модель.

View містить WinForms UI, яке не містить бізнес-логіки. Render викликає диспетчер з Core для візуальних компонентів, яка зв'язує Update з UI. Код застосунку приведено в Додатку В.

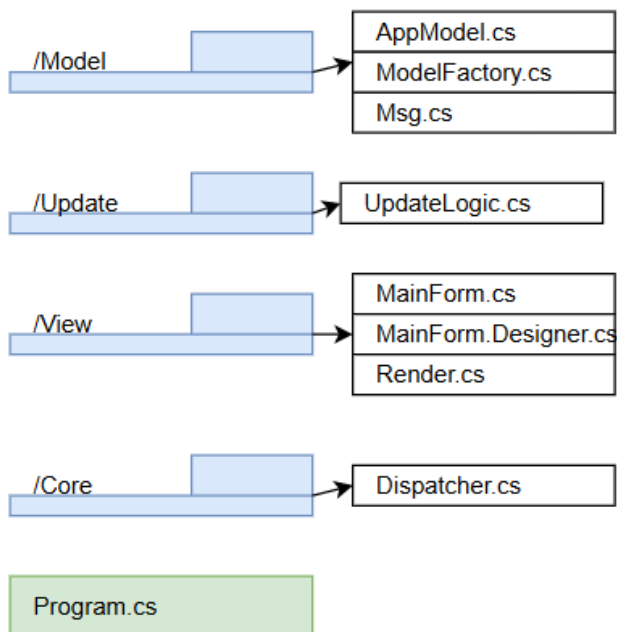


Рис. 3.3. Архітектурна структура застосунку «Functional Counter»

Застосунок має простий інтерфейс (рис. 3.4) тому, що основна увага приділена архітектурним та функціональним підходам.

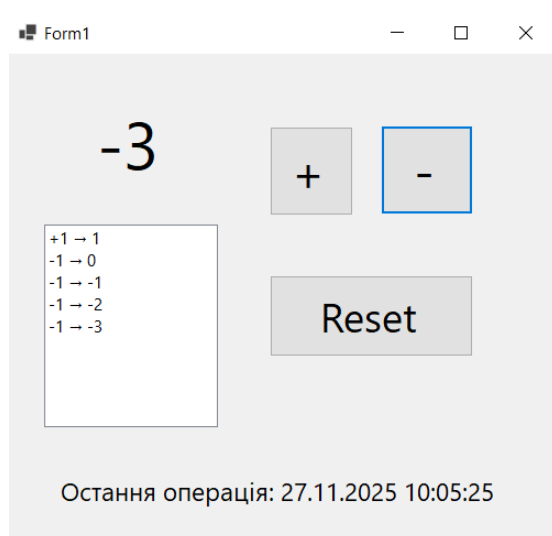


Рис. 3.4. Інтерфейс програми «Functional Counter»

Наступним кроком була розробка більш складного застосунку MatrixCalculator («Матричний калькулятор») на основі MVU. Архітектуру застосунку представлено на рис. 3.5.

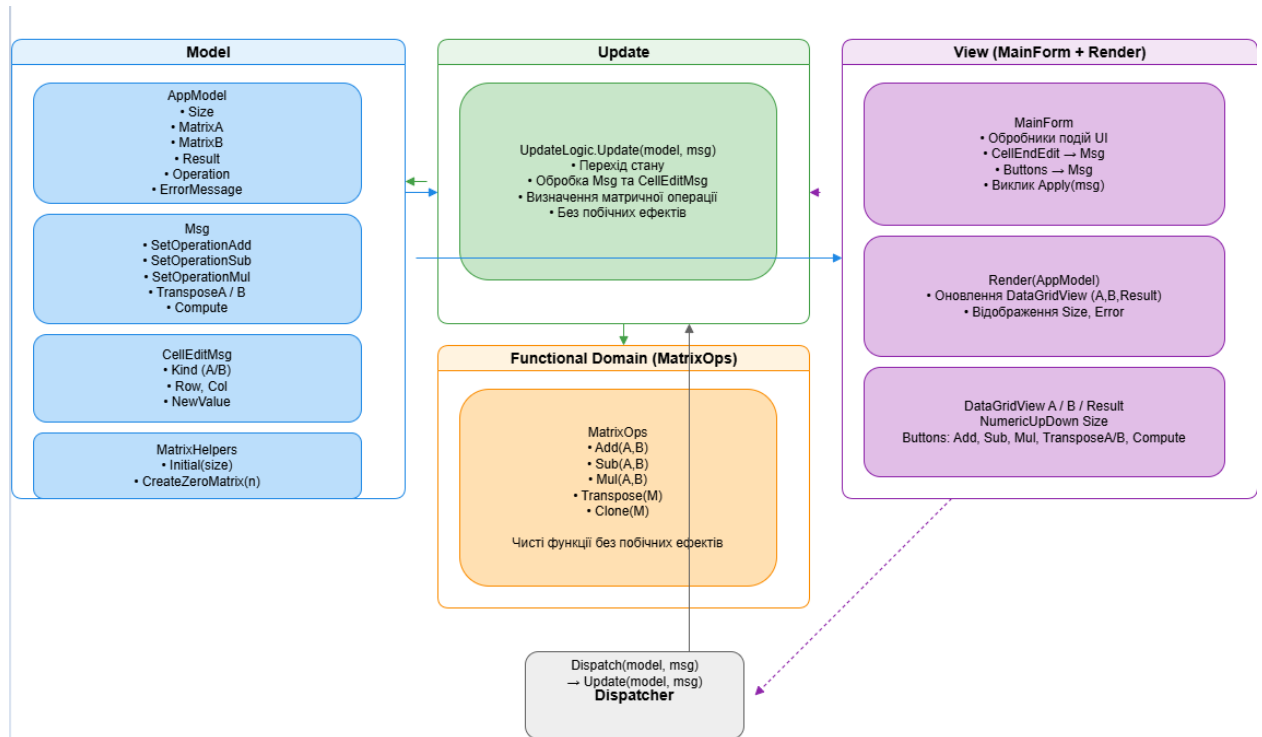


Рис. 3.5. Архітектура застосунку «MatrixCalculator» (MVU)

Код застосунку наведено в додатку Г. Структура застосунку представлено схемою рис. 3.6.

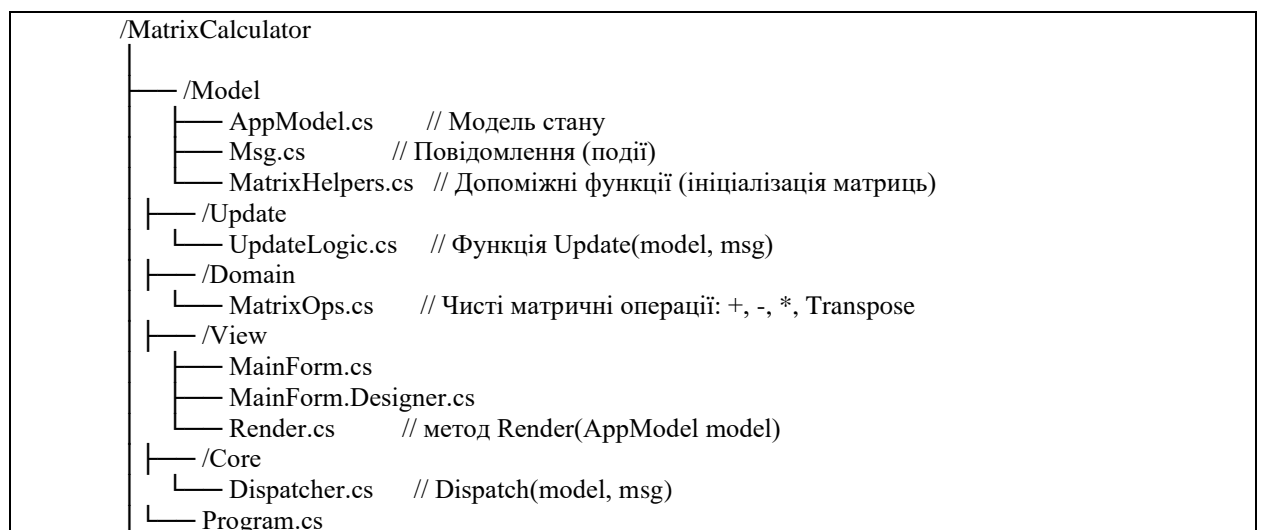


Рис. 3.6. Схема проєкту MatrixCalculator

Відповідно до критеріїв та метрик, які було розроблено в п.3.2 було проведена оцінка «Functional Counter» та «MatrixCalculator». Результати цього дослідження наведено в табл. 3.1.

Таблиця 3.1

Експериментальна оцінка ключових критеріїв та метрик
для «Functional Counter» та «MatrixCalculator»

Основний аспект	Ключовий критерій	Значення метрик
1. Зв'язність (Cohesion)	Функціональна однорідність	$M_{Pure} \approx 0.75 - 0.9$ (Висока)
2. Зчеплення (Coupling)	Декларативна залежність	$ICI = 0$ для чистої логіки. NIV/NOV (Низьке).
3. Прозорість та зрозумілість потоку даних	Референційна прозорість	$DR \approx 1$ (Максимальна декларативність).
4. Тестованість	Ізольована перевірка	$PFS \approx 0.75$ (Висока частка чистих функцій). $EDR = 0$ для Update.
5. Верифікація коректності	Формальна гарантія коректності	$PBT_{Cov} \approx 0.5$ (Можливість покриття ключових властивостей).

Як бачимо з таблиці можна констатувати середній рівень відповідності розробленим критеріям. Але це пов'язано з вибором інструментів та мови програмування для реалізації проєкту (C# та WinForms), що обумовлено наявними знаннями та навиками здобувача. Висновок підтверджує попередні результати з розділу 2 (рис. 2.2). У майбутніх дослідженнях доцільно обрати мову Scala.

Висновки до розділу 3

У функціональному програмуванні метрики зв'язності та зчеплення формалізуються через композицію чистих функцій та незмінних структур: зв'язність визначається узгодженістю набору функцій, що реалізують цілісну трансформацію, а зчеплення — кількістю зовнішніх залежностей, доступом до стану чи побічними ефектами. Прозорість потоку даних забезпечується передбачуваними перетвореннями без прихованих переходів, що спрощує аналіз і підвищує архітектурну ясність. Завдяки детермінованості та відсутності побічних ефектів функціональні модулі легко тестуються й верифікуються: чисті функції перевіряються незалежно від середовища, а формальна типізація та property-based testing забезпечують автоматизацію й точність перевірки логіки.

Доцільне впровадження функціонального програмування у процес інженерії програмного забезпечення передбачає поступову інтеграцію його концепцій у гібридні середовища, що поєднують традиційні парадигми, зокрема об'єктно-орієнтовану, починаючи з локального використання чистих функцій, незмінних структур даних та декларативних трансформацій у бізнес-логіці й обробці даних, що забезпечує детермінованість і зменшення побічних ефектів. Особливий акцент робиться на принципах імутабельності та контролю стану, які підвищують масштабованість і надійність систем, особливо у розподілених та реактивних архітектурах, де важливо підтримувати ефективні механізми роботи з незмінними структурами. Функціональний стиль доцільно поєднувати з архітектурними патернами, такими як MVU, CQRS, Event Sourcing та реактивне програмування, а в складних доменах застосовувати для моделювання Value Objects і Domain Services, що забезпечує формальну строгість та прозорість бізнес-логіки. Успішна інтеграція функціональної парадигми потребує розвитку компетентностей команди у сфері тестування, композиційного мислення та

розуміння функціональних абстракцій, що в сукупності дозволяє досягти високої якості, гнучкості та стійкості програмних систем.

Розроблений тестовий проєкт «Functional Counter» дозволив перевірити запропоновані рекомендації та здійснити оцінку відповідності визначеним критеріям і метрикам. Результати засвідчили середній рівень наближення до функціональної парадигми, що пояснюється використанням інструментів C# та WinForms, обраних з огляду на наявні знання та практичні навички здобувача. Отримані висновки узгоджуються з попередніми результатами дослідження та підтверджують їхню валідність. Перспективним напрямом подальших робіт є застосування мови Scala, яка завдяки глибокій інтеграції функціональних концепцій може забезпечити більш високий рівень відповідності критеріям та створити умови для формування архітектур із підвищеною строгістю та передбачуваністю.

ВИСНОВКИ

У межах виконаної магістерської роботи було комплексно досліджено вплив парадигм функціонального програмування на процеси проектування та архітектурного формування сучасного програмного забезпечення. Актуальність цього напрямку зумовлена тим, що вимоги до надійності, паралельності, масштабованості та передбачуваності програмних систем безпосередньо пов'язані з архітектурними підходами до керування станом, організації модулів та інтеграції асинхронних механізмів. На цьому тлі функціональна парадигма демонструє низку переваг, які роблять її дедалі важливішою частиною інженерії ПЗ, навіть у тих мовах і фреймворках, що традиційно базувалися на імперативних або об'єктно-орієнтованих підходах.

У першій частині роботи було проведено огляд сучасного стану функціонального програмування та визначено ключові принципи, які формують основу парадигми: незмінність даних, чисті функції, детермінованість обчислень, композиційність та декларативність. Дослідження засвідчило, що ці принципи є не лише теоретичними основами, а й практичними інструментами забезпечення архітектурної стійкості. Саме вони дозволяють уникати проблем, пов'язаних із прихованими побічними ефектами, складністю відстеження змін стану та неконтрольованими взаємодіями між об'єктами — проблемами, які стають особливо критичними у великих та розподілених системах.

Подальший аналіз показав, що між функціональним, імперативним та об'єктно-орієнтованим підходами існують суттєві відмінності, які проявляються передусім у моделях керування станом, механізмах організації модулів та підходах до забезпечення паралельності. Імперативний стиль зосереджений на послідовному змінному стані, тоді як ООП привносить концепцію інкапсульованих об'єктів зі своїми внутрішніми змінними. Функціональний стиль, навпаки, мінімізує мутабельність, що істотно знижує ризики виникнення гонок даних та конфліктів у багатопоточних режимах. Це

підкреслює принципову відмінність у підходах до проєктування архітектур та вказує на важливість функціональних принципів у сучасних умовах високонавантажених і розподілених систем.

Значну увагу було приділено аналізу використання функціональних підходів у гібридних мовах програмування, таких як Scala, Kotlin, JavaScript (у поєднанні з React/Redux) та C#. Проведене дослідження доводить, що ступінь “функціональності” мов безпосередньо впливає на їхню здатність підтримувати архітектури з низьким зчепленням, високою когерентністю та передбачуваністю. Scala демонструє найвищу відповідність функціональним принципам та широко застосовується у розподілених платформах, зокрема у реактивних системах (Akka) та великих даних (Spark). Kotlin інтегрує функціональні можливості у JVM-середовище та пропонує збалансованість між ООП і ФП. JavaScript у зв’язці з React/Redux упроваджує односторонній потік даних і чисті ред’юсери, що фактично переносить функціональні підходи в архітектуру користувацьких інтерфейсів. У C# LINQ і незмінні колекції формують декларативний стиль обробки даних навіть у межах об’єктної парадигми.

Окремим напрямом дослідження став вплив функціонального програмування на сучасні архітектурні шаблони, зокрема реактивні та подієві архітектури. Було встановлено, що чисті функції забезпечують детермінованість та передбачуваність у реактивних потоках даних, мінімізують побічні ефекти та створюють основу для ефективної паралелізації. Незмінність даних особливо важлива в контексті Event Sourcing, де події є незмінними фактами, а стан є похідним від чистої композиції цих подій. Це забезпечує прозорість історії змін, можливість відтворення стану та стійкість до збоїв. У Model-View-Update (MVU) частина View реалізується як чиста функція від стану, що робить інтерфейс легко прогнозованим і придатним до масштабування. Таким чином, функціональна парадигма суттєво впливає на архітектурні рішення у сучасних шаблонах, які потребують високої надійності та реактивності.

Важливим результатом роботи стало визначення системи критеріїв і метрик, які дають змогу оцінювати архітектурну якість програмних систем, реалізованих у функціональному стилі. Особливу увагу приділено метрикам зв'язності та зчеплення, прозорості та зрозумілості потоку даних, а також тестованості й верифікації функціональних модулів. Дослідження підтвердило, що функціональні системи демонструють значно спрощену тестованість завдяки відсутності прихованого стану та побічних ефектів. Метрики прозорості, такі як лінійність потоку даних і глибина композиції функцій, дозволяють точно оцінити структурну якість системи, що є важливим для довгострокової підтримуваності.

У практичній частині було розроблено рекомендації щодо доцільного впровадження функціонального стилю в інженерії ПЗ. Вони включають поступову інтеграцію ФП у гібридні системи, використання незмінних структур даних, ізоляцію побічних ефектів, застосування MVU, Event Sourcing та CQRS у відповідних доменах.

Для демонстрації практичної частини було створено тестовий Windows Forms застосунок у стилі MVU з використанням функціонального підходу. Структура проекту демонструє чітке відділення Model, Update, View та Core (Dispatch), забезпечуючи модульність, прозорість даних та чистоту логіки. Приклад підтверджує можливість використання функціонального стилю навіть у традиційно імперативних UI-фреймворках, що є важливою ілюстрацією універсальності досліджуваних принципів.

Узагальнюючи результати роботи, можна зробити висновок, що функціональна парадигма має суттєвий і багатогранний вплив на сучасні методи проектування та архітектуру програмного забезпечення. Вона забезпечує зменшення складності керування станом, підсилює передбачуваність системної поведінки, підвищує тестованість та надійність, а також полегшує побудову реактивних, подієвих і розподілених рішень. Інтеграція функціональних концепцій у гібридні мови та архітектурні патерни робить їх одним із найважливіших напрямів розвитку сучасної інженерії ПЗ.

Отримані теоретичні результати і практичні рекомендації можуть бути використані в проєктуванні складних програмних рішень, що потребують високої якості, масштабованості та стійкості до еволюційних змін.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Abelson H. Structure and interpretation of computer programs, second edition. Cambridge, Mass : MIT Press, 1996. 883 p.
2. Backus J. Can programming be liberated from the von Neumann style?. *Communications of the ACM*. 1978. Vol. 21, no. 8. P. 613–641. URL: <https://doi.org/10.1145/359576.359579> (date of access: 25.11.2025).
3. Bird R. Introduction to functional programming (prentice hall international series in computer science). Prentice Hall College Div, 1992. 293 p.
4. Bird R. J. An introduction to functional programming. Englewood Cliffs, N.J : Prentice-Hall, 1988. 293 p.
5. Boehm B. A spiral model of software development and enhancement. *ACM SIGSOFT software engineering notes*. 1986. Vol. 11, no. 4. P. 14–24. URL: <https://doi.org/10.1145/12944.12948> (date of access: 24.11.2025).
6. Brooks F. P. The Mythical Man-Month. *ACM SIGPLAN Notices*. 1975. Vol. 10, no. 6. P. 193. URL: <https://doi.org/10.1145/390016.808439> (date of access: 24.11.2025).
7. Church A. The calculi of lambda-conversion. New York : Kraus Reprint Corporation, 1965. 82 p.
8. Clements P., Bass L., Kazman R. Software architecture in practice. Addison-Wesley Longman, Incorporated, 2012. 600 p.
9. C M. R. Clean agile: back to basics. Prentice Hall, 2019. 240 p.
10. Curry H. B. The Undecidability of λ K-Conversion. *Foundations of mathematics*. Berlin, Heidelberg, 1969. P. 10–14. URL: https://doi.org/10.1007/978-3-642-86745-3_2 (date of access: 25.11.2025).
11. DeMarco T. Structured analysis and system specification. New York : Yourdon inc., 1979. 352 p.
12. Dijkstra E. W. Notes on structured programming. 2nd ed. Eindhoven, Netherlands : Technological University, Dept. of Mathematics, 1970. 84 p.
13. EDA(Event driven architecture). Akasai (Seoul/Korea). URL: https://akasai.space/architecture/about_event_driven_architecture/.

- 14.Event Sourcing pattern. *Microsoft Learn*.
URL: <https://learn.microsoft.com/en-us/azure/architecture/patterns/event-sourcing>.
- 15.Fowler M. Patterns of enterprise application architecture. Addison-Wesley Professional, 2002. 560 p.
- 16.Garlan D., Shaw M. An introduction to software architecture. *Advances in software engineering and knowledge engineering*. 1993. P. 1–39.
URL: https://doi.org/10.1142/9789812798039_0001 (date of access: 24.11.2025).
- 17.Hutton G. Higher-order functions for parsing. *Journal of functional programming*. 1992. Vol. 2, no. 3. P. 323–343.
URL: <https://doi.org/10.1017/s0956796800000411> (date of access: 25.11.2025).
- 18.Kanber S. C. CQRS (command query responsibility segregation) nedir?. *Medium*. URL: <https://sefikcankanber.medium.com/cqrs-command-query-responsibility-segregation-nedir-16b196376389>.
- 19.Landin P. J. The mechanical evaluation of expressions. *The computer journal*. 1964. Vol. 6, no. 4. P. 308–320.
URL: <https://doi.org/10.1093/comjnl/6.4.308> (date of access: 24.11.2025).
- 20.LISP 1. 5 programmer's manual / J. McCarthy et al. MIT Press, 2018. 112 p.
- 21.Martin M., Martin R. Agile principles, patterns, and practices in C#. Pearson Education, Limited.
- 22.Martin R. Robert C. Martin clean code collection (collection). Pearson Education, Limited.
- 23.McCarthy J. A basis for a mathematical theory of computation. 1962.
URL: <http://hdl.handle.net/1721.1/6099> (date of access: 24.11.2025).
- 24.McCarthy J. Recursive functions of symbolic expressions and their computation by machine, Part I. *Communications of the ACM*. 1960. Vol. 3, no. 4. P. 184–195.
URL: <https://doi.org/10.1145/367177.367199> (date of access: 24.11.2025).
- 25.Meijer E., Fokkinga M., Paterson R. Functional programming with bananas, lenses, envelopes and barbed wire. *Functional programming languages and computer architecture*. Berlin, Heidelberg, 1991. P. 124–

144. URL: https://doi.org/10.1007/3540543961_7 (date of access: 24.11.2025).
26. Milner R. A theory of type polymorphism in programming. *Journal of computer and system sciences*. 1978. Vol. 17, no. 3. P. 348–375. URL: [https://doi.org/10.1016/0022-0000\(78\)90014-4](https://doi.org/10.1016/0022-0000(78)90014-4) (date of access: 24.11.2025).
27. Myers G. J. The art of software testing. New York : John Wiley & Sons, Ltd., 2004.
28. Object-oriented software engineering: a use case driven approach / ed. by J. Ivar. [S.l.] : ACM Press, 1993. 528 p.
29. Okasaki C. Purely functional data structures. Cambridge, U.K : Cambridge University Press, 1998. 220 p.
30. Parnas D. L. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*. 1972. Vol. 15, no. 12. P. 1053–1058. URL: <https://doi.org/10.1145/361598.361623> (date of access: 24.11.2025).
31. Parnas D. L. Software aging. *16th international conference on software engineering*, Sorrento, Italy, 16 May 1994. URL: <https://doi.org/10.1109/icse.1994.296790> (date of access: 24.11.2025).
32. Reselman B. Understanding the 7 principles of functional programming. *TechTarget and Informa Tech's Digital Businesses Combine*. URL: <https://www.theserverside.com/tip/Understanding-the-principles-of-functional-programming>.
33. Singh Gill N. Introduction to functional programming languages. *XenonStack*. URL: <https://www.xenonstack.com/insights/functional-programming>.
34. Sperber M., Schlegel M. Evolution of functional UI paradigms. *FUNARCH '25: 3rd ACM SIGPLAN international workshop on functional software architecture*, Singapore Singapore. New York, NY, USA, 2025. P. 27–38. URL: <https://doi.org/10.1145/3759163.3760429> (date of access: 26.11.2025).
35. Sperber M. Six Years of FUNAR. *FUNARCH '25: 3rd ACM SIGPLAN international workshop on functional software architecture*, Singapore Singapore. New York, NY, USA, 2025. P. 21–26.

URL: <https://doi.org/10.1145/3759163.3760428> (date of access: 26.11.2025).

Додаток А

Порівняльна таблиця для гібридних мов програмування

Критерій	Scala	Kotlin	JavaScript (React/Redux)	C# (LINQ)
Ступінь наближення до чистої ФП	Дуже високий: глибока інтеграція функціональних концепцій, строгі типи, імутабельність за замовчуванням	Середній– високий: функціональні інструменти, але збереження ООП-основи	Середній: залежить від бібліотек (React/Redux, Ramda), мова сама по собі не є чисто функціонально ю	Середній: LINQ і лямбда- функції, але загальна модель залишається ООП
Підтримка незмінності (immutability)	Повноцінна: імутабельні структури є стандартом	Часткова: val, data-класи, додаткові бібліотеки незмінності	Відсутня на рівні мови, реалізується через Redux/Immer	Часткова: через Immutable Collections у .NET
Чисті функції та вищі функції	Підтримуютьс я повністю	Підтримуютьс я широко	Підтримуються (функції – перший клас), залежать від стилю розробника	Підтримуютьс я, особливо у LINQ

Критерій	Scala	Kotlin	JavaScript (React/Redux)	C# (LINQ)
Композиція функцій	Розвинена, підтримується на рівні мови та стандартної бібліотеки	Частково підтримується	Реалізується через бібліотеки (Ramda, RxJS)	Обмежена, частково присутня в LINQ
Підтримка декларативних моделей обчислення	Висока: Akka, Spark, колекції	Висока у колекціях та DSL	Дуже висока у React/Redux	Середня: LINQ сприяє декларативності
Придатність для побудови реактивних / розподілених систем	Дуже висока: Akka, потокові системи, масштабована конкурентність	Висока: корутини, Flow, Compose	Висока в контексті фронтенда; RxJS для реактивності	Середня: можливе використання TPL та async/await
Підтримка безпечної паралелізації	Природна: імутабельність та акторна модель	Покращена, але залежить від застосовуваних підходів	Досяжна через функціональні бібліотеки	Частково підтримується через async/await, але стан об'єктів лишається проблемою

Критерій	Scala	Kotlin	JavaScript (React/Redux)	C# (LINQ)
Здатність спрощувати архітектурний дизайн	Дуже висока: сприяє чистій архітектурі та слабкій зчепленості	Висока: полегшує модульність і чисті інтерфейси	Висока у фронтенді: односпрямований потік даних	Середня: залежить від стилю використання LINQ
Загальна оцінка функціональності	5	4	3	3

Додаток Б

Опис критеріїв та метрик оцінки архітектурних підходів
на відповідність функціональному стилю

Основний аспект	Ключовий критерій (що оцінюємо)	Основні метрики (чим вимірюємо)	Принцип функціонального стилю
1. Зв'язність (Cohesion) модулів	Функціональна однорідність (Ступінь логічної узгодженості функцій у модулі).	Міра чистоти модуля (M_{Pure}): Частка чистих функцій у модулі. Ступінь Використання Внутрішніх Типів (SUIT): Спільне використання специфічних доменних типів функціями модуля.	Чисті функції та принцип єдиної відповідальності.
2. Зчеплення (Coupling) модулів	Декларативна залежність (Залежність лише через явні входи/виходи функцій).	Індекс нечистого Зчеплення (ICI): Кількість точок доступу до зовнішнього змінного стану або I/O операцій. Кількість вхідних/вихідних посилок (NIV/NOV): Кількість зовнішніх модулів-викликачів та модулів, що викликаються.	Мінімізація побічних ефектів та детермінованість.
3. Прозорість та зрозумілість потоку даних	Референційна прозорість (Можливість заміни виразу його значенням).	Рівень декларативності (DR): Співвідношення описових (композиція, map) до процедурних (for, if/else) конструкцій. Глибина Композиції (CD): Середня кількість послідовних функцій у ланцюгу перетворень. Індекс побічних ефектів.	Композиційність та незмінність даних.
4. Тестованість модулів	Ізольована перевірка (Можливість тестування функції)	Частка чистих функцій (PFS): Загальна частка чистих функцій у кодовій базі. Коефіцієнт залежності	Детермінованість та відсутність прихованого стану.

Основний аспект	Ключовий критерій (що оцінюємо)	Основні метрики (чим вимірюємо)	Принцип функціонального стилю
	незалежно від середовища).	від середовища (EDR): Частка функцій, які вимагають ініціалізації зовнішніх ресурсів (БД, мережа).	
5. Верифікація коректності	Формальна гарантія коректності (Доведення властивостей функцій).	Покриття Property-Based тестами (PBT Cov.): Частка критичних властивостей, перевірених PBT. Рівень використання типових гарантій: Використання параметричного поліморфізму та складних типів (наприклад, монади).	Система типів та математична основа функцій.

```
namespace WinFormsApp13.Model
{
    public record AppModel(
        int Count,
        IReadOnlyList<string> History,
        DateTime? LastUpdated
    );

    namespace WinFormsApp13.Model
    {
        public enum Msg
        {
            Increment,
            Decrement,
            Reset
        }
    }

    using System;
    using System.Collections.Generic;
    using System.Linq;
    using System.Text;
    using System.Threading.Tasks;

    namespace WinFormsApp13.Model
    {
        public static class ModelFactory
        {
            public static AppModel Initial =>
                new AppModel(0, new List<string>(), null);
        }
    }

    using WinFormsApp13.Model;
    using WinFormsApp13.Update;

    namespace WinFormsApp13.Core
    {
        public static class Dispatcher
        {
            public static AppModel Dispatch(AppModel model, Msg msg)
            {
                return UpdateLogic.Update(model, msg);
            }
        }
    }

    using System;
    using System.Collections.Generic;
    using System.Linq;
    using System.Text;
    using System.Threading.Tasks;
```

```

using WinFormsApp13.Model;

namespace WinFormsApp13.Update
{
    public static class UpdateLogic
    {
        public static AppModel Update(AppModel model, Msg msg)
        {
            var now = DateTime.Now;
            switch (msg)
            {
                case Msg.Increment:
                {
                    int newCount = model.Count + 1;
                    var newHistory = model.History
                        .Append($"+1 → {newCount}")
                        .ToList();
                    return model with
                    {
                        Count = newCount,
                        History = newHistory,
                        LastUpdated = now
                    };
                }
                case Msg.Decrement:
                {
                    int newCount = model.Count - 1;
                    var newHistory = model.History
                        .Append($"-1 → {newCount}")
                        .ToList();
                    return model with
                    {
                        Count = newCount,
                        History = newHistory,
                        LastUpdated = now
                    };
                }
                case Msg.Reset:
                {
                    int newCount = 0;
                    var newHistory = model.History
                        .Append($"Reset → {newCount}")
                        .ToList();
                    return model with
                    {
                        Count = newCount,
                        History = newHistory,
                        LastUpdated = now
                    };
                }
                default:
                    return model;
            }
        }
    }
}

using WinFormsApp13.Model;
using WinFormsApp13.Core;
using WinFormsApp13.Update;

```

```

namespace WinFormsApp13
{
    public partial class Form1 : Form
    {
        private AppModel _model = Model.ModelFactory.Initial;
        public Form1()
        {
            InitializeComponent();
        }
        private void btnIncrement_Click2(object sender, EventArgs e)
        => Dispatch(Msg.Increment);

        private void btnDecrement_Click2(object sender, EventArgs e)
            => Dispatch(Msg.Decrement);

        private void btnReset_Click2(object sender, EventArgs e)
            => Dispatch(Msg.Reset);

        private void MainForm_Load2(object sender, EventArgs e)
            => Render(_model);
        private void Dispatch(Msg msg)
        {
            _model = Dispatcher.Dispatch(_model, msg);
            Render(_model);
        }
        private void Render(AppModel model)
        {
            lblCount.Text = model.Count.ToString();

            lstHistory.Items.Clear();
            foreach (var item in model.History)
                lstHistory.Items.Add(item);

            lblLastUpdated.Text = model.LastUpdated is null
                ? "Ще не оновлювалось"
                : $"Остання операція: {model.LastUpdated}";
        }
    }
}

```

```
using MatrixCalculator.Model;
using MatrixCalculator.Core;

namespace MatrixCalculator;

public partial class Form1 : Form
{
    private AppModel _model = MatrixHelpers.Initial(3);
    private Button btnAdd;
    private Button btnSub;
    private Button btnMul;
    private Button btnTransposeA;
    private Button btnTransposeB;
    private Button btnCompute;

    private DataGridView dgvA;
    private DataGridView dgvB;
    private DataGridView dgvResult;

    private NumericUpDown numSize;

    private Label lblError;
    public Form1()
    {
        InitializeComponent();
        InitializeDynamicControls();
        Render(_model);
    }
    private void btnAdd_Click(object sender, EventArgs e)
        => Apply(Msg.SetOperationAdd);

    private void btnSub_Click(object sender, EventArgs e)
        => Apply(Msg.SetOperationSub);

    private void btnMul_Click(object sender, EventArgs e)
        => Apply(Msg.SetOperationMul);

    private void btnTransposeA_Click(object sender, EventArgs e)
        => Apply(Msg.SetOperationTransposeA);

    private void btnTransposeB_Click(object sender, EventArgs e)
        => Apply(Msg.SetOperationTransposeB);

    private void btnCompute_Click(object sender, EventArgs e)
        => Apply(Msg.Compute);

    private void NumSize_ValueChanged(object sender, EventArgs e)
    {
        int newSize = (int)numSize.Value;
        // простий варіант - перестворити модель з новим розміром
        _model = MatrixHelpers.Initial(newSize);
        Render(_model);
    }

    private void DgvA_CellEndEdit(object sender, DataGridViewCellEventArgs e)
    {
        if (e.RowIndex < 0 || e.ColumnIndex < 0) return;
    }
}
```



```

        if (double.TryParse(dgvA[e.ColumnIndex,
e.RowIndex].Value?.ToString(), out double value))
        {
            var msg = new CellEditMsg(Msg.EditCellA, e.RowIndex,
e.ColumnIndex, value);
            Apply(msg);
        }
    }

    private void DgvB_CellEndEdit(object sender, DataGridViewCellEventArgs e)
    {
        if (e.RowIndex < 0 || e.ColumnIndex < 0) return;
        if (double.TryParse(dgvB[e.ColumnIndex,
e.RowIndex].Value?.ToString(), out double value))
        {
            var msg = new CellEditMsg(Msg.EditCellB, e.RowIndex,
e.ColumnIndex, value);
            Apply(msg);
        }
    }

    private void Apply(object msg)
    {
        _model = Dispatcher.Dispatch(_model, msg);
        Render(_model);
    }

    private void Form1_Load(object sender, EventArgs e)
    => Render(_model);

    private void InitializeDynamicControls()
    {
        // Загальні параметри форми
        this.Text = "Матричний калькулятор (MVU + FP)";
        this.Width = 1100;
        this.Height = 650;
        this.StartPosition = FormStartPosition.CenterScreen;

        // ===== NumericUpDown для розміру матриці =====
        numSize = new NumericUpDown
        {
            Minimum = 1,
            Maximum = 10,
            Value = _model.Size,
            Location = new System.Drawing.Point(20, 20),
            Width = 60
        };
        numSize.ValueChanged += NumSize_ValueChanged;
        this.Controls.Add(numSize);

        var lblSize = new Label
        {
            Text = "Розмір матриць n × n:",
            AutoSize = true,
            Location = new System.Drawing.Point(90, 22)
        };
        this.Controls.Add(lblSize);

        // ===== Кнопки операцій =====
        btnAdd = new Button
        {
            Text = "A + B",

```

```

        Location = new System.Drawing.Point(20, 60),
        Width = 80
    };
    btnAdd.Click += (s, e) => Apply(Msg.SetOperationAdd);
    this.Controls.Add(btnAdd);

    btnSub = new Button
    {
        Text = "A - B",
        Location = new System.Drawing.Point(110, 60),
        Width = 80
    };
    btnSub.Click += (s, e) => Apply(Msg.SetOperationSub);
    this.Controls.Add(btnSub);

    btnMul = new Button
    {
        Text = "A * B",
        Location = new System.Drawing.Point(200, 60),
        Width = 80
    };
    btnMul.Click += (s, e) => Apply(Msg.SetOperationMul);
    this.Controls.Add(btnMul);

    btnTransposeA = new Button
    {
        Text = "T(A) ",
        Location = new System.Drawing.Point(290, 60),
        Width = 80
    };
    btnTransposeA.Click += (s, e) =>
        Apply(Msg.SetOperationTransposeA);
    this.Controls.Add(btnTransposeA);

    btnTransposeB = new Button
    {
        Text = "T(B) ",
        Location = new System.Drawing.Point(380, 60),
        Width = 80
    };
    btnTransposeB.Click += (s, e) =>
        Apply(Msg.SetOperationTransposeB);
    this.Controls.Add(btnTransposeB);

    btnCompute = new Button
    {
        Text = "Обчислити",
        Location = new System.Drawing.Point(470, 60),
        Width = 120
    };
    btnCompute.Click += (s, e) => Apply(Msg.Compute);
    this.Controls.Add(btnCompute);

    // ===== DataGridView для матриці A =====
    var lblA = new Label
    {
        Text = "Матриця A",
        AutoSize = true,
        Location = new System.Drawing.Point(20, 100)
    };
    this.Controls.Add(lblA);

    dgvA = new DataGridView

```

```

        {
            Location = new System.Drawing.Point(20, 120),
            Width = 320,
            Height = 200,
            AllowUserToAddRows = false,
            AllowUserToDeleteRows = false,
            RowHeadersWidth = 50,
            ColumnHeadersHeightSizeMode =
DataGridViewColumnHeadersHeightSizeMode.AutoSize
        };
        dgvA.CellEndEdit += DgvA_CellEndEdit;
        this.Controls.Add(dgvA);

        // ===== DataGridView для матриці B =====
        var lblB = new Label
        {
            Text = "Матриця B",
            AutoSize = true,
            Location = new System.Drawing.Point(380, 100)
        };
        this.Controls.Add(lblB);

        dgvB = new DataGridView
        {
            Location = new System.Drawing.Point(380, 120),
            Width = 320,
            Height = 200,
            AllowUserToAddRows = false,
            AllowUserToDeleteRows = false,
            RowHeadersWidth = 50,
            ColumnHeadersHeightSizeMode =
DataGridViewColumnHeadersHeightSizeMode.AutoSize
        };
        dgvB.CellEndEdit += DgvB_CellEndEdit;
        this.Controls.Add(dgvB);

        // ===== DataGridView для результату =====
        var lblResult = new Label
        {
            Text = "Результат",
            AutoSize = true,
            Location = new System.Drawing.Point(740, 100)
        };
        this.Controls.Add(lblResult);

        dgvResult = new DataGridView
        {
            Location = new System.Drawing.Point(740, 120),
            Width = 320,
            Height = 200,
            AllowUserToAddRows = false,
            AllowUserToDeleteRows = false,
            ReadOnly = true,
            RowHeadersWidth = 50,
            ColumnHeadersHeightSizeMode =
DataGridViewColumnHeadersHeightSizeMode.AutoSize
        };
        this.Controls.Add(dgvResult);

        // ===== Label для повідомлень про помилки =====
        lblError = new Label
        {
            Text = "",

```

```

        AutoSize = false,
        Width = 1040,
        Height = 40,
        Location = new System.Drawing.Point(20, 340),
        ForeColor = System.Drawing.Color.DarkRed
    };
    this.Controls.Add(lblError);
}
}

using MatrixCalculator.Model;

namespace MatrixCalculator;

public partial class Form1
{
    private void Render(AppModel model)
    {
        // 1. Відображення матриць A, B, Result у DataGridView
        RenderMatrix(dgvA, model.MatrixA);
        RenderMatrix(dgvB, model.MatrixB);
        RenderMatrix(dgvResult, model.Result);

        // 2. Відображення розміру
        numSize.Value = model.Size;

        // 3. Відображення помилки, якщо є
        lblError.Text = string.IsNullOrEmpty(model.ErrorMessage)
            ? ""
            : model.ErrorMessage;
    }

    private void RenderMatrix(DataGridView grid, double[,] matrix)
    {
        int n = matrix.GetLength(0);
        grid.RowCount = n;
        grid.ColumnCount = n;

        for (int i = 0; i < n; i++)
        {
            grid.Rows[i].HeaderCell.Value = i.ToString();
            for (int j = 0; j < n; j++)
            {
                grid.Columns[j].HeaderText = j.ToString();
                grid[j, i].Value = matrix[i, j];
            }
        }
    }
}

namespace MatrixCalculator.Model;

public enum MatrixOperation
{
    None,
    Add,
    Subtract,
    Multiply,
    TransposeA,

```

```

        TransposeB
    }

    public record AppModel(
        int Size,
        double[,] MatrixA,
        double[,] MatrixB,
        double[,] Result,
        MatrixOperation Operation,
        string? ErrorMessage
    );

    namespace MatrixCalculator.Model;

    public static class MatrixHelpers
    {
        public static double[,] CreateZeroMatrix(int n)
        {
            return new double[n, n];
        }

        public static AppModel Initial(int size = 3)
        {
            var a = CreateZeroMatrix(size);
            var b = CreateZeroMatrix(size);
            var r = CreateZeroMatrix(size);

            return new AppModel(
                Size: size,
                MatrixA: a,
                MatrixB: b,
                Result: r,
                Operation: MatrixOperation.None,
                ErrorMessage: null
            );
        }
    }

    namespace MatrixCalculator.Model;

    public enum Msg
    {
        ChangeSize,           // змінено n
        EditCellA,           // змінено елемент A[i,j]
        EditCellB,           // змінено елемент B[i,j]
        SetOperationAdd,
        SetOperationSub,
        SetOperationMul,
        SetOperationTransposeA,
        SetOperationTransposeB,
        Compute,             // натиснули "Обчислити"
        ClearError           // очистити повідомлення про помилку
    }

    // Для передачі координат клітинки з View до Update:
    public readonly record struct CellEditMsg(
        Msg Kind,
        int Row,
        int Col,
        double NewValue
    );

```

```

using MatrixCalculator.Domain;
using MatrixCalculator.Model;

namespace MatrixCalculator.Update;

public static class UpdateLogic
{
    public static AppModel Update(AppModel model, object message)
    {
        switch (message)
        {
            case Msg msg:
                return UpdateSimple(model, msg);

            case CellEditMsg cellMsg:
                return UpdateCell(model, cellMsg);

            default:
                return model;
        }
    }

    private static AppModel UpdateSimple(AppModel model, Msg msg)
    {
        return msg switch
        {
            Msg.ChangeSize => model, // обробимо окремо з параметром (краще через окремий msg тип)
            Msg.SetOperationAdd => model with { Operation = MatrixOperation.Add, ErrorMessage = null },
            Msg.SetOperationSub => model with { Operation = MatrixOperation.Subtract, ErrorMessage = null },
            Msg.SetOperationMul => model with { Operation = MatrixOperation.Multiply, ErrorMessage = null },
            Msg.SetOperationTransposeA => model with { Operation = MatrixOperation.TransposeA, ErrorMessage = null },
            Msg.SetOperationTransposeB => model with { Operation = MatrixOperation.TransposeB, ErrorMessage = null },
            Msg.ClearError => model with { ErrorMessage = null },
            Msg.Compute => Compute(model),
            _ => model
        };
    }

    private static AppModel UpdateCell(AppModel model, CellEditMsg msg)
    {
        var n = model.Size;
        var mA = model.MatrixA;
        var mB = model.MatrixB;

        try
        {
            if (msg.Kind == Msg.EditCellA)
            {
                var clone = MatrixOps.Clone(mA);
                clone[msg.Row, msg.Col] = msg.NewValue;
                return model with { MatrixA = clone, ErrorMessage = null };
            }
            else if (msg.Kind == Msg.EditCellB)
            {
                var clone = MatrixOps.Clone(mB);
                clone[msg.Row, msg.Col] = msg.NewValue;
                return model with { MatrixB = clone, ErrorMessage = null };
            }
        }
    }
}

```

```

    }
}
catch (Exception ex)
{
    return model with { ErrorMessage = $"Помилка редагування
клітинки: {ex.Message}" };
}

return model;
}

private static AppModel Compute(AppModel model)
{
    try
    {
        var op = model.Operation;
        var a = model.MatrixA;
        var b = model.MatrixB;

        double[,] result = op switch
        {
            MatrixOperation.Add => MatrixOps.Add(a, b),
            MatrixOperation.Subtract => MatrixOps.Sub(a, b),
            MatrixOperation.Multiply => MatrixOps.Mul(a, b),
            MatrixOperation.TransposeA => MatrixOps.Transpose(a),
            MatrixOperation.TransposeB => MatrixOps.Transpose(b),
            _ => MatrixHelpers.CreateZeroMatrix(model.Size)
        };

        return model with
        {
            Result = result,
            ErrorMessage = null
        };
    }
    catch (Exception ex)
    {
        return model with
        {
            ErrorMessage = $"Помилка обчислення: {ex.Message}"
        };
    }
}

using MatrixCalculator.Model;
using MatrixCalculator.Update;

namespace MatrixCalculator.Core;

public static class Dispatcher
{
    public static AppModel Dispatch(AppModel model, object msg)
    => UpdateLogic.Update(model, msg);
}

```