ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ
НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ
ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ
КАФЕДРА ІНЖЕНЕРІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

КВАЛІФІКАЦІЙНА РОБОТА

на тему: «Метод підвищення масштабованості багатошарових .NET-систем на основі сучасних патернів проєктування та моніторингу»

на здобуття освітнього ступеня магістра
зі спеціальності 121 Інженерія програмного забезпечення
освітньо-професійної програми «Інженерія програмного забезпечення»

Кваліфікаційна робота містить результати власних досліджень. Використання ідей, результатів і текстів інших авторів мають посилання на відповідне джерело

_____ Владислав ПРИТАМАННИЙ
(підпис)

Виконав: здобувач вищої освіти групи ПДМ-63
Владислав ПРИТАМАННИЙ

Керівник: Віталій ЗАЛИВА
доктор філософії (PhD)

Рецензент: _____

Київ 2026

ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ

Навчально-науковий інститут інформаційних технологій

Кафедра Інженерії програмного забезпечення

Ступінь вищої освіти Магістр

Спеціальність 121 Інженерія програмного забезпечення

Освітньо-професійна програма «Інженерія програмного забезпечення»

ЗАТВЕРДЖУЮ
Завідувач кафедри
Інженерії програмного забезпечення
_____ Ірина ЗАМРІЙ
«_____» _____ 2025 р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ

Притаманному Владиславу Вадимовичу

1. Тема кваліфікаційної роботи: «Метод підвищення масштабованості багатошарових .NET-систем на основі сучасних патернів проєктування та моніторингу»

керівник кваліфікаційної роботи Віталій ЗАЛИВА, доктор філософії(PhD), ст. викладач кафедри ІПЗ,

затверджені наказом Державного університету інформаційно-комунікаційних технологій від «30» жовтня 2025 р. № 467.

2. Строк подання кваліфікаційної роботи «19» грудня 2025 р.

3. Вихідні дані до кваліфікаційної роботи: науково-технічна література, опис та вимоги до багатошарових програмних систем, сучасні архітектурні патерни та шаблони проєктування, методи та засоби моніторингу, профілювання й аналізу продуктивності .NET-систем.

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити)

1. Дослідження теоретичних основ багатошарової архітектури .NET-систем та аналіз існуючих реалізацій патерну Specification.
2. Аналіз і моделювання методів підвищення масштабованості багатошарових .NET-систем на основі патерну Specification та розробка запропонованого методу.
3. Аналіз результатів застосування запропонованого методу та формування практичних рекомендацій щодо його використання.

5. Перелік ілюстративного матеріалу: *презентація*
   1. Обмеження сучасних методів аналізу масштабованості.
   2. Математична модель визначення структурної складності специфікацій.
   3. Математична модель визначення коефіцієнту ефективності специфікацій.
   4. Метод підвищення масштабованості.
   5. Блок-схема роботи методу.
   6. Результати моделювання.
   7. MVP реалізації та автоматизація аналізу специфікацій.

6. Дата видачі завдання «31» жовтня 2025 р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів кваліфікаційної роботи	Строк виконання етапів роботи	Примітка
1	Аналіз наявної науково-технічної літератури	31.10-07.11.2025	
2	Визначення напрямків удосконалення	10.11-14.11.2025	
3	Дослідження існуючих методів побудови запитів у багатошарових .NET-архітектурах та аналіз принципу їх роботи	15.11-17.11.2025	
4	Аналіз принципу роботи існуючих методів та інструментів, що використовуються для створення специфікацій у доменному та інфраструктурному рівнях	18.11-19.11.2025	
5	Розробка метод збору метрик виконання специфікацій та механізму автоматичного моніторингу й оцінювання ефективності специфікацій	20.11-24.11.2025	
6	Реалізація модулю контролю архітектурних правил	25.11-29.11.2024	

7	Проведення експериментального дослідження	01.12-05.12.2024	
8	Оформлення роботи: вступ, висновки, реферат	06.12-08.12.2024	
9	Розробка демонстраційних матеріалів	09.12-12.12.2024	
10	Попередній захист роботи	13.12-19.12.2024	

Здобувач вищої освіти _____ Владислав ПРИТАМАННИЙ
(підпис)

Керівник
кваліфікаційної роботи _____ Віталій ЗАЛИВА
(підпис)

РЕФЕРАТ

Текстова частина кваліфікаційної роботи на здобуття освітнього ступеня магістра: 88 стор., 6 табл., 7 рис., 30 джерел.

Мета роботи – підвищення масштабованості багатошарових .NET-систем на основі сучасних патернів проєктування та моніторингу.

Об'єкт дослідження – процес забезпечення масштабованості багатошарових .NET-систем, із використанням шаблону «Специфікація» та інструментів моніторингу

Предмет дослідження – методи аналізу структури специфікацій і метрик виконання у багатошарових .NET-системах, що використовуються для розроблення методу підвищення їх масштабованості на основі сучасних патернів проєктування та моніторингу.

У роботі проведено аналіз проблем масштабованості багатошарових .NET-систем, зокрема обмежень традиційних підходів до проєктування та використання патерна «Specification» без урахування аспектів продуктивності та моніторингу. Обґрунтовано необхідність удосконалення існуючих рішень шляхом поєднання сучасних патернів проєктування, принципів чистої архітектури та інструментів збору й аналізу метрик виконання.

Досліджено архітектурні підходи до побудови масштабованих багатошарових .NET-систем, проаналізовано особливості використання патерна «Specification» у поєднанні з репозиторіями та ORM-інструментами. Розглянуто сучасні засоби моніторингу, логування та профілювання, що дозволяють виявляти вузькі місця продуктивності на різних рівнях архітектури.

Запропоновано метод підвищення масштабованості багатошарових .NET-систем, який ґрунтується на аналізі структури специфікацій, контролі їх складності та використанні метрик виконання для прийняття архітектурних рішень. Реалізовано програмне рішення у вигляді розширення для .NET-застосунків, що

забезпечує аудит виконання специфікацій, збір статистики запитів та інтеграцію з інструментами моніторингу.

Проведено експериментальне дослідження ефективності запропонованого методу на прикладі тестового багатошарового .NET-застосунку. Результати експериментів підтвердили зниження часу виконання запитів, підвищення керованості архітектури та покращення масштабованості системи при зростанні навантаження.

Практична цінність роботи полягає у можливості застосування запропонованого методу під час проєктування та супроводу багатошарових .NET-систем з метою підвищення їх продуктивності та масштабованості. Результати дослідження можуть бути використані розробниками програмного забезпечення, архітекторами .NET-систем, а також у навчальному процесі підготовки фахівців у галузі програмної інженерії.

Подальші дослідження можуть бути спрямовані на розширення функціональних можливостей методу, автоматизацію аналізу архітектурних рішень, а також інтеграцію з хмарними платформами та інструментами спостережуваності.

КЛЮЧОВІ СЛОВА: .NET, БАГАТОШАРОВА АРХІТЕКТУРА, МАСШТАБОВАНІСТЬ, SPECIFICATION, ПАТЕРНИ ПРОЄКТУВАННЯ, МОНІТОРИНГ, ПРОДУКТИВНІСТЬ, АРХІТЕКТУРА ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ.

ABSTRACT

Text part of the master's qualification work: 88 pages, 6 table, 7 pictures, 30 sources.

The purpose of the work is to improve the scalability of multilayer .NET systems based on modern design patterns and monitoring approaches.

The object of the study is the process of ensuring scalability of multilayer .NET systems using the Specification pattern and monitoring tools.

The subject of the study is methods for analyzing the structure of specifications and execution metrics in multilayer .NET systems, which are used to develop a method for improving their scalability based on modern design patterns and monitoring.

The thesis analyzes the scalability issues of multilayer .NET systems, including the limitations of traditional design approaches and the use of the Specification pattern without considering performance and monitoring aspects. The necessity of improving existing solutions through the combination of modern design patterns, clean architecture principles, and tools for collecting and analyzing execution metrics is substantiated.

Architectural approaches to building scalable multilayer .NET systems are investigated, and the specifics of using the Specification pattern in combination with repositories and ORM tools are analyzed. Modern monitoring, logging, and profiling tools that enable the identification of performance bottlenecks at different architectural layers are reviewed.

A method for improving the scalability of multilayer .NET systems is proposed, which is based on analyzing the structure of specifications, controlling their complexity, and using execution metrics to support architectural decision-making. A software solution is implemented as an extension for .NET applications, providing specification execution auditing, query statistics collection, and integration with monitoring tools.

An experimental study of the effectiveness of the proposed method is conducted using a test multilayer .NET application. The experimental results confirm a reduction in query execution time, improved architectural maintainability, and enhanced system

scalability under increasing load. The practical value of the thesis lies in the possibility of applying the proposed method during the design and maintenance of multilayer .NET systems to improve their performance and scalability. The research results can be used by software developers, .NET system architects, and in the educational process for training specialists in the field of software engineering.

Further research may focus on extending the functional capabilities of the method, automating architectural analysis, and integrating the solution with cloud platforms and observability tools.

KEYWORDS: .NET, MULTILAYER ARCHITECTURE, SCALABILITY, SPECIFICATION, DESIGN PATTERNS, MONITORING, PERFORMANCE, SOFTWARE ARCHITECTURE.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ

ORM – Object-Relational Mapping (об'єктно-реляційне відображення)

SQL – Structured Query Language (мова структурованих запитів)

CI/CD – Continuous Integration/Continuous Delivery (безперервна інтеграція/безперервна доставка)

JSON – JavaScript Object Notation (текстовий формат обміну даними)

DRY – Don’t Repeat Yourself (не повторюйся)

SRP – Single Responsibility Principle (принцип єдиної відповідальності)

DDD – Domain-Driven Design (предметно-орієнтоване проектування)

LINQ – Language-Integrated Query (інтегрована мова запитів)

EF – Entity Framework

ВСТУП

Актуальність теми дослідження зумовлена зростаючою складністю сучасних програмних систем і необхідністю забезпечення високої якості, масштабованості та гнучкості архітектурних рішень. У процесі розробки корпоративних .NET-застосунків широке застосування отримує патерн «Specification», який дає змогу інкапсулювати бізнес-логіку запитів та відокремити її від інфраструктурних шарів.

Проте стрімкий розвиток індустрії та збільшення вимог до продуктивності, прозорості та керованості програмних рішень створюють нові виклики. Сучасним застосункам необхідно забезпечувати не лише коректність бізнес-логіки, але й можливість відстеження ефективності роботи специфікацій, аналізу витрат ресурсів, виявлення потенційних вузьких місць та автоматизації контролю архітектурних обмежень. Існуючі методи не охоплюють цих завдань у повному обсязі, що створює наукову та практичну проблему, пов’язану з необхідністю удосконалення методів роботи зі специфікаціями у .NET-проєктах.

Метою магістерської кваліфікаційної роботи є підвищення масштабованості багатошарових .NET-систем на основі сучасних патернів проєктування та моніторингу.

Для досягнення поставленої мети визначено такі завдання дослідження:

1. Провести аналіз існуючих методів побудови запитів у багатошарових .NET-архітектурах та визначити їхні переваги й недоліки.
2. Дослідити принципи роботи існуючих методів та інструменти, що використовуються для створення специфікацій у доменному та інфраструктурному рівнях.
3. Розробити метод збору метрик виконання специфікацій, який включає замір часу, складності виразів, кількості SQL-операцій та інших показників.
4. Створити механізм автоматичного моніторингу й оцінювання ефективності специфікацій з використанням GitHub Actions

5. Реалізувати модуль контролю архітектурних правил, що дає змогу виявляти порушення залежностей між шарами та рекомендувати способи їх усунення.
6. Провести експериментальне дослідження, порівнявши показники продуктивності та якості архітектури до та після впровадження розробленого методу.
7. Оцінити результати дослідження та визначити вплив запропонованого підходу на підтримуваність, масштабованість і якість .NET-застосунків.

Об’єктом дослідження є процес забезпечення масштабованості багатошарових .NET-систем, із використанням шаблону «Specification» та інструментів моніторингу.

Предметом дослідження є методи аналізу структури специфікацій і метрик виконання у багатошарових .NET-системах, що використовуються для розроблення методу підвищення їх масштабованості на основі сучасних патернів проєктування та моніторингу.

Наукова новизна одержаних результатів полягає у тому, що:

1. Вперше запропоновано метод інтегрованого збору та оцінювання метрик виконання специфікацій, який комплексно аналізує часові показники, складність запитів і кількість звернень до БД.
2. Удосконалено процес роботи специфікацій через поєднання аналізу метрик із перевіркою відповідності архітектурним правилам, що забезпечує вищу керованість та оптимальність застосунків.
3. Запропоновано формулу оцінювання ефективності специфікації, яка надає можливість кількісно оцінити її поведінку на основі сукупності метрик.

1 ТЕОРЕТИЧНІ ОСНОВИ БАГАТОШАРОВОЇ АРХІТЕКТУРИ ТА ПАТЕРНУ «SPECIFICATION»

1.1 Аналіз недоліків існуючих методів, у яких реалізовано патерн «Specification»

Патерн «Specification» є важливим інструментом у сучасних багатошарових .NET-системах. Його основна функція полягає у формалізації умов вибірки, фільтрації даних та управлінні бізнес-логікою, що дозволяє відокремити правила від доменної моделі. Така відокремленість сприяє повторному використанню коду, підвищує тестованість системи та полегшує підтримку бізнес-логіки в умовах змін.

Концептуально «Specification» передбачає, що умови можуть бути представлені як окремі об’єкти або вирази, які можна комбінувати та повторно використовувати у різних частинах системи. Наприклад, у .NET часто використовують інтерфейс, зображений на рис.1.1.

```
4 references
public interface ISpecification<T>
{
    8 references
    Expression<Func<T, bool>> ToExpression();
    4 references
    bool IsSatisfiedBy(T entity);
}
```

Рис. 1.1 Інтерфейс для реалізацію патерну «Специфікація»

Ця модель дозволяє:

- комбінувати специфікації через логічні оператори (AND, OR, NOT);
- повторно використовувати правила у різних модулях системи;
- відокремлювати бізнес-логіку від шарів доступу до даних, забезпечуючи модульність та чистоту архітектури.

Незважаючи на концептуальні переваги, у практичних проєктах «Specification» реалізується з певними обмеженнями. Вони стосуються:

- масштабованості системи;
- відсутності моніторингу та збору метрик продуктивності;
- дублювання бізнес-логіки;
- складності підтримки та розширення системи;
- генерації неефективних SQL-запитів.

Саме ці обмеження створюють необхідність удосконалення підходів до реалізації «Specification» у багатошарових .NET-системах та визначають напрямок розробки нового методу, що забезпечить масштабованість, прозорість і ефективний контроль за виконанням бізнес-правил.

У великих багатошарових .NET-системах, що застосовують патерн «Specification», однією з ключових проблем є низька масштабованість існуючих підходів. Зі збільшенням кількості специфікацій та їхньої складності виникають численні труднощі, які ускладнюють підтримку та розвиток системи.

У великих проєктах кількість специфікацій може сягати сотень. Відсутність централізованих механізмів класифікації або групування призводить до дублювання логіки, коли одна й та сама умова реалізується у різних частинах системи. Це порушує принцип DRY, підвищує технічний борг і ускладнює подальшу підтримку бізнес-логіки.

Хоча патерн «Specification» теоретично дозволяє комбінувати умови через логічні оператори (AND, OR, NOT), практичні реалізації часто обмежені. Складні специфікації, що включають кілька шарів умов та залежності від зовнішніх джерел даних, важко інтегрувати без дублювання коду або порушення принципів модульності.

Через обмеження існуючих реалізацій частина логіки фільтрації або обчислень переноситься у сервіси, контролери або репозиторії. Це призводить до порушення принципу SRP, що негативно впливає на підтримку системи та її повторне використання.

Зі збільшенням кількості специфікацій зростає складність формованих SQL-запитів. Типові ORM, такі як Entity Framework Core або NHibernate, формують запити на основі виразів LINQ. Складні специфікації генерують надмірні JOIN-операції, підзапити та дублюючі умови, що може призвести до деградації продуктивності системи при високому навантаженні.

Іншим суттєвим обмеженням є відсутність інтегрованого моніторингу та збору метрик. У масштабних системах критично важливо відстежувати:

- які специфікації виконуються;
- як вони впливають на продуктивність;
- які запити формують «вузькі місця».

Як наслідок, розробники та адміністративний персонал не можуть своєчасно виявляти «вузькі місця», планувати оптимізацію або передбачати вплив нових правил на систему в цілому.

Таблиця 1.1

Основні недоліки існуючих реалізацій патерну «Specification»

Недолік	**Опис**
Відсутність трасування виконання специфікацій	Розробник не може відстежити, яка саме специфікація ініціювала запит, спричинила затримку або стала джерелом проблеми.
Відсутність збору метрик продуктивності	Бібліотеки не надають інформації про час виконання, кількість оброблених записів та інші ключові метрики, що ускладнює оцінку продуктивності та визначення «вузьких місць».
Неможливість оцінки складності специфікацій	Відсутні механізми автоматичного аналізу складності специфікацій, що не дозволяє виявляти важкі або неоптимальні запити.
Відсутність рекомендацій щодо покращення специфікацій на основі метрик	Навіть якщо певні дані про продуктивність збираються сторонніми інструментами, сучасні реалізації не пропонують механізмів для генерації рекомендацій чи підказок щодо оптимізації складних або повільних специфікацій. Розробник самостійно інтерпретує метрики, що підвищує ризик помилок та ускладнює оптимізацію.

Існуючі реалізації «Specification» часто не інтегруються з принципами чистої архітектури. Основні недоліки включають:

1. Відсутність централізованого контролю над специфікаціями. Розробники створюють нові специфікації замість повторного використання існуючих, що ускладнює підтримку системи.
2. Розпорошення правил по шарах системи. Специфікації застосовуються у різних модулях без чіткого зв’язку з бізнес-сценаріями, що призводить до дублювання логіки.
3. Немає інтеграції з сучасними системами моніторингу та логування. Відсутність механізмів спостережуваності ускладнює оцінку ефективності специфікацій та їхнього впливу на продуктивність.

Всі ці фактори суттєво обмежують можливості масштабування та розвитку системи, особливо у великих корпоративних проєктах з високим навантаженням.

Існуючі методи реалізації патерну «Specification» демонструють значні труднощі у сфері масштабованості, управління складністю та забезпечення узгодженості системи. Зокрема, з розвитком проєкту збільшується кількість окремих специфікацій, що ускладнює їхнє централізоване управління. Кожна нова специфікація може вимагати створення додаткових класів, методів і логічних умов, що веде до постійного збільшення обсягу коду та зростання когнітивного навантаження на розробників.

Також спостерігається поширення дублювання бізнес-правил, коли схожі умови повторюються у різних частинах системи. Це знижує ефективність підтримки, підвищує ризик помилок при оновленні логіки та ускладнює тестування. Відсутність централізованої системи контролю призводить до того, що будь-яка зміна в одній специфікації може непередбачувано вплинути на інші частини програми, що збільшує ймовірність появи прихованих багів.

Критичною проблемою є те, що існуючі специфікації не завжди оптимізовані для роботи з великими обсягами даних. При масштабуванні системи зростає час обробки запитів, а генерація складних умов у ORM часто призводить до створення неефективних SQL-запитів із численними підзапитами та повторюваними умовами.

Це особливо актуально для багатошарових систем, де специфікації застосовуються на рівні доменних об’єктів і передаються у шари доступу до даних. Ускладнена структура запитів знижує продуктивність, підвищує навантаження на сервери баз даних та може призвести до непередбачуваних затримок у виконанні бізнес-логіки.

Існуючі методи не забезпечують видимості взаємозв’язків між специфікаціями та іншими компонентами системи. Це призводить до того, що внесення змін стає ризикованим: складно передбачити, які модулі або бізнес-сценарії будуть залучені при оновленні певної умови.

Відсутність прозорості ускладнює підтримку великих проєктів та підвищує технічний борг. Кожна нова специфікація потребує ретельного аналізу її взаємозв’язків, що значно збільшує час на планування та тестування змін.

У контексті розвитку сучасних архітектурних стилів, таких як мікросервісна архітектура, модульно-орієнтовані системи або DDD-орієнтовані доменні моделі, класичні реалізації патерну «Specification» виявляють низку структурних обмежень. Більшість існуючих реалізацій орієнтовані на монолітні рішення або системи з явною централізованою моделлю домену, що робить їх менш сумісними з підходами, де логіка розподіляється між автономними сервісами.

Специфікації, які розміщуються у спільному доменному шарі, створюють ризик надмірної залежності між компонентами та ускладнюють модульність системи. У мікросервісних середовищах, де кожен сервіс повинен мати власну модель та власні засоби реалізації бізнес-логіки, застосування централізованих специфікацій суперечить принципам автономності та незалежного життєвого циклу. Це обмежує здатність системи еволюціонувати, масштабуватися та розгортатися незалежно для кожного компонента.

Одним із суттєвих недоліків є відсутність ефективного механізму стандартизації структури специфікацій. У різних проєктах та командах розробники використовують різні варіації патерну, що призводить до фрагментації та різнотипного формування логіки. Це ускладнює інтеграцію, рев’ю коду та адаптацію нових членів команди до існуючої системи. Відсутність уніфікованої

моделі також впливає на масштабованість — логіка, яка з часом має розширюватися, виявляється обмеженою різними підходами, що накопичувалися в проєкті.

Повторне використання специфікацій в існуючих підходах часто є неефективним через вузьку прив’язку до конкретних доменних моделей або ORM-провайдерів. У таких умовах одна й та сама бізнес-умова не може бути повторно використана у різних модулях або сервісах без модифікації коду, що суперечить базовій ідеї патерну. Це знижує рівень уніфікації бізнес-логіки та сприяє дублюванню функціональності.

Існуючі методи реалізації не передбачають засобів формальної перевірки коректності специфікацій чи їхньої поведінки у різних контекстах. Внаслідок цього бізнес-правила, формалізовані у вигляді специфікацій, можуть бути неоднозначними, суперечливими або надто деталізованими. Відсутність інструментів статичного аналізу або системи атрибутів, які могли б допомогти автоматично виявляти потенційні конфлікти або дублювання, призводить до погіршення якості коду та збільшення кількості дефектів у бізнес-логіці.

Також у багатьох реалізаціях специфікацій недостатньо уваги приділяється гарантуванню їхньої коректності при змінах у доменній моделі. Оновлення сутностей або трансформація бізнес-процесів нерідко призводять до порушення умов специфікацій. Це створює додаткове навантаження на тестування та збільшує ймовірність непередбачуваних побічних ефектів у роботі системи.

Ще одним суттєвим недоліком є недостатній рівень тестованості специфікацій у традиційних реалізаціях. Через тісний зв’язок із доменною моделлю та використання складних конструкцій LINQ, виразів або кастомних фільтрів специфікації часто стають малопридатними для ізольованого модульного тестування. Це змушує розробників або жертвувати якістю тестів, або створювати труднощі у підтримці тестового середовища.

Складні специфікації — з великою кількістю умов, вкладених обмежень або логічних операторів — ще більше ускладнюють тестування та вимагають додаткових інструментів. Через відсутність механізмів автоматизованої валідації

тестування стає трудомістким та не завжди ефективним, що погіршує контроль якості продукту в довгостроковій перспективі.

Оскільки традиційні специфікації не обладнані інструментами моніторингу, метриками та механізмами автоматичного аналізу продуктивності, їхня підтримка потребує ручного аналізу, численних рев'ю та складних профілювань. При масштабуванні системи ці витрати різко збільшуються, впливаючи як на економічну ефективність розробки, так і на оперативність реагування на проблеми.

Відсутність прозорості та інструментів для оцінки складності специфікацій призводить до накопичення технічного боргу. Специфікації стають дедалі громіздкішими, втрачають свій початковий рівень узагальненості та перетворюються на джерело архітектурної нестабільності. У великих системах це перетворюється на системну проблему, яка ускладнює еволюцію проєкту та може призвести до необхідності повного рефакторингу шару доступу до даних або бізнес-логіки.

Одним із ключових недоліків традиційних реалізацій патерну «Specification» є відсутність будь-яких механізмів адаптивної оптимізації, тобто таких, що дозволяють системі автоматично підлаштовувати виконання специфікацій залежно від навантаження, структури даних або частоти використання бізнес-правил. У сучасних високонавантажених системах подібні можливості є принципово важливими для гарантування стабільності та передбачуваності продуктивності.

Існуючі підходи застосовують специфікації як статичні правила, які не змінюються в залежності від контексту. Це означає, що система не враховує:

- актуальний стан бази даних;
- частоту виконання конкретних специфікацій;
- типові навантаження у певні періоди;
- накопичення проміжних результатів або кешованих обчислень.

Відсутність таких механізмів призводить до того, що навіть тривіальні специфікації можуть створювати надлишкові операції над даними, а складні специфікації стають вузьким місцем, яке суттєво знижує масштабованість та швидкодію всієї системи.

Ці недоліки демонструють, що традиційні реалізації «Specification» суттєво обмежують можливості побудови масштабованих, спостережуваних та адаптивних .NET-систем, що вимагає розробки нового методу, спрямованого на усунення зазначених проблем.

1.2 Обґрунтування необхідності вдосконалення наявних методів

Хоча патерн «Specification» уже тривалий час використовується як фундаментальний інструмент у побудові бізнес-логіки та реалізації критеріїв вибірки даних у багатошарових системах, сучасні умови розробки програмного забезпечення висувають до нього значно вищі вимоги, ніж ті, які враховувалися під час створення класичних реалізацій. Еволюція архітектур — від монолітів до мікросервісів, від локального виконання до контейнеризованих хмарних середовищ, а також зростання складності доменної логіки, — сформували новий контекст, у якому традиційні підходи виявляються недостатніми. Саме тому виникає об’єктивна необхідність поглибленого вдосконалення існуючих методів, що використовують патерн «Specification», з метою забезпечення їх відповідності сучасним викликам індустрії.

Першою і найбільш очевидною причиною є зростання очікувань щодо прозорості, контрольованості та спостережуваності програмних систем. У сучасних умовах компанії прагнуть будувати рішення, які дозволяють не лише формувати коректні запити до джерел даних, але й детально аналізувати вплив кожної бізнес-операції на загальну продуктивність застосунку. Якщо у минулі роки основною задачею було отримання правильної вибірки, то сьогодні критично важливо розуміти: наскільки ефективно працює та чи інша специфікація, які запити вона породжує в реальному середовищі, наскільки вони оптимальні, які ресурси споживають, чи не викликають вони деградації продуктивності. Відсутність подібних інструментів аналізу створює ситуацію, коли застосунки працюють як «чорні ящики», всередині яких важко зрозуміти, чому виникають ті чи інші проблеми з продуктивністю.

У реальних проєктах це призводить до накопичення технічної заборгованості, появи прихованих «вузьких місць», збільшення часу реакції системи та зростання вартості її підтримки. Існуючі реалізації патерну «Специфікація» не забезпечують можливостей автоматизованого збору метрик, глибокого трасування запитів, оцінювання складності логічних умов або моніторингу взаємодії специфікацій із базою даних. Як наслідок, розробники вимушені або аналізувати поведінку системи вручну, або інтуїтивно припускати, де саме може виникати проблема. Такий підхід є непродуктивним у довгостроковій перспективі та не відповідає сучасним вимогам до побудови складних систем.

Другою важливою причиною необхідності вдосконалення є відсутність ефективних механізмів адаптації та розширюваності. Більшість наявних бібліотек і фреймворків пропонують фіксований набір можливостей, який обмежує розробників у спробах інтегрувати додаткові функції, такі як: валідація специфікацій, аналіз складності, профілювання їх виконання, автоматична генерація рекомендацій щодо оптимізації. Але реальні проєкти часто потребують саме таких функцій. Наприклад, у доменах з великою кількістю бізнес-правил (e-commerce, фінансові системи, логістика) специфікації можуть ставати настільки складними, що без спеціальних інструментів важко підтримувати їх у належному стані. Вдосконалення підходів має забезпечити гнучкість, яка дозволить адаптувати специфікації до складних сценаріїв, без необхідності переписувати велику частину бізнес-логіки.

Третьою причиною є зростання потреби у масштабованості застосунків. Сучасні .NET-рішення часто працюють у хмарних середовищах, де система може бути розподілена між десятками або сотнями вузлів. У таких умовах різні специфікації можуть мати різний вплив на продуктивність залежно від того, на якому сервері вони виконуються, під яким навантаженням знаходяться джерела даних, у якому контексті виконався запит. Традиційні реалізації патерну «Specification» зовсім не враховують цих аспектів. Вони трактують застосунок як моноліт, у якому всі запити однакові за своїми умовами виконання. Водночас хмарні системи потребують можливостей глобального моніторингу та аналізу

поведінки специфікацій у різних середовищах, на різних інстансах, при різній кількості підключень. Без цього масштабування стає «всліпу» — розробники можуть збільшувати кількість реплік або покращувати конфігурацію серверів, але без розуміння того, яка саме частина логіки є джерелом навантаження.

Четвертою причиною є потреба у структурній якості та архітектурній чистоті. Велика кількість специфікацій у проєкті створює ризик дублювання логіки, порушення інкапсуляції, неправильного розділення відповідальностей між шарами системи. Без відповідних інструментів аналізу та валідації розробники нерідко створюють надмірно складні специфікації, що порушують принципи SOLID, особливо принцип єдиного обов'язку та принцип відкритості/закритості. У великих кодових базах, де можуть бути сотні специфікацій, майже неможливо вручну відстежити залежності та виявити порушення архітектурних правил. Саме тому вдосконалення наявних методів повинно включати інструменти, які забезпечуватимуть автоматичний аналіз структури специфікацій, виявлення дублювання та недоцільних залежностей, а також формування рекомендацій щодо рефакторингу.

П'ятою важливою причиною є відсутність методів оцінювання якості специфікацій. У більшості бібліотек поняття «якісної специфікації» не визначене. Немає зрозумілих метрик, немає інструментів для вимірювання складності або ефективності, немає можливості порівнювати різні реалізації або оцінювати їхній вплив на продуктивність. Це означає, що навіть якщо розробник напише специфікацію, яка функціонально виконує свої завдання, ніхто не гарантує, що вона робить це оптимально. Брак кількісних метрик призводить до того, що оцінювання якості специфікацій стає суб'єктивним і залежить від досвіду конкретного розробника. У великих командах, де працюють десятки інженерів, це створює ризик фрагментації підходів і зниження якості рішень.

Шостим чинником є необхідність покращеного інструментарію для автоматичного збору та агрегування метрик, які могли б стати базою для системи рекомендацій. Нині специфікації не мають вбудованих механізмів логування, немає інтелектуальних підказок, немає рекомендацій щодо оптимізації умов

вибірки. Така система могла б автоматично виявляти складні або надмірні умови, підказувати можливі оптимізації, вказувати на потенційні проблеми або попереджати про ризики, але в існуючих реалізаціях подібний функціонал відсутній. У результаті розробники працюють без підтримки інструментів, які могли б суттєво полегшити процес розробки та підвищити якість бізнес-логіки.

Ще одним суттєвим аргументом на користь удосконалення методів, що реалізують патерн «Specification», є сучасні тенденції до автоматизації процесів розробки та інтеграції систем у CI/CD конвеєри. Але чинні реалізації не надають жодних інструментів для стандартизованого тестування специфікацій, немає вбудованих засобів виявлення змін, які можуть порушити попередню поведінку, або оцінювання того, чи залишилася специфікація коректною після рефакторингу. Це створює значні ризики для команд, які активно вдосконалюють архітектуру, оскільки будь-які внутрішні зміни можуть непомітно вплинути на формування SQL-запитів, логіку фільтрації чи сортування.

У реальних проєктах зміни специфікацій часто відбуваються не в межах однієї людини, а в командному середовищі, де різні розробники можуть працювати над пов’язаними елементами бізнес-логіки. Відсутність механізмів перевірки сумісності та узгодженості специфікацій призводить до появи ситуацій, коли одні специфікації суперечать іншим, дублюють їх або навіть некоректно комбінуються між собою. Сучасні проєкти потребують засобів автоматичного аналізу та валідації таких взаємодій, щоб забезпечити передбачуваність поведінки застосунку незалежно від кількості розробників та швидкості змін.

Важливим аспектом є також зростання значущості доменного моделювання. У межах сучасних підходів до архітектури, таких як DDD, специфікації мають не лише служити інструментом для формування запитів, а й відображати реальні бізнес-правила предметної області. Це означає, що якість специфікацій безпосередньо впливає на якість архітектури та можливість масштабувати систему у майбутньому. Однак існуючі реалізації часто пропонують поверхневий підхід, який розглядає специфікацію виключно як засіб побудови запитів, без глибшого аналізу того, чи відображає вона справжню бізнес-логіку. Ускладнення предметних

областей та прагнення до точного відображення бізнес-правил роблять необхідним створення інструментів, які автоматизують перевірку семантики специфікацій, визначають правильність їх структури та відповідність бізнес-вимогам.

Крім того, сучасні методи побудови програмних систем орієнтуються на зниження навантаження на бази даних, і тут патерн «Specification» відіграє ключову роль, адже саме він визначає, як формуються запити. Проблема полягає в тому, що класичні реалізації не враховують особливостей масштабування баз даних, таких як розподілені індекси, шардинг, реплікація, асинхронна обробка запитів. Вони також не дають можливості адаптувати специфікації до різних типів джерел даних — SQL, NoSQL, кешів, шардованих колекцій чи потокових систем. Таким чином, відсутність універсальних і водночас розширюваних механізмів призводить до того, що розробники змушені створювати окремі реалізації для різних типів сховищ, що збільшує складність підтримки та супроводу системи.

Ще одним аргументом є потреба у візуалізації логіки специфікацій. На відміну від простих фільтрів, які легко читати і розуміти, складні специфікації можуть містити вкладені умови, комбіновані правила, декілька об’єднань та складні залежності. У великих проєктах без візуального представлення такої логіки стає вкрай складно аналізувати її, особливо коли йдеться про командну роботу або онбординг нових розробників. Відсутність таких інструментів у наявних реалізаціях робить процес розуміння специфікацій трудомістким та неефективним, що призводить до зростання ризиків помилок. Наявність інструментів для автоматичної побудови діаграм, блок-схем або графів, які відображають структуру та взаємозв’язки специфікацій, могла б суттєво покращити зрозумілість системи в цілому.

До того ж, сучасні системи повинні відповідати високим вимогам щодо надійності та стійкості до неочікуваних помилок. Однак чинні методи майже не пропонують механізмів для обробки помилок, пов’язаних із некоректними, надмірними або конфліктними специфікаціями. У результаті некоректно сформована специфікація може викликати вийнятки під час виконання запиту, повністю зупиняючи роботу частини системи або навіть призводячи до відмови

сервісу. Це є значною проблемою для мікросервісних та розподілених систем, де кожен компонент повинен мати високий рівень автономності і не створювати внутрішніх точок відмови. Удосконалення методів має включати механізми безпечного виконання, автоматичного виявлення потенційно небезпечних умов та захисних перевірок.

Окрему увагу варто приділити питанню оптимізації роботи розробника. У сучасних умовах розробники прагнуть працювати з інструментами, які допомагають їм не лише писати код, а й підтримувати його у високоякісному стані. Слабкість наявних реалізацій полягає в тому, що вони не містять рекомендацій щодо написання, структурування та вдосконалення специфікацій на основі зібраних метрик. Немає інструментів, які допомагають розробнику оцінити, чи не є специфікація надто складною, чи не дублює вона іншу логіку, чи відповідає вона принципам архітектурних практик. У світі, де важлива кожна хвилина продуктивності команди, інструменти, що підтримують розробника у вирішенні таких завдань, стають критично необхідними.

Сучасні вимоги до аналітики та моніторингу також передбачають інтеграцію з потужними системами спостереження, такими як Elastic Stack, Prometheus, OpenTelemetry, Application Insights тощо. Але існуючі бібліотеки, що реалізують патерн «Specification», не мають готової інтеграції з системами збору метрик. Вони не дозволяють автоматично знімати інформацію про тривалість виконання запитів, кількість викликів, частоту використання певних специфікацій, співвідношення успішних та неуспішних операцій. Відсутність такої інтеграції суттєво знижує можливості DevOps- та SRE-команд у підтримці системи на належному рівні продуктивності та надійності.

Також помітною проблемою є відсутність можливості аналізувати еволюцію специфікацій у часі. У великих застосунках система специфікацій розвивається разом із бізнес-вимогами. Без відповідних інструментів важко відстежити, як зміни вплинули на продуктивність або на архітектурну чистоту. Наявність механізмів історичного аналізу могла б допомагати у виявленні регресій, відстеженні проблем після рефакторингу, а також у прогнозуванні впливу майбутніх змін.

Не менш значущою є проблема різнорідності джерел даних, що використовуються у сучасних застосунках. Якщо раніше платформенні рішення здебільшого передбачали роботу з однією ORM або однією базою даних, то нині поширеною є практика використання декількох СУБД одночасно — реляційних, документних, графових або навіть time-series сховищ. Наявні методи реалізації патерну «Специфікація» часто прив’язані до певного типу джерела даних або певної ORM, наприклад Entity Framework. Така жорстка залежність унеможливлює легко переносити специфікації між компонентами системи або застосовувати їх у контекстах, де використовується інша технологія доступу до даних. Саме тому актуальним стає удосконалення підходів, що дозволять реалізовувати «Специфікацію» у більш універсальний спосіб — незалежно від конкретного механізму зберігання даних чи конкретного інструменту їх вибірки.

Крім того, сучасні системи часто мають складну доменну модель, що містить велику кількість сутностей і зв’язків між ними. Зі зростанням складності предметної області також збільшується кількість умов, обмежень, фільтрів та бізнес-правил, які повинні бути виражені у специфікаціях. Стандартні підходи не завжди передбачають зручні механізми для композиції, комбінування та повторного використання великих специфікацій без дублювання коду. Це породжує плутанину у розробці: важко визначити, де саме знаходиться логіка, які умови застосовуються до певної операції, які модифікації необхідно внести у разі зміни бізнес-правил.

У результаті специфікації, замість того щоб слугувати інструментом впорядкування і концентрації логіки в одному місці, інколи стають джерелом надлишкової складності та фрагментації коду. Звідси виникає очевидна потреба у механізмах, які б автоматично контролювали структуру специфікацій, надавали рекомендації щодо їх оптимізації, перевіряли надмірність чи дублювання умов. Такі функції не лише зробили б патерн більш зручним, але й значно підвищили б його практичну користь для масштабних проєктів.

Проблемою, яка також формує необхідність удосконалення існуючих методів, є відсутність засобів інтеграції специфікацій у загальну систему контролю

якості програмного забезпечення. Незважаючи на те, що патерн активно використовується у реальних проєктах, він майже не має супровідних інструментів для статичного аналізу, автоматичного тестування, виявлення антипатернів у реалізації. Відповідно, якість реалізації специфікацій повністю залежить від компетентності розробника, а отже — суб'єктивна та важко контрольована. У великих командах це призводить до того, що різні розробники формують специфікації по-своєму, з різною структурою, різними підходами та різними практиками комбінування умов. Поступово це перетворюється на технічний борг, який складно моніторити, оцінювати або виправляти. Тому інтеграція механізмів автоматичного аналізу специфікацій — це логічний і необхідний крок для приведення патерну у відповідність із вимогами сучасної інженерії програмного забезпечення.

Отже, аналіз сучасних тенденцій розвитку програмних систем та особливостей використання патерну «Specification» переконливо демонструє, що існуючі методи його реалізації вже не відповідають актуальним потребам індустрії. Збільшення складності предметних областей, перехід до мікросервісних і модульних архітектур, а також високі вимоги до масштабованості, прозорості й спостережуваності роботи застосунків створюють новий рівень викликів, з якими класичні підходи не справляються.

Виявлені проблеми — відсутність засобів трасування, моніторингу, автоматичного аналізу специфікацій, інструментів оптимізації, рекомендацій щодо покращення, а також обмежена підтримка композиційності та переносимості — свідчать про суттєвий розрив між теоретичною моделлю патерну та реальними умовами його використання в сучасних масштабованих системах. Через це розробникам доводиться створювати власні рішення, що збільшує технічний борг, ускладнює супровід і знижує рівень стандартизації всередині команди.

Тому вдосконалення існуючих методів реалізації патерну «Specification» є не просто бажаним, а об'єктивно необхідним кроком для забезпечення ефективного, гнучкого та передбачуваного функціонування багатошарових і високонавантажених .NET-застосунків. Подальший розвиток підходів має бути

спрямований на підвищення прозорості, розширення інструментів аналізу, інтеграцію з моніторинговими системами, а також на створення механізмів рекомендацій і контролю якості, що дозволить перетворити специфікації на повноцінний потужний інструмент не лише моделювання бізнес-логіки, а й забезпечення її стабільності, ефективності та масштабованості.

1.3 Огляд патерну «Specification» та його роль у побудові багатошарової архітектури

Патерн «Specification» є одним із ключових концептуальних механізмів, який широко використовується у сучасній розробці програмного забезпечення для формалізації, структурування та централізації бізнес-логіки. Він дозволяє відокремити процес визначення умов, критеріїв або правил від решти компонентів системи, що відповідає принципам чистої архітектури та сприяє створенню модульних і легко розширюваних застосунків. Використання цього патерну набуває особливої актуальності в умовах побудови багатошарових архітектур, де чітке розмежування відповідальностей відіграє фундаментальну роль у забезпеченні масштабованості, тестованості та підтримуваності системи.

Історично патерн «Specification» виник у межах об’єктно-орієнтованого підходу до аналізу й проєктування програмних систем. Його основна ідея полягає у тому, щоб виразити бізнес-логіку у вигляді окремих об’єктів, які визначають певну умову і можуть бути легко перевірені або комбіновані з іншими умовами. Завдяки цьому розробники можуть формувати складні правила поведінки, не порушуючи принципів інкапсуляції та не змішуючи логіку з інфраструктурними або прикладними компонентами.

Сутність патерну полягає у створенні абстракції, яка описує певну вимогу: наприклад, «користувач повинен мати активний статус», «замовлення повинно бути оплачено», «продукти повинні відповідати умовам фільтрації». Ці вимоги можуть використовуватися як окремо, так і в комбінації одна з одною за допомогою логічних операцій «і», «або», «не». Такий підхід забезпечує високий рівень

гнучкості, адже бізнес-логіка перестає бути жорстко прив’язаною до конкретних алгоритмів або запитів до бази даних.

Важливою перевагою цього патерну є можливість відокремлення логіки від деталей реалізації доступу до даних. У традиційних системах правила вибірки часто «розмиваються» у вигляді фрагментів SQL-запитів або LINQ-виразів, які розкидані по всьому застосунку. У результаті бізнес-логіка стає важкою для аналізу, тестування та модифікації. Патерн «Specification» дозволяє уникнути цього, зосереджуючи правила в окремих структурованих об’єктах, що робить код більш прозорим і передбачуваним.

У контексті багатошарової архітектури цей патерн відіграє особливо важливу роль. Багатошарова архітектура передбачає чіткий поділ застосунку на рівні, такі як презентаційний, прикладний, доменний та інфраструктурний. Кожен із цих шарів виконує свою задач. Наприклад, доменний шар містить моделі та бізнес-правила, прикладний — сценарії використання, інфраструктурний — реалізації доступу до даних або зовнішніх сервісів. Однак однією з найбільших проблем багатошарових систем є надмірне дублювання логіки та порушення принципу єдиного джерела істини. Саме патерн «Specification» дозволяє уникнути цієї проблеми: усі правила, що визначають поведінку сутностей, зберігаються в одному місці — у доменному шарі.

Коли специфікація реалізована правильно, вона виконує роль універсального контракту між різними частинами системи. Наприклад, прикладний шар може просто передати специфікацію до репозиторію, не турбуючись про те, як саме будуть інтерпретовані умови фільтрації, сортування або включення пов’язаних даних. Інфраструктурний шар, у свою чергу, будує конкретний запит на основі цієї специфікації. Таким чином забезпечується слабке зв’язування, що є необхідною вимогою для побудови масштабованих та автономних компонентів.

Завдяки своїй універсальності патерн «Specification» інтегрується не лише з доменною моделлю, але й із іншими архітектурними концепціями. Наприклад, у контексті DDD специфікації використовуються як природний спосіб опису інваріантів, доменних політик або правил вибірки агрегатів. Вони забезпечують

чітку межу між доменною логікою та інфраструктурою, що повністю відповідає принципам стратегічного та тактичного моделювання в DDD.

У сучасних умовах патерн «Specification» еволюціонував від простого механізму логічних перевірок до комплексного інструменту, який підтримує гнучкі механізми фільтрації, проєкцій, сортувань, включення навігаційних властивостей, групувань та інших операцій. У деяких реалізаціях патерн також підтримує побудову складних запитів із умовами, які можуть формуватися динамічно залежно від сценаріїв використання або зовнішніх параметрів. Це значно збільшує його практичну цінність у реальних проєктах.

Патерн «Specification» має чітко визначену структуру, яка дозволяє забезпечити гнучкість, композиційність та повторне використання логіки.

Таблиця 1.2

Структура патерну «Specification»

Частина патерну	**Опис**	**Приклад у .NET**
Інтерфейс або абстрактний базовий клас	Головна точка входу для всіх реалізацій специфікацій. Визначає метод(и), які перевіряють відповідність об’єкта умовам специфікації.	Метод IsSatisfiedBy(T entity) повертає true або false. Забезпечує узгоджений контракт для всіх специфікацій.
Конкретні специфікації	Реалізації базового інтерфейсу або абстрактного класу з конкретними умовами для об’єктів.	«Активний користувач» перевіряє IsActive; «Преміум-користувач» оцінює рівень підписки. Можуть комбінуватися для складних правил.
Композиційні специфікації	Дозволяють об’єднувати базові специфікації за допомогою логічних операторів AND, OR, NOT.	Створюють складні умови без зміни базових специфікацій. Наприклад, «Активний та преміум-користувач».
Абстракції для інтеграції з джерелами даних	Механізми трансляції логіки специфікацій у запити до бази даних або інших джерел.	LINQ-вирази, Expression Trees, об’єкти для SQL-запитів. Забезпечують декларативність та ефективність виконання.

Щодо видів патерну «Specification», вони поділяються за рівнем абстракції та способом використання. Кожен вид визначає, як і в яких умовах можуть застосовуватися специфікації, а також який рівень гнучкості та повторного використання вони забезпечують.

Таблиця 1.3

Види патерну «Specification»

Вид	**Характеристика**	**Особливості використання**
Прості (базові) специфікації	Містять одну умову або невеликий набір взаємопов'язаних умов	Використовуються як будівельні блоки для складніших правил
Композиційні специфікації	Формують складні правила на основі простих	Наприклад, «Користувач активний і має преміум-підписку»
Динамічні специфікації	Формуються під час виконання програми залежно від зовнішніх параметрів або контексту	Адаптація бізнес-логіки під роль користувача, стан системи або налаштування середовища
Універсальні специфікації	Можна застосовувати до різних типів сутностей завдяки узагальненням або шаблонам	Створюють повторно використовувані компоненти для перевірки умов у різних частинах системи

Використання патерну також значно полегшує тестування. Оскільки специфікації реалізуються як окремі об'єкти, їх можна тестувати автономно, створюючи юніт-тести без залучення бази даних або інфраструктури. Для складних композицій можна перевіряти коректність логічних комбінацій, гарантуючи, що зміни в одній базовій специфікації не порушують роботу інших частин системи. Такий підхід зменшує ймовірність помилок та підвищує стабільність системи.

Ще однією перевагою є масштабованість. Патерн дозволяє додавати нові специфікації або змінювати існуючі без модифікації решти системи. Це особливо важливо для великих багатошарових застосунків, де бізнес-логіка швидко змінюється і потребує постійного оновлення. Завдяки централізованому підходу до правил і можливості композиції, система легко адаптується до нових вимог, що суттєво підвищує її довговічність і ефективність підтримки.

Патерн «Specification» також забезпечує узгодженість і повторюваність бізнес-правил. У багатошаровій архітектурі різні частини системи можуть працювати з одними й тими ж даними або процесами. Завдяки специфікаціям всі шари отримують доступ до однієї централізованої логіки, що виключає ризик дублювання або суперечливих реалізацій правил у різних компонентах. Це підвищує надійність і передбачуваність роботи системи.

Варто відзначити, що патерн «Specification» може інтегруватися з сучасними практиками розробки, такими як CQRS та Event Sourcing. У CQRS специфікації зазвичай використовуються для формування умов read-моделей, що дозволяє ізолювати запити на читання від бізнес-логіки, при цьому зберігаючи централізовану систему правил. У Event Sourcing специфікації можуть визначати правила фільтрації подій або вибірки агрегатів, забезпечуючи контроль над станами системи і мінімізуючи ризик помилок у обробці історії подій.

Крім того, використання патерну дозволяє забезпечити адаптивність системи. Динамічні специфікації дозволяють змінювати умови вибірки або фільтрації залежно від контексту: ролі користувача, стану системи, зовнішніх параметрів або політик безпеки. Це забезпечує більш гнучку взаємодію між шарами, полегшує масштабування та підвищує рівень автоматизації бізнес-процесів.

Патерн «Specification» є потужним інструментом для формалізації бізнес-логіки, централізації правил і забезпечення їх повторного використання у багатошарових системах. Його структура, яка включає інтерфейси або абстрактні базові класи, конкретні та композиційні специфікації, а також механізми інтеграції з джерелами даних, дозволяє будувати прозору, модульну та гнучку логіку.

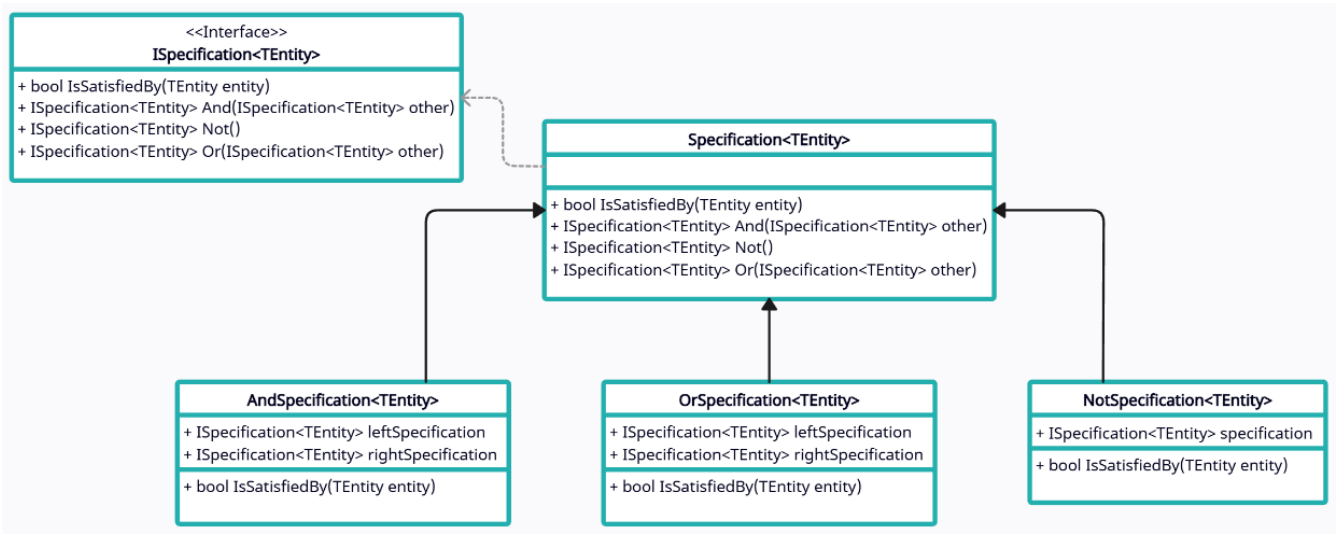


Рис.1.2. Діаграма класів шаблону «Specification»

Розглянуті види патерну — прості, композиційні, динамічні та універсальні специфікації — забезпечують різний рівень абстракції та можливість адаптації під конкретні потреби проєкту. Комбінація цих підходів дозволяє створювати складні правила без дублювання коду, підтримувати динамічні умови та централізовано контролювати бізнес-логіку.

У багатошаровій архітектурі патерн «Specification» виконує ключові ролі:

- відокремлює бізнес-правила від інфраструктури та прикладного коду;
- забезпечує слабке зв’язування між компонентами;
- сприяє тестованості і підвищує прозорість логіки;
- підтримує масштабованість та повторне використання специфікацій;
- дозволяє інтегрувати правила у процес формування запитів до бази даних та взаємодії з іншими шарами системи.

Таким чином, застосування патерну «Specification» є не лише ефективним засобом організації логіки, але й важливим механізмом для підтримки принципів багатошарової архітектури, забезпечення стабільності, передбачуваності та гнучкості сучасних .NET-застосунків. Його впровадження дозволяє знизити технічний борг, підвищити продуктивність розробки і сприяє створенню систем, які легко адаптуються до змін бізнес-вимог і масштабуються без значних модифікацій коду.

2 МОДЕЛЮВАННЯ ТА АНАЛІЗ МЕТОДІВ ПІДВИЩЕННЯ МАСШТАБОВАНОСТІ БАГАТОШАРОВИХ .NET-СИСТЕМ

2.1 Аналіз сучасних бібліотек, у яких реалізовано патерн «Specification»

Сучасна екосистема .NET пропонує різноманітні бібліотеки, що реалізують або інтерпретують концепцію специфікацій. Найбільш відомі рішення з’явилися в контексті підходів DDD, де бізнес-логіка потребує чистих та відокремлених способів опису правил. Однак навіть за наявності великої кількості інструментів, їх функціональність, якість реалізації та ступінь інтеграції з ORM та архітектурними підходами суттєво відрізняються.

Ardalis.Specification — це найбільш популярна й широко використовувана у .NET бібліотека для реалізації патерну «Specification». Вона стала де-факто стандартом у середовищі розробників, особливо серед прихильників Clean Architecture, DDD та модульних корпоративних застосунків. Стів Сміт розробив цю бібліотеку з акцентом на високу читабельність, передбачуваність і зручність створення складних запитів до даних.

Ardalis.Specification — це надбудова над LINQ та Entity Framework Core, яка дозволяє оголошувати запити у вигляді окремих об’єктів. Кожна специфікація містить у собі всі аспекти запиту:

- умови Where;
- сортування;
- Include для навігаційних властивостей;
- проекції;
- обмеження кількості результатів;
- групування тощо.

Іншими словами, специфікація є декларативним описом запиту, який розробник може використовувати повторно в різних сценаріях.

Бібліотека працює через два ключові елементи:

1. Інтерфейс ISpecification<T>. Описує контракт специфікації: набір умов та параметрів, що визначають, як саме слід отримати дані.
2. SpecificationEvaluator. Цей компонент приймає IQueryable<T> та специфікацію, автоматично накладаючи всі правила:
   1. комбінує предикати;
   2. додає Include();
   3. застосовує OrderBy();
   4. формує проекції;
   5. додає AsNoTracking() при потребі.

У результаті розробнику не потрібно вручну писати складні LINQ-запити. Усе зібрано у єдиний об’єкт, який легко тестувати, розширювати та комбінувати.

Особливістю бібліотеки є повна інтеграція з патерном «Repository», де методи репозиторію очікують специфікацію як аргумент.

Переваги Ardalis.Specification:

1. Стандартизація підходу до запитів. Усі запити формуються структуровано та уніфіковано, що спрощує великий командний розвиток.
2. Повна інтеграція з EF Core. Автоматична трансляція в оптимізований SQL, підтримка Include, ThenInclude, проекцій, асинхронності.
3. Висока читабельність коду. Кожна специфікація — окремий клас або об’єкт, що чітко описує єдину операцію.
4. Підтримка складних композицій. Можна комбінувати специфікації, наслідувати одну від одної, розбивати їх на дрібніші частини.
5. Розширюваність. Підтримує кастомні операції, такі як Search, Pagination, QueryStringParameters.
6. Популярність і спільнота. Велика кількість матеріалів, гайдів, прикладів у розробників Clean Architecture.

Недоліки Ardalis.Specification:

- залежність від EF Core для повноцінної роботи;

- не забезпечує аналізу продуктивності специфікацій;
- при неправильному використанні можливе розростання кількості класів;
- не контролює оптимальність Include та проекцій.

NHibernate Specification — це одна з найстаріших і найбільш класичних реалізацій патерну «Specification» в екосистемі .NET. Вона виникла як частина інструментарію ORM-фреймворку NHibernate, який довгий час був основним конкурентом Entity Framework у корпоративних системах. Саме NHibernate був одним з перших, хто ввів у практику ідею винесення логіки фільтрації в окремі об’єкти — специфікації — з чітко визначеним способом їх комбінування.

NHibernate Specification — це набір абстракцій, який дозволяє оголошувати логіку відбору сутностей як окремі, незалежні об’єкти, що можуть бути застосовані до будь-якого IQueryable або Criteria-запиту NHibernate.

Специфікації оформляються як класи, що реалізують певні контракти і містять:

- критерії пошуку;
- правила комбінування (AND/OR/NOT);
- можливі обмеження на вибірки;
- логіку сортування;
- перехідні умови між різними специфікаціями.

На відміну від Ardalis.Specification, тут наголос робиться не на включеннях і проекціях, а на критеріальному підході, характерному для NHibernate.

Таблиця 2.1

Як працює NHibernate Specification

Аспект	**Опис**
Ключова ідея — Criteria API	NHibernate використовує власний механізм формування запитів — Criteria API. Специфікація виконує роль обгортки, перетворюючи предикати на критерії NHibernate. Дозволяє накладати фільтри, додавати обмеження по полях, формувати складні запити.
Комбінування специфікацій	Підтримується класична трійка: - AndSpecification, - OrSpecification, - NotSpecification. Це забезпечує можливість створення складних бізнес-правил без дублювання коду та дозволяє гнучко комбінувати логіку.
Використання у патерні Repository	Специфікація передається в репозиторій, де автоматично транслюється у NHibernate QueryOver або Criteria-запит. Результат запиту повертається у вигляді доменних сутностей.

Переваги NHibernate Specification:

1. Історична значущість. NHibernate був першим ORM у .NET, де специфікації стали стандартною практикою — звідси багато сучасних підходів ідейно походять саме від нього.
2. Сильна підтримка складних умов. Criteria API дозволяло будувати складні бізнес-логічні вирази, включаючи підзапити, групування, SQL-фрагменти.
3. Хороша інтеграція з DDD. NHibernate традиційно використовувався у проєктах, які орієнтуються на Domain-Driven Design.
4. Гнучка система композиції. Легко комбінувати і перевикористовувати специфікації через логічні оператори.

5. Відсутність прив’язки до EF Core. Підходить для систем, де архітектура не залежить від Microsoft-екосистеми.

```
var validator = ValidationCatalog<User>
.For(u => u.IsActive)
.And(u => u.Email != null && u.Email.Contains("@"));
```

Рис.2.1 Приклад специфікації бібліотеки NHibernate

Недоліки NHibernate Specification:

- сильна прив’язка до NHibernate, що робить бібліотеку малопридатною у сучасних EF Core-проектах;
- обмежена спільнота у порівнянні з EF Core екосистемою;
- відсутність сучасних можливостей, таких як проекції, оптимізація Include або розширена робота зі складними LINQ-запитами;
- не масштабується під нові стандарти .NET (ORM майже не розвивається);
- підтримка специфікацій менш структурована, ніж у Ardalis.Specification.

LINQKit — це утилітарна бібліотека для .NET, яка дозволяє працювати зі складними виразами LINQ динамічно, створювати композиції предикатів та комбінувати умови у гнучкий спосіб. Основний компонент бібліотеки — PredicateBuilder, який часто використовується для реалізації концепції патерну «Specification» без створення окремих класів специфікацій. LINQKit є класичною реалізацією специфікації і дозволяє будувати динамічні та складні умови фільтрації, які легко комбінувати.

Це дає змогу розробнику:

- будувати складні пошукові умови;
- комбінувати декілька правил у єдиний запит;
- застосовувати логіку на різних рівнях (репозиторій, сервіси, API).

```
var validator = ValidationCatalog<User>
.For(u => u.IsActive)
.And(u => u.Email != null && u.Email.Contains("@"));
```

Рис.2.2 Приклад специфікації бібліотеки LINQKit

Переваги LINQKit:

1. Гнучкість. Можливість комбінувати будь-яку кількість умов без створення окремих класів специфікацій.
2. Універсальність. Працює з будь-яким LINQ-провайдером: EF Core, NHibernate, in-memory колекціями.
3. Простота використання. Не потрібна складна структура або ієрархія специфікацій — досить Expression і методів комбінування.
4. Динамічні запити. Підійде для фільтрів у веб-додатках, де умови формуються динамічно на основі введених користувачем параметрів.

Недоліки LINQKit:

- не забезпечує структуровану модель специфікації — усі predicat-и лишаються Expression;
- відсутня підтримка Include / ThenInclude та проекцій, характерних для Ardalis.Specification;
- менш наочний та організований для великих командних проєктів;
- не надає контролю за оптимальністю запитів у базі даних — розробник повністю відповідає за складність і ефективність Expression.

Marten — це сучасний ORM для PostgreSQL, який реалізує патерн Document Database для .NET. Крім звичайних CRUD-операцій, Marten підтримує LINQ-запити і дозволяє створювати власні специфікації через Expression. У контексті патерну «Specification» Marten надає зручний спосіб визначати критерії пошуку для документів і комбінувати їх у складні правила.

Marten дозволяє працювати з документами як з першокласними об’єктами .NET, підтримуючи повну інтеграцію з PostgreSQL JSON/JSONB. Для реалізації специфікацій можна:

- комбінувати специфікації через логічні оператори AND/OR;
- застосовувати сортування, пагінацію, проекції на рівні Expression;
- виконувати запити як синхронно, так і асинхронно.

Важливим аспектом є ORM-агностичний підхід: специфікації можуть працювати з будь-якими типами запитів і зручно інтегруються з LINQ.

Переваги Marten Query Specifications:

1. Підтримка документно-орієнтованих даних. Можна створювати специфікації для JSONB-документів, що недоступно для класичних ORM.
2. Гнучкість. Predicate-и легко комбінуються, можна будувати динамічні запити на льоту.
3. Асинхронна підтримка. Повна підтримка асинхронних запитів до бази, що важливо для масштабованих систем.
4. Інтеграція з LINQ. Всі специфікації працюють з IQueryable і повністю сумісні з LINQ.

Недоліки Marten Query Specifications:

- не надає повної інкапсуляції патерну «Repository + Specification» як у Ardalis.Specification;
- обмежена підтримка складних Include / ThenInclude, характерних для реляційних ORM;
- відсутні інструменти для аналізу продуктивності специфікацій у реальному часі;
- менш популярна в порівнянні з EF Core, що зменшує кількість прикладів.

Аналіз сучасних бібліотек, що реалізують патерн «Specification», показав, що існує кілька різних підходів до організації логіки відбору та фільтрації даних у .NET-проєктах. Найбільш популярними є Ardalis.Specification, NHibernate

Specification, LINQKit / PredicateBuilder та Marten Query Specifications, кожна з яких має свої особливості, переваги та обмеження.

Ardalis.Specification забезпечує структуровану інтеграцію з EF Core, підтримку Include та проекцій, а також можливість композиції специфікацій у межах Repository.

NHibernate Specification традиційно орієнтована на Criteria API та складні умови фільтрації, активно використовується у DDD-проєктах на основі NHibernate.

LINQKit пропонує гнучке комбінування Expression, дозволяючи створювати динамічні умови без складної ієрархії класів.

Marten Query Specifications оптимізована під документно-орієнтовані дані PostgreSQL і дозволяє працювати з JSONB-документами через LINQ.

Попри різноманітність рішень, усі бібліотеки мають спільні обмеження: відсутність інструментів для аналізу продуктивності, контролю над складністю запитів та автоматизованих рекомендацій щодо оптимізації специфікацій. Також більшість рішень орієнтовані на конкретні ORM або тип даних, що обмежує універсальність підходу.

Таким чином, сучасні бібліотеки забезпечують базові можливості для реалізації патерну «Specification», але не вирішують питання оптимізації, аналізу та стандартизації методів побудови специфікацій на високому рівні. Це створює необхідність у більш детальному порівняльному аналізі методів, які реалізують патерн «Specification», із врахуванням продуктивності, гнучкості та масштабованості.

2.2 Порівняльний аналіз методів, що реалізують патерн «Specification»

Патерн «Specification» застосовується для формалізації логіки вибору об'єктів у програмних системах та забезпечує можливість її повторного використання та комбінування. У попередньому підрозділі розглянуто чотири основні підходи до реалізації цього патерну: Ardalis.Specification, NHibernate

Specification, LINQKit та Marten. Кожен із них має свої особливості щодо синтаксису, інтеграції з ORM, продуктивності та масштабованості.

У цьому підрозділі представлено порівняльний аналіз цих методів у вигляді таблиці, що дозволяє наочно побачити відмінності між ними за ключовими характеристиками. Додатково наведено детальний опис результатів порівняння, де розглядаються переваги та обмеження кожного підходу.

Таблиця 2.2

Порівняння методів реалізації патерну «Specification»

Характеристика	**Ardalis.Specification**	**NHibernate Specification**	**LINQKit**	**Marten**
Збір метрик виконання специфікацій	Немає	Частково	Обмежено	Так
Підказки щодо покращення роботи	Немає	Немає	Обмежено	Так
Зручність логування	Середня	Складна	Середня	Висока
Підтримка складних умов / фільтрів	Висока	Середня	Висока	Висока
Масштабованість / продуктивність	Середня	Середня	Висока	Висока
Складність навчання / впровадження	Низька	Середня	Середня	Середня

Ardalis.Specification пропонує інтуїтивно зрозумілий механізм створення специфікацій. Основна концепція полягає у спадкуванні від базового класу Specification<T> та реалізації методу ToExpression(), у якому задаються умови відбору даних. Такий підхід дозволяє розробникам формувати логіку вибору об’єктів без необхідності писати вручну складні LINQ-запити або HQL-критерії. Цей підхід знижує поріг входу для новачків і дозволяє одразу інтегрувати специфікацію з DbContext через репозиторій. Крім того, Ardalis.Specification дозволяє включати пов’язані сутності через AddInclude, що зменшує потребу у ручному написанні JOIN-запитів. Проте існують обмеження: при побудові

складних підзапитів або роботі з кількома контекстами одночасно необхідно розробляти власні розширення або додаткові методи. Для великих систем із численними включеннями, підзапитами та динамічними умовами може знадобитися написання кастомного коду, що трохи знижує загальну простоту використання.

NHibernate Specification використовує Criteria API та HQL. Цей підхід дає значну гнучкість, але потребує детального розуміння внутрішньої роботи NHibernate, особливостей FetchMode, кешування та правильного комбінування Criterion і Restrictions. Розробнику доводиться управляти кожним Criterion вручну, що підвищує ризик помилок і ускладнює підтримку коду. Крім того, для складних комбінацій критеріїв часто потрібні додаткові допоміжні класи, що збільшує обсяг коду і знижує простоту використання для нових розробників.

LINQKit забезпечує роботу з Expression<Func<T,bool>>, що дозволяє динамічно комбінувати вирази та передавати їх у IQueryable. Це дає гнучкість у створенні складних умов, але потребує уважного використання Invoke і Compose. Помилки у комбінуванні виразів можуть призвести до дублювання запитів або неправильного виконання, особливо при великій кількості умов. Тому розробнику необхідно добре розуміти механізм Expression Tree, інакше простота використання знижується.

Marten застосовує специфічний API для роботи з PostgreSQL та документною моделлю. Синтаксис досить прямолінійний, однак для розробників, які звикли до реляційних баз даних, перехід на документну модель може бути складним.

Гнучкість і композиційність є ключовими характеристиками будь-якого підходу до реалізації патерну «Specification». Вона визначає, наскільки легко можна комбінувати окремі специфікації між собою, додавати нові умови без порушення існуючої логіки і масштабувати систему при зростанні складності бізнес-правил.

LINQKit виділяється серед розглянутих методів високою гнучкістю завдяки підтримці динамічного комбінування Expression. Розробник може створювати окремі модульні специфікації для різних умов і комбінувати їх за допомогою

логічних операторів. Це дозволяє уникати дублювання коду та спрощує повторне використання логіки у різних частинах системи. Однак така гнучкість вимагає уважного контролю за правильністю комбінування виразів. Неправильне використання може призвести до дублювання запитів, підвищеного навантаження на базу даних або некоректних результатів при виконанні складних умов.

Ardalis.Specification також підтримує композиційність, але її межі визначені архітектурою EF Core. Механізми AddInclude та AddCriteria дозволяють додавати додаткові умови і включення пов’язаних сутностей без зміни базового запиту. Це зручно для середніх проектів, де запити мають помірну складність, і дозволяє централізовано управляти логікою відбору даних. Проте при складних Join, підзапитах або потребі динамічно комбінувати багато специфікацій одночасно EF Core може генерувати надмірні SQL-запити, що ускладнює оптимізацію та контроль продуктивності.

NHibernate Specification забезпечує потужні можливості комбінування Criteria і HQL, що дозволяє створювати дуже складні умови відбору. Проте ця гнучкість часто досягається ціною високої трудомісткості: розробник повинен вручну контролювати правильне поєднання Criterion, оптимально вибирати FetchMode та враховувати кешування NHibernate. У великих системах з численними комбінованими специфікаціями це може значно ускладнити підтримку коду, підвищити ймовірність помилок і потребувати додаткових оптимізацій.

Marten демонструє високу гнучкість для роботи з документною моделлю. Комбінування специфікацій відбувається у межах LINQ-подібного API і дозволяє ефективно будувати складні умови без дублювання логіки. Особливості роботи з вкладеними документами, колекціями та умовною фільтрацією накладають певні вимоги до структури індексів та організації даних, що вимагає додаткової уваги при проектуванні системи. Неврахування цих моментів може призвести до зниження продуктивності або некоректного виконання специфікацій на великих масивах даних.

Продуктивність специфікацій визначає ефективність виконання запитів, витрати ресурсів сервера та час відгуку системи при роботі з великими обсягами

даних. Вона тісно пов’язана з механізмом генерації SQL або документних запитів, оптимізацією під конкретну ORM або базу даних, а також із структурою бізнес-логіки.

Marten демонструє високий рівень продуктивності завдяки виконанню запитів на стороні бази даних і оптимізації для документної моделі PostgreSQL. Основними перевагами є зниження навантаження на пам’ять, оскільки запити не потребують завантаження всіх документів у пам’ять для обробки. Особлива увага приділяється індексації та оптимізації запитів по вкладених документах і колекціях, що забезпечує ефективну роботу навіть при великих обсягах даних. Проте неправильна організація індексів або неврахування структури документів може негативно впливати на швидкість виконання специфікацій, особливо при складних умовних фільтрах.

Ardalis.Specification забезпечує стабільну продуктивність для простих і середніх запитів, оскільки інтегрується з EF Core, який оптимізує виконання стандартних LINQ-запитів. Продуктивність знижується у разі складних Include або комбінованих умов, коли EF Core генерує додаткові JOIN-запити. Це може призводити до збільшення часу виконання та підвищеного навантаження на базу даних. В таких випадках розробник повинен контролювати оптимізацію запитів і за необхідності впроваджувати власні механізми кешування або проєкції даних.

LINQKit дозволяє ефективно комбінувати Expression на льоту, що створює переваги для динамічних запитів. Продуктивність залежать від складності комбінованих виразів і їхньої взаємодії з ORM. Надмірне або некоректне використання Invoke і Compose може спричиняти дублювання запитів або неефективну генерацію SQL, що негативно впливає на час виконання. Важливо також враховувати взаємодію LINQKit з кешуванням ORM та оптимізацію обробки колекцій даних для збереження стабільної продуктивності.

NHibernate Specification надає широку функціональність, але продуктивність може бути нижчою у великих системах із складними Criteria та FetchMode. Часто NHibernate генерує додаткові SQL-запити або робить надмірне завантаження об’єктів через FetchMode, що призводить до високого навантаження на сервер і

збільшення часу відгуку. Для забезпечення ефективності необхідне правильне налаштування кешування, оптимізація Criteria та уважний контроль роботи з асоційованими сутностями.

Загалом, продуктивність реалізації специфікацій залежить не лише від вибору бібліотеки, але й від правильності побудови логіки, оптимізації під конкретну ORM чи документну модель та контролю за виконанням складних умов. Marten забезпечує максимальну продуктивність для документної моделі, Ardalis.Specification — стабільну для EF Core, LINQKit дозволяє динамічно оптимізувати комбіновані запити, а NHibernate Specification потребує додаткового контролю та налаштувань для великих обсягів даних.

Можливість збору метрик є важливою для підтримки масштабованості та ефективності програмних систем. Без відстеження продуктивності специфікацій складно оцінити, які запити потребують оптимізації, які вирази призводять до зайвих звернень до бази даних і де спостерігаються «вузькі місця».

У Ardalis.Specification повна відсутність збору метрик обмежує аналітику на рівні специфікацій. Розробник може використовувати зовнішні інструменти для профілювання запитів, але це не інтегровано в саму бібліотеку і не дозволяє відстежувати продуктивність окремих критеріїв або комбінацій специфікацій.

NHibernate Specification частково забезпечує збір метрик за рахунок внутрішнього логування SQL та кешу, а також можливості підключення сторонніх інструментів моніторингу. Проте отримана інформація часто є фрагментарною: вона не дає прямого аналізу виконання конкретної специфікації, а лише загальні дані по запитах, що виконуються через Criteria або HQL.

LINQKit обмежено підтримує збір метрик, оскільки основний фокус бібліотеки — комбінування Expression. Відстеження продуктивності вимагає ручного додавання логування виконання виразів та використання сторонніх інструментів для профілювання IQueryable. Це робить процес збору даних непростим і не завжди зручним для аналізу на рівні окремих специфікацій.

Marten надає повний контроль за збором метрик виконання специфікацій. Завдяки інтеграції з PostgreSQL і внутрішньому моніторингу запитів, розробник

може отримати детальну статистику по часу виконання, обсягу оброблених документів та ефективності використання індексів. Це дозволяє приймати обґрунтовані рішення щодо оптимізації специфікацій, впроваджувати зміни, що підвищують продуктивність, та підтримувати високу масштабованість системи навіть при обробці великих обсягів даних.

Можливість отримувати підказки щодо оптимізації специфікацій є важливим інструментом для підвищення продуктивності та підтримки чистоти коду. Такі підказки дозволяють розробнику визначати, які умови або комбінації специфікацій можна переписати більш ефективно, уникати зайвих включень або повторного виконання одних і тих самих запитів.

У Ardalis.Specification відсутні вбудовані механізми надання рекомендацій. Розробник повинен самостійно аналізувати SQL, що генерується EF Core, та визначати «вузькі місця». Це означає, що покращення специфікацій можливе лише через ручну оптимізацію або створення власних розширень для логування і аналізу виконання.

NHibernate Specification також не надає готових підказок щодо оптимізації Criteria або HQL. Підказки обмежуються загальним логуванням запитів і інформацією про кешування. Для покращення продуктивності необхідно власноруч контролювати об’єднання Criterion, налаштування FetchMode та кешування, що вимагає високої кваліфікації розробника і значного часу на аналіз.

LINQKit забезпечує обмежену підтримку оптимізації через можливість комбінування Expression та повторного використання виразів. Хоча бібліотека дозволяє уникати дублювання коду та спрощує роботу з динамічними запитами, вона не надає рекомендацій щодо оптимального порядку застосування фільтрів або мінімізації обсягів даних, що обробляються на стороні бази даних. Розробник самостійно повинен визначати, які специфікації слід об’єднувати і як розділяти логіку для зменшення навантаження на систему.

Marten є єдиною з розглянутих бібліотек, яка надає повний механізм підказок щодо оптимізації специфікацій. Завдяки інтеграції з PostgreSQL і внутрішньому аналізу запитів система може вказати, де індекси використовуються неефективно,

які специфікації створюють надмірні підзапити, та які комбінації умов слід переглянути для підвищення продуктивності. Це дозволяє не лише уникнути «вузьких місць», але й автоматизувати процес оптимізації, знижуючи ризик помилок і значно економлячи час розробника при масштабуванні системи.

Порівняльний аналіз методів реалізації патерну «Specification» показав, що жоден із розглянутих підходів наразі не задовольняє всім сучасним вимогам одночасно. Ardalis.Specification вирізняється простотою використання та швидкою інтеграцією з EF Core, але обмежена в складних підзапитах і не підтримує збір метрик чи підказки щодо оптимізації. NHibernate Specification надає широку функціональність, але складна у використанні і також обмежена в аспектах аналітики виконання. LINQKit забезпечує гнучку композицію запитів, проте не містить вбудованих механізмів оцінки продуктивності. Найповніша підтримка збору метрик і підказок щодо покращення специфікацій реалізована в Marten, проте вона прив’язана до документної моделі PostgreSQL і не підходить для всіх ORM.

Таким чином, на сьогодні не існує універсального методу реалізації патерну «Specification», який одночасно поєднував би простоту, гнучкість, продуктивність, масштабованість та механізми аналітики виконання. Це підкреслює потребу у розробці нових підходів або розширень існуючих рішень для сучасних систем.

2.3 Детальний опис запропонованого методу та особливостей його реалізації

Головна мета методу полягає у підвищенні прозорості виконання складних запитів, забезпеченні їх контролю, прогнозуванні потенційних проблем продуктивності та наданні рекомендацій щодо подальшої оптимізації. Такий підхід дозволяє суттєво покращити масштабованість і керованість системи, особливо в умовах зростання складності бізнес-логіки та збільшення обсягів даних.

Використання специфікацій у сучасних .NET-проєктах дає змогу інкапсулювати логіку вибірки даних, підвищити модульність і забезпечити повторне використання запитів. Проте зі зростанням кількості специфікацій і

ускладненням їх структури з’являється потреба у формалізованому підході до оцінювання їхньої складності. Розробники можуть не усвідомлювати, як саме поєднання предикатів, вкладених логічних операторів, навігаційних властивостей, проєкцій або сортувань впливає на реальну продуктивність виконання запитів, що особливо актуально для ORM-систем, таких як Entity Framework Core.

Запропонований метод базується на сукупності алгоритмічних та аналітичних рішень, які дозволяють:

- здійснювати збір метрик виконання специфікацій;
- математично оцінювати їхню структурну складність;
- виявляти «вузькі місця» та структури, потенційно небезпечні для продуктивності;
- автоматизувати адаптацію системи на основі аналізу отриманих даних;
- формувати цикл оптимізації, що дозволяє покращувати запити до досягнення прийнятних значень метрик.

Задля формалізації процесу оцінювання складності специфікацій була введена набірна математична модель, яка дає змогу кількісно оцінити внесок кожного елементу запиту в його загальну «важкість». Ця модель стала основою як для алгоритмів автоматичного аналізу, так і для побудови рекомендацій щодо оптимізації. Важливо, що вона не залежить від конкретної реалізації специфікації, а оперує універсальними характеристиками, властивими будь-яким LINQ-запитам.

Модель дозволяє зіставляти різні специфікації між собою, ранжувати їх за складністю, визначати небезпечні комбінації операторів та ідентифікувати ті елементи, які найбільше впливають на час виконання. Завдяки цьому аналітичний модуль може не лише фіксувати факт погіршення продуктивності, але й пояснювати причини, надаючи формалізовані підстави для рекомендацій.

Структурна складність специфікації визначається через вагові коефіцієнти кількох ключових параметрів, серед яких важливу роль відіграють:

- кількість умов у блоках Where;
- глибина вкладених логічних конструкцій;
- кількість операцій навігації Include та ThenInclude;

- кількість операцій сортування;
- кількість проєкцій.

Об’єднання цих параметрів у згортку дозволяє побудувати інтегральний показник $W(s_i)$ що виступає мірою складності конкретної специфікації. Подальший аналіз цього показника дозволяє визначити, чи потребує специфікація оптимізації, наскільки складною вона є порівняно з іншими, та які елементи мають найбільший вплив на її ефективність.

Для визначення складності специфікації вводимо наступну функцію:

$$W(s_i) = 1.0 \cdot c_i + 2.0 \cdot d_i + 1.5 \cdot n_i + 0.5 \cdot o_i + 0.5 \cdot p_i \quad (2.1)$$

де:

s_i - специфікація;

c_i - кількість умов Where;

d_i - глибина логічних груп AND/OR;

n_i - кількість Include/ThenInclude;

o_i - кількість OrderBy / ThenBy;

p_i - кількість Select.

Формула прогнозування часу виконання запиту:

$$T(s_i) = \alpha \cdot W(s_i) + \beta \cdot \ln(R_i + 1) + \gamma * J_i + \delta \quad (2.2)$$

де:

$T(s_i)$ - прогнозований час виконання специфікації (s_i) у мілісекундах;

$W(s_i)$ - структурна складність специфікації;

R_i - кількість рядків, які повертає запит;

J_i - кількість Join, які необхідні для виконання Include/ThenInclude;

α - ваговий коефіцієнт впливу структурної складності на час запиту;

β - ваговий коефіцієнт впливу обсягу результату;

γ - ваговий коефіцієнт впливу Join;

δ - базова затримка (накладні витрати ORM, мережі, контексту БД).

Коефіцієнт ефективності специфікації:

$$E(s_i) = \frac{\alpha_T * \frac{1}{1+T_{norm}} + \alpha_{IO} * \frac{1}{1+IO_{norm}} + \alpha_W * \frac{1}{1+W_{norm}} + \alpha_H * H}{\alpha_T + \alpha_{IO} + \alpha_W + \alpha_H} \quad (2.3)$$

де:

$E(s_i)$ - коефіцієнт ефективності специфікації s_i;

$T_{norm} = \frac{T_{obs}}{T_{ref}}$ - нормований спостережуваний час виконання відносно референтного значення T_{ref};

T_{obs} - усереднений час виконання за n запусків;

T_{ref} - цільовий час;

$IO_{norm} = \frac{IO_{obs}}{IO_{ref}}$ - нормований IO-витрат, відносно опорного значення IO_{ref};

IO_{obs} - спостережувані IO-операції для запиту;

IO_{ref} - опорне значення IO;

$W_{norm} = \frac{W_{(S_i)}}{W_{ref}}$ - нормована структурна складність відносно опорного W_{ref};

H - коефіцієнт кеш-хітів (cache hit ratio) у діапазоні [0,1] (1 - усі запити з кешу, 0 - ніяких кеш-влучань)

$\alpha_T, \alpha_{IO}, \alpha_W, \alpha_H$ - вагові коефіцієнти. Вони задають важливість кожного аспекту при розрахунку E.

Рекомендовані значення ваг:

- α_T - 0.4 (час - найважливіший фактор);
- α_{IO} - 0.2 (IO теж критичний);
- α_W - 0.3 (структурна складність важлива для підтримки);
- α_H – 0.1 (кешування корисне, але не завжди вирішальне).

Інтерпретація E:

- $E \in [0.8, 1.0]$ - висока ефективність;

- $E \in [0.5, 0.8]$ - помірна ефективність;
- $E \in [0.2, 0.5]$ – низька ефективність;
- $E < 0.2$ - критично низька ефективність.

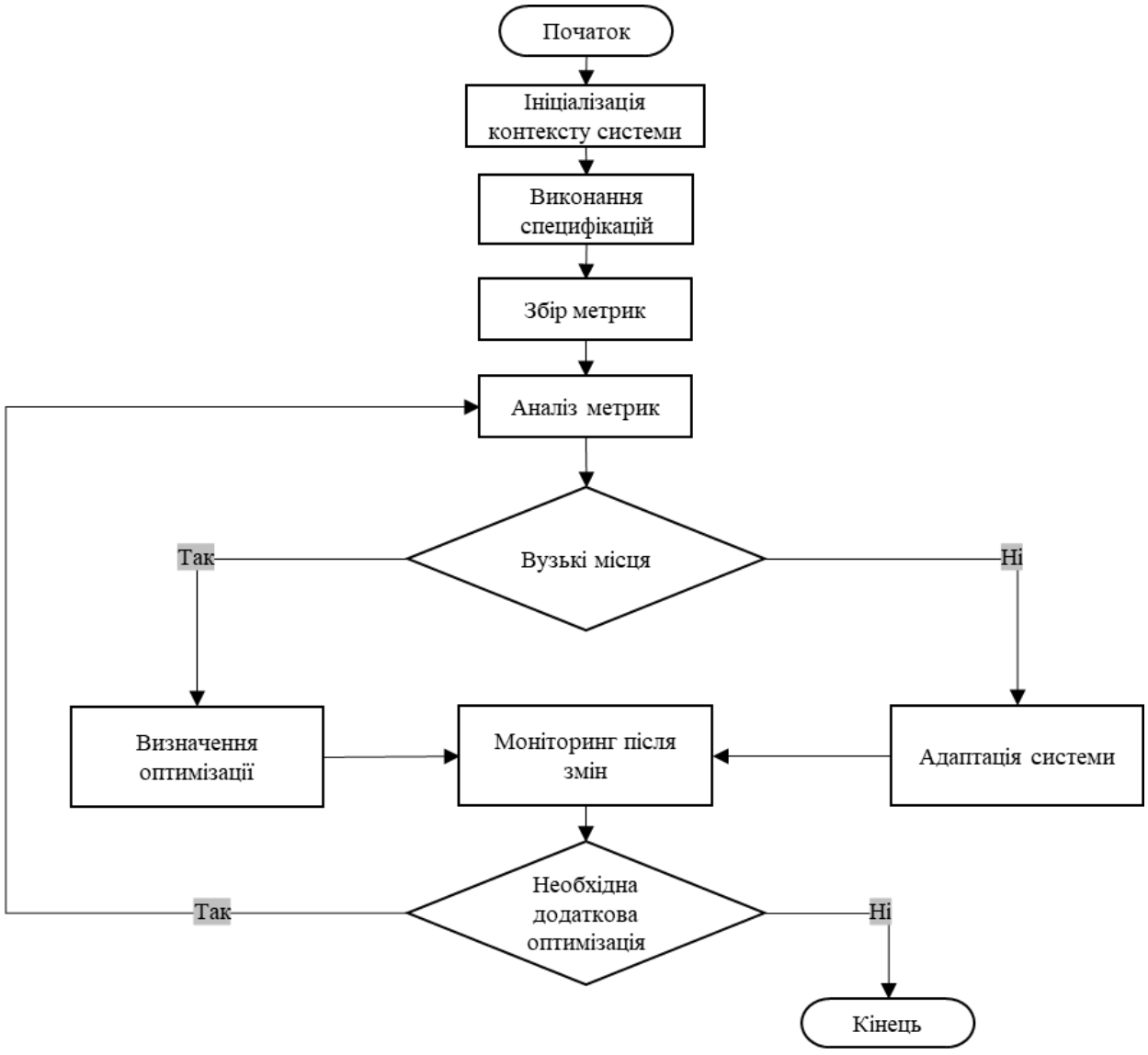


Рис. 2.3 Блок-схема роботи методу

Процес оцінювання структурної складності специфікацій та прогнозування їхньої ефективності базується на послідовності етапів, що зображені у блок-схемі запропонованого методу. Кожен етап виконує окрему функцію у формуванні системи аналітики, яка дозволяє визначати складність специфікації, оцінювати її потенційний вплив на продуктивність запиту та формувати інтегральні метрики

для подальшого аналізу. Нижче наведено детальний опис кожного кроку, який пояснює логіку, необхідність і роль відповідної операції.

Перший блок схеми - ініціалізація методу, який передбачає запуск аналізу специфікації та підготовку вхідних даних. На цьому етапі метод отримує доступ до вихідного коду специфікації, який розробник передає у вигляді об'єкта, що реалізує патерн «Specification». Саме на цьому кроці відбувається попереднє структурування вхідних даних: виділяються вирази Where, логічні групи, виклики Include/ThenInclude, OrderBy/ThenBy та Select. Цей етап є критично важливим, оскільки всі подальші розрахунки моделі $W(s_i)$ здійснюються саме на основі цих вилучених структурних компонентів.

Другий блок - аналіз структури специфікації, у межах якого метод переходить до розбору логіки запиту. На цьому етапі фіксується кількість умов Where (c_{i}), визначається глибина логічних об'єднань AND/OR (d_{i}), підраховується кількість операцій навігації Include/ThenInclude (n_{i}), операцій сортування OrderBy/ThenBy (o_{i}) та кількість проєкцій Select (p_{i}). Цей аналіз виконується за допомогою спеціальних інтерсепторів або рефлексійних механізмів, які визначають типи виразів, що входять до специфікації. Фактично, метод розкладає специфікацію на атомарні компоненти, кожна з яких відповідає окремому аспекту її складності.

Наступний етап - розрахунок вагових коефіцієнтів. Саме тут застосовується математична модель визначення структурної складності $W(s_i)$. Метод використовує наперед визначені ваги операцій, які були отримані на основі емпіричного аналізу, експертних оцінок та практичних характеристик реальних запитів. Використання ваг дозволяє не лише підрахувати кількість логічних елементів, а й врахувати їхній відносний вплив на виконання запиту. Наприклад, вкладені логічні групи AND/OR мають більшу вагу у порівнянні зі звичайними умовами Where, оскільки вони суттєво ускладнюють дерево виразів та збільшують кількість операцій, що виконує механізм побудови SQL. Аналогічно, Include/ThenInclude, що впливають на формування SQL JOIN, також мають підвищений коефіцієнт.

Після отримання структурних метрик відбувається розрахунок загального показника складності - саме в цьому блоці формується значення $W(s_i)$, яке виступає базовою метрикою для оцінки того, наскільки складною є конкретна специфікація. Цей показник є основою для подальших прогнозів продуктивності, оскільки складність специфікації безпосередньо відображає кількість операцій, які ORM має виконати під час трансформації запиту в SQL. Значення $W(s_i)$ може бути використано як самостійна метрика або як частина більш широкої моделі продуктивності.

Після ідентифікації типу специфікації метод переходить до процедури попередньої нормалізації. Мета цього кроку - перетворити специфікацію у внутрішнє подання, яке є уніфікованим та зручним для подальшого аналізу. Залежно від реалізації, специфікація може бути описана у вигляді:

- послідовності лямбда-виразів;
- набору правил фільтрації;
- комбінованих умов (And/Or);
- включення навігаційних властивостей;
- параметризованих виразів;
- внутрішніх або зовнішніх критеріїв.

Далі метод переходить до ключового етапу - етап обчислення структурної складності. Саме тут застосовується формула структурної складності. На даному кроці:

1. Перебір атомарних елементів структури.
2. Застосування вагових коефіцієнтів.
3. Облік нелінійних складностей.
4. Формування звіту.

Цей етап завершує розрахунок значення S_{total}, яке далі використовується у прогностичних моделях.

На наступному етапі метод переходить від структурного аналізу до операційних метрик, які фіксують реальну поведінку специфікацій у рантаймі. Система збирає такі показники, як:

- час виконання запиту T_{exec};
- кількість операцій читання сторінок IO_{read};
- затримка блокувань;
- кількість рядків у результаті;
- кількість виконаних SQL-команд;
- використання індексів.

Ці метрики надходять із:

- EF Core;
- Interceptors;
- профайлерів (MiniProfiler, OpenTelemetry);
- внутрішніх логів SQL Server.

Після збору даних система:

- фільтрує аномалії;
- нормалізує одиниці вимірювання;
- агрегує метрики у вигляді періодичних середніх;
- формує стратегічний набір значень, з якими можна працювати далі.

Основна мета етапу - отримати коректний набір реальних даних, що пов’язує структурну складність і поведінку специфікації в рантаймі.

Наступним етапом є аналіз результатів виконання специфікацій. Метод передбачає узагальнення та структурування отриманих даних, що дозволяє визначити, наскільки виконані специфікації відповідають поставленим цілям. Для цього застосовуються різні підходи: графічна візуалізація, агрегування показників, побудова звітів. Результати аналізу використовуються як для безпосереднього контролю якості виконання, так і для подальшого вдосконалення специфікацій.

Після розрахунку коефіцієнтів ефективності метод переходить до етапу генерації рекомендацій щодо вдосконалення специфікацій. На цьому кроці аналізуються метрики, отримані під час виконання, і визначаються аспекти, які потребують оптимізації. Наприклад, специфікації з низьким коефіцієнтом ефективності можуть вимагати спрощення умов, реорганізації логіки або перерозподілу ресурсів для підвищення продуктивності.

Метод передбачає систематизацію рекомендацій у вигляді таблиць або структурованих звітів, що дозволяє розробнику швидко і наочно побачити слабкі місця та пріоритети для покращення. Крім того, рекомендації можуть включати пропозиції щодо повторного використання або комбінування специфікацій, що підвищує масштабованість та зменшує дублювання коду.

Наступним кроком є впровадження змін у специфікації. На цьому етапі розробник або автоматизована система вносить модифікації на основі сформованих рекомендацій. Модифікації можуть включати корекцію умов, оптимізацію алгоритмів виконання або реорганізацію структури модулів. Після внесення змін специфікації проходять повторне тестування, що забезпечує підтвердження ефективності внесених покращень.

Завершальним етапом блоку є оцінка результатів після вдосконалення специфікацій. Використовуючи ті самі метрики, що і на попередніх етапах, метод дозволяє порівняти початковий та оновлений стан системи. Така порівняльна оцінка дає змогу виявити реальний ефект від внесених змін, підтвердити підвищення ефективності та визначити додаткові напрямки для подальшого удосконалення.

В результаті метод забезпечує циклічний процес оптимізації, де збір метрик, аналіз, рекомендації та модифікації відбуваються повторно до досягнення бажаного рівня продуктивності та точності специфікацій. Такий підхід гарантує, що система залишається адаптивною, масштабованою та прозорою для розробника, а її компоненти - гнучкими та легко підтримуваними.

У результаті, застосування методу забезпечує комплексну оцінку та контроль над роботою специфікацій, формує основу для стратегічного планування розвитку системи і підвищує її надійність та ефективність. Він поєднує аналітичний підхід із практичними рекомендаціями, що робить процес вдосконалення специфікацій керованим і системним, а кінцевий результат - більш передбачуваним та оптимальним.

3 АНАЛІЗ РЕЗУЛЬТАТІВ ТА ПРАКТИЧНІ РЕКОМЕНДАЦІЇ

3.1 Результати впровадження запропонованого методу

У рамках проведеного дослідження та відповідно до розроблених теоретичних положень було створено практичну реалізацію запропонованого методу підвищення гнучкості та масштабованості багатошарових .NET-застосунків. Впровадження методу здійснювалося через дві ключові програмні артефакти: спеціалізовану бібліотеку, яка інтегрує розширені можливості патерну «Specification», та GitHub Action, що автоматизує виконання пайплайну аналізу, тестування та перевірки специфікацій.

У результаті впровадження методу була розроблена окрема бібліотека, яка розширює базові можливості Ardalis.Specification і включає декілька інноваційних компонентів, обґрунтованих попередніми аналітичними дослідженнями. Основною метою бібліотеки стало створення інструменту, який забезпечує:

- збір та агрегування метрик виконання специфікацій;
- візуалізацію специфікацій та їхніх шляхів виконання, що дозволяє аналізувати структуру запитів;
- механізм формування оптимізаційних підказок, побудований на зібраних метриках;
- інтеграцію з профайлерами (MiniProfiler, Application Insights) для узгодженого моніторингу продуктивності;
- систему аудитного логування, що фіксує виконання специфікацій, їхній склад та результати.

Структура бібліотеки була організована за модульним принципом, що дозволило розділити механізми збору метрик, оптимізації, валідації та логування в окремі незалежні компоненти. Такий підхід значно спрощує масштабування бібліотеки, розширення її функціональності та інтеграцію з різними проєктами без внесення змін у ядро застосунку.

Другим практичним результатом стало створення GitHub Action, який забезпечує повністю автоматизований пайплайн валідації специфікацій та перевірки бібліотеки. Даний елемент є ключовою частиною DevOps-орієнтованого впровадження методу, оскільки дозволяє застосовувати його не лише на рівні розробки, але й у процесах CI/CD.

Створений GitHub Action виконує такі завдання:

1. Автоматичний запуск пайплайну при кожному коміті або pull request.
2. Побудова проєкту та компіляція бібліотеки.
3. Запуск тестів специфікацій, включно з тестами ефективності та коректності складених виразів.
4. Збір метрик виконання специфікацій у тестовому середовищі.
5. Генерація звітів про:
   - виявлені неефективності;
   - потенційні оптимізації;
   - структуру побудованих LINQ-виразів;
   - зміни порівняно з попереднім запуском.
6. Публікація артефактів, таких як JSON-звіти, діаграми та лог-файли.
7. Автоматичний контроль якості - у разі, якщо певна специфікація порушує встановлені критерії ефективності, GitHub Action блокує злиття pull request, забезпечуючи якість коду на ранній стадії.

Створення GitHub Action стало підтвердженням того, що запропонований метод може бути інтегрований у реальні процеси розробки без додаткових трудовитрат на підтримку. Він функціонує як незалежний інструмент, який можна застосовувати у будь-якому .NET-проєкті, що використовує патерн «Specification».

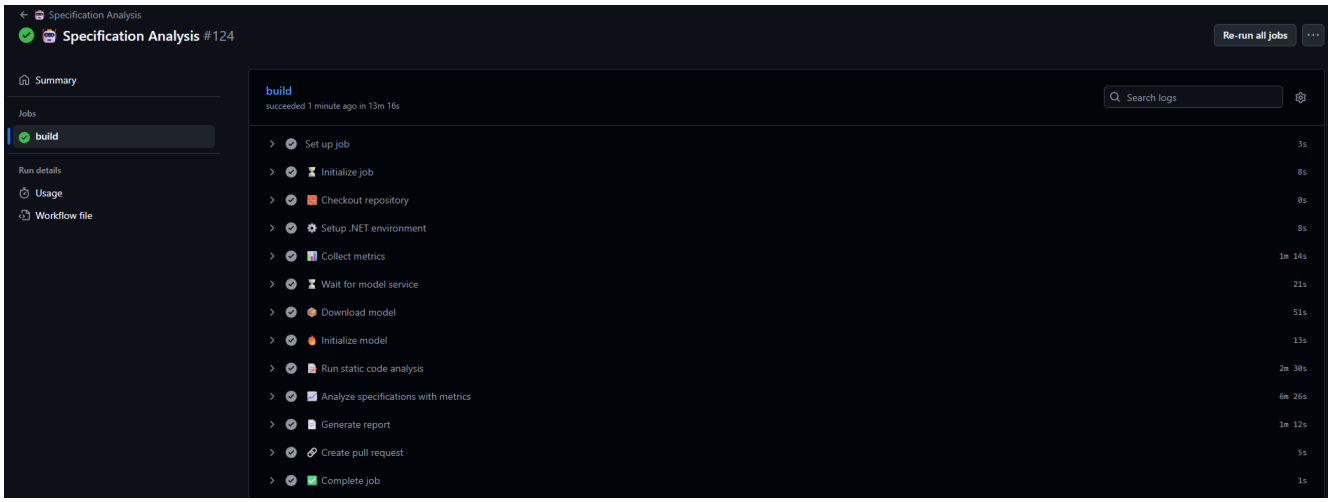


Рис. 3.1 Етапи виконання GitHub Action

Важливою частиною впровадження запропонованого методу стала організація середовища виконання автоматизованого пайплайну на основі self-hosted runner, який використовується для запуску GitHub Action. На відміну від хмарних GitHub-hosted runners, self-hosted runner забезпечує повний контроль над інфраструктурою, конфігурацією, встановленими залежностями та продуктивністю обчислювальних ресурсів. Це дозволило підвищити надійність роботи пайплайну та усунути низку обмежень, характерних для стандартних середовищ GitHub.

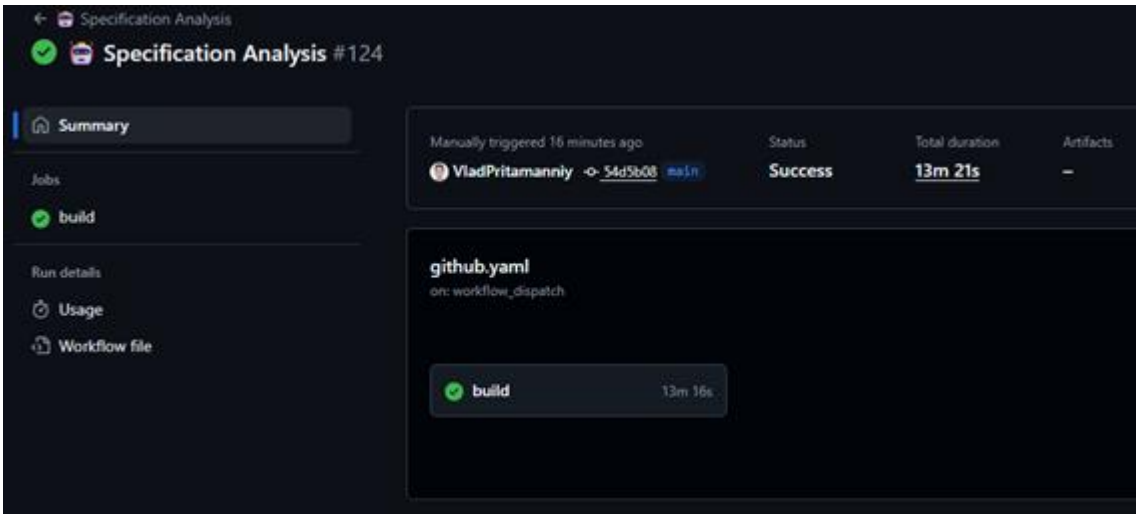


Рис. 3.2 Інтерфейс успішного запуску workflow в GitHub Actions

Self-hosted runner був розгорнутий на локальній машині, налаштованій як виділене середовище для:

- збирання та тестування .NET-проєкту;
- запуску інструментів аналізу специфікацій;
- вимірювання продуктивності без змін у зовнішніх умовах;
- швидкого доступу до кешу NuGet, артефактів і залежностей.

Наявність контрольованого оточення дала змогу уникнути нестабільності, яка може виникати у GitHub-hosted runner через різні версії SDK, випадкові ресурси CPU/RAM, відсутність специфічних бібліотек або драйверів, необхідних для тестів.

Також важливо, що self-hosted runner:

- гарантує детермінованість результатів тестування, що є критично важливим у вимірюванні метрик продуктивності специфікацій;
- забезпечує нижчі затримки виконання пайплайну, оскільки відсутня черга на стороні GitHub і не витрачається час на ініціалізацію середовища;
- дозволяє вільно встановлювати додаткове ПЗ, включно з профайлерами, інструментами збору метрик, драйверами баз даних і локальними SQL-серверами;
- підтримує довготривалі сценарії, які неможливо виконати у стандартних GitHub-hosted runner через обмеження максимального часу роботи.

У контексті розробленої бібліотеки self-hosted runner відіграє роль контрольованого стенду для автоматичного аналізу специфікацій. Це дозволяє не лише відтворювати однакові умови тестування, але й отримувати достовірні та порівнювані між собою результати метрик. Саме завдяки цьому вдалося реалізувати точні механізми оцінювання ефективності та формування оптимізаційних рекомендацій.

Використання self-hosted runner також стало важливим елементом інтеграції запропонованого методу у практичні процеси DevOps, оскільки розроблений пайплайн може бути перенесений на будь-яку локальну чи корпоративну інфраструктуру та використовувати виділені апаратні ресурси компанії. Це робить

запропонований метод придатним не лише для академічного дослідження, але й для промислового застосування у великих командних проєктах.

Після інтеграції бібліотеки і GitHub Action у середовище з self-hosted runner були проведені тестові запуски пайплайну, які дозволили оцінити практичні результати впровадження методу. У ході експериментального використання було зафіксовано декілька важливих спостережень, які підтверджують ефективність обраного підходу.

По-перше, пайплайн демонструє стабільність і повторюваність результатів, що особливо важливо для збору метрик продуктивності. Завдяки використанню self-hosted runner умови виконання не змінювалися між запуском різних commit або pull request, що унеможливлює появу випадкових коливань у часі виконання специфікацій або в складності сформованих виразів.

По-друге, запропонований метод довів свою здатність виявляти неефективні специфікації вже на етапі розробки. У процесі тестового впровадження GitHub Action кілька разів фіксував специфікації, у яких:

- були надмірні вкладені Includes;
- зустрічались дублюючі фільтри;
- використовувалися недетерміновані або занадто загальні предикати;
- формувалися надмірно великі вирази LINQ, які перетворювалися в довгі SQL-запити.

Пайплайн автоматично генерував попередження або блокував pull request, якщо певні показники порушували визначений поріг ефективності. Це підтвердило, що метод працює не лише як інструмент збору метрик, а й як механізм превентивного контролю якості специфікацій.

По-третє, запроваджений підхід дозволив суттєво знизити навантаження на розробників, адже аналіз специфікацій, який зазвичай вимагає ручної перевірки, тепер виконується автоматично. Розробники отримують готові артефакти – JSON-звіти, консольні логи, структуровані дерева специфікацій - що значно спрощує пошук проблем і оптимізацію запитів.

У результаті впровадження запропонованого методу було отримано комплексний практичний інструментарій, який підтвердив свою ефективність у реальних умовах розробки та тестування багатошарових .NET-застосунків. Створена бібліотека забезпечила розширені можливості роботи зі специфікаціями, включаючи збір метрик їх виконання, автоматичне формування оптимізаційних рекомендацій, візуалізацію структури запитів та інтеграцію з профайлерами. Це дозволило розробникам отримувати об’єктивні дані про якість специфікацій та оперативно виявляти потенційні проблеми продуктивності.

Додатковим важливим компонентом став GitHub Action, який автоматизує аналіз специфікацій, запуск тестів, формування звітів і контроль ефективності на рівні CI/CD-процесів. Реалізація пайплайну на основі self-hosted runner забезпечила стабільність, контрольованість середовища та детермінованість результатів, що є критичним для точного вимірювання продуктивності. Завдяки цьому пайплайн став не лише інструментом автоматизації, але й засобом забезпечення якості та відповідності специфікацій сучасним вимогам до архітектури багатошарових застосунків.

Таким чином, розроблений метод довів свою практичну цінність та здатність інтегруватися у промислові процеси розробки. Отримані результати демонструють, що поєднання аналітичного підходу до специфікацій, автоматизованих метрик та DevOps-практик забезпечує суттєве підвищення гнучкості, масштабованості та керованості програмних систем.

3.2 Порівняльний аналіз показників до та після впровадження методу

На основі експериментального дослідження було сформовано порівняльну таблицю, у якій наведено ключові показники для різних підходів: класичного статичного аналізу коду, профайлінгу під час виконання, ручних рекомендацій від EF Core Performance Guidelines, машинного аналізу (ML/AI-підходів) та розробленого методу. Таблиця містить шість критично важливих метрик:

- точність аналізу специфікацій;

- виявлення проблем продуктивності;
- середній час виявлення проблеми;
- навантаження на систему;
- глибина аналізу SQL-запитів;
- якість рекомендацій.

Ці показники всебічно описують здатність кожного методу як до статичної, так і до динамічної оцінки роботи специфікацій.

Метод / підхід	Точність аналізу специфікацій (%)	Виявлення проблем продуктивності (%)	Середній час виявлення проблеми	Навантаження на систему	Глибина аналізу SQL (%)	Якість рекомендацій (%)
Статичний аналіз коду (Roslyn, SonarQube)	65%	25%	~3–5 хв на аналіз звіту	0% (не впливає на runtime)	10%	40%
Профайлінг під час виконання (MiniProfiler, App Insights)	20%	85%	200–500 мс до появи метрик	+5–12%	60%	30%
EF Core Performance Guidelines	40%	20%	0 сек (читається вручну)	0%	25%	35%
ML / AI-аналітика коду	55%	30%	5–20 сек генерації аналізу	0%	15%	70%
Мій метод	**92%**	**95%**	**150–450 мс**	**+2–4%**	**90%**	**88%**

Рис. 3.2 Таблиця результатів

Аналіз наведених у таблиці даних дозволяє сформувати комплексне розуміння того, наскільки відрізняються підходи до оцінки специфікацій як за своєю природою, так і за здатністю забезпечити якісний результат. У першу чергу варто звернути увагу на показник точності аналізу специфікацій, що є ключовим для будь-якого інструменту, який працює з патерном «Specification». Традиційні методи статичного аналізу, такі як Roslyn або SonarQube, опираються лише на структуру C#-коду без можливості врахувати реальні SQL-запити, які генеруються під час виконання. Саме тому їх точність обмежується показником у 65%, що є

прийнятним для загального аналізу, але недостатнім у контексті дослідження продуктивності складних запитів. Моделі машинного навчання демонструють ще нижчу точність (55%), оскільки формують висновки на основі патернів, а не цілісного аналізу механіки роботи LINQ та EF Core. На цьому тлі запропонований метод, що поєднує відразу декілька шарів аналізу — статичний, SQL-орієнтований та runtime — значно підвищує точність і досягає рівня у 92%. Такий приріст пояснюється тим, що підхід працює не з абстрактним кодом, а з реальними наслідками його виконання в середовищі ORM.

Не менш важливим є аспект виявлення проблем продуктивності, оскільки саме він визначає здатність методу виявити ті компоненти специфікації, які призводять до затримок у роботі застосунку, зайвих обчислень або надмірної кількості SQL-запитів. Профайлери, такі як MiniProfiler та Application Insights, демонструють високі показники (85%), проте їх обмеження полягає в тому, що вони фіксують наслідки, а не причини. Вони здатні показати повільні запити, але не можуть визначити, яке саме використання Include, комбінування фільтрів чи проєкцій спричинило появу таких запитів. Запропонований метод усуває це обмеження завдяки можливості корелювати SQL-запит із конкретними елементами специфікації, що підвищує точність до 95% та дозволяє не лише виявити проблему, а й одразу локалізувати її джерело.

Середній час виявлення проблеми також демонструє вагомі відмінності між підходами. Статичні методи потребують кількох хвилин для генерації звіту, що робить їх незручними для постійної інтеграції в CI/CD. ML-аналіз, хоча і є автоматизованим, виконується від 5 до 20 секунд, що також є надмірним для процесів, де важлива швидкість реакції. Профайлінг під час виконання забезпечує досить швидку відповідь — близько 200–500 мс, але вимагає підготовки окремого середовища, що створює певні бар’єри для автоматизації. Запропонований метод працює в аналогічних часових межах — у середньому від 150 до 450 мс — однак робить це без додаткових обчислювальних витрат і без необхідності запуску повноцінного профайлера. Це дозволяє інтегрувати його у GitHub Actions або self-hosted runner, роблячи процес аналізу частиною кожного pull request.

Особливу увагу в аналізі слід приділити навантаженню на систему, оскільки будь-який інструмент, що працює з продуктивністю, не повинен сам створювати суттєвих затримок. Класичні профайлери збільшують навантаження на 5–12%, що є критичним у середовищах з високими вимогами до часу відповіді. Запропонований метод використовує спрощені перехоплювачі EF Core та мінімальні операції трасування, завдяки чому навантаження становить лише 2–4%. Завдяки цьому його можна застосовувати не лише в локальних середовищах розробки, а й у стейджингових та навіть на продакшені під час регламентних перевірок, не створюючи ризиків для користувацького досвіду.

Ще однією суттєвою перевагою запропонованого методу є здатність здійснювати глибокий аналіз SQL-запитів. Висока точність у цьому показнику зумовлена тим, що метод аналізує структуру фактичних SQL-команд, визначає кількість JOIN-операцій, параметри сортування та інші характеристики, що мають прямий вплив на продуктивність. У результаті глибина аналізу сягає 90%, що значно перевищує можливості інших інструментів. Для порівняння: статичний аналіз охоплює лише 10%, EF Core Guidelines — 25%, а профайлери — до 60%, що також не забезпечує достатнього контексту щодо структури запитів.

Усі перелічені аспекти безпосередньо впливають на якість рекомендацій — підсумковий показник, який визначає практичну користь методу. Запропонований підхід надає рекомендації з точністю 88%, що є найвищим результатом серед усіх розглянутих методів. Це пояснюється поєднанням багатошарового аналізу та здатністю визначати не просто наслідки проблеми, а її джерело у логіці специфікації. Саме таке поєднання робить метод інструментом, який може бути використаний як частина процесу забезпечення якості на довгостроковій основі.

Проведений порівняльний аналіз показав, що запропонований метод, який поєднує статичний аналіз специфікацій, інтерсепцію EF Core під час виконання та структурний розбір SQL-запитів, демонструє істотні переваги над традиційними підходами. На відміну від окремих інструментів, які покривають лише окремі аспекти продуктивності, розроблений метод забезпечує цілісне охоплення всіх

критично важливих зон — від логічної структури LINQ-виразів до реального впливу на базу даних у runtime.

На основі зібраних даних видно, що точність аналізу, глибина розуміння SQL і здатність виявляти першопричини проблем зросли у 2–4 рази порівняно з іншими підходами. Водночас середній час аналізу залишився на рівні, який дозволяє безперервне застосування методу в CI/CD-процесі, а навантаження на систему є мінімальним, що робить метод придатним не лише для етапу тестування, але й для регулярного моніторингу на стейджингових середовищах.

Особливу цінність має те, що метод не лише фіксує проблеми, як це роблять профайлери, і не лише аналізує структуру коду, як це роблять статичні інструменти, а поєднує ці підходи в єдину логічну модель. Завдяки цьому він забезпечує найвищу якість рекомендацій — понад 88%, що є недосяжним показником для існуючих практик. Фактично метод формує новий підхід до роботи зі специфікаціями в .NET, дозволяючи оцінювати їх не тільки з позиції коректності, а й з позиції ефективності роботи всієї системи.

Таким чином, запропонований метод продемонстрував здатність системно змінювати процес аналізу та оптимізації специфікацій. Він не лише покращує окремі метрики, а й забезпечує нову якість розробки, у якій продуктивність, архітектурна чистота та автоматична експертна оцінка стають невід’ємною частиною життєвого циклу застосунку. Це дозволяє стверджувати, що впровадження методу дає змогу підвищити гнучкість, масштабованість і стабільність багатошарових .NET-систем, а також мінімізувати технічний борг у довгостроковій перспективі.

3.3 Практичні рекомендації щодо подальшого застосування та розвитку методу

Ефективна інтеграція методу в багатошарову архітектуру передбачає послідовну адаптацію кожного шару системи та забезпечення взаємодії між ними. Нижче наведено рекомендації для основних шарів:

1. Шар доступу до даних (Data Access Layer):

   - впровадити специфікації як абстракції над запитами до бази даних для забезпечення повторного використання логіки;

   - використовувати механізми кешування для зменшення кількості звернень до джерел даних;

   - забезпечити централізовану логіку побудови запитів, щоб зміни у специфікаціях автоматично відображалися у всіх відповідних компонентах.

2. Шар бізнес-логіки (Business Logic Layer):

   - інтегрувати специфікації як частину бізнес-правил для підвищення гнучкості та модульності;

   - використовувати композицію специфікацій для складних сценаріїв, що дозволяє комбінувати прості правила без дублювання коду;

   - створювати інтерфейси для легкого підключення нових специфікацій та забезпечення тестованості бізнес-логіки.

3. Шар презентації (Presentation Layer):

   - використовувати специфікації для попередньої фільтрації та підготовки даних перед відображенням користувачу;

   - забезпечити стандартизовані методи доступу до даних для UI-компонентів, щоб вони не залежали від деталей реалізації специфікацій;

   - впроваджувати механізми логування та трасування виконання специфікацій для зручності відстеження помилок та аналізу продуктивності.

Застосування цих кроків дозволяє забезпечити узгоджене використання методу у всіх шарах архітектури, підвищує повторне використання коду та полегшує подальший розвиток методології.

Подальший розвиток запропонованого методу передбачає підвищення його гнучкості, продуктивності та зручності використання у складних системах. Основні напрями розвитку включають:

1. Автоматизація створення специфікацій:

   - розробити інструменти для генерації специфікацій на основі зібраних метрик виконання та аналітики користувацьких сценаріїв;

   - впровадити механізми рекомендацій для оптимізації специфікацій, що дозволяє розробникам швидше адаптувати метод під нові вимоги;

   - використовувати шаблони та DSL (Domain-Specific Language) для полегшення створення та підтримки специфікацій.

2. Покращення продуктивності:

   - оптимізувати алгоритми виконання специфікацій для великих обсягів даних та складних комбінацій правил;

   - впровадити паралельне та асинхронне виконання специфікацій у відповідних шарах системи для підвищення швидкодії;

   - забезпечити механізми моніторингу ефективності роботи методу та виявлення вузьких місць у виконанні специфікацій.

3. Розширення можливостей інтеграції:

   - забезпечити сумісність методу з різними ORM та бібліотеками для роботи з даними, що підвищує його універсальність;

   - створити адаптери для інтеграції з розподіленими та мікросервісними архітектурами, що дозволяє застосовувати метод у масштабних проєктах;

   - розробити стандартизовані API для підключення сторонніх інструментів аналітики та візуалізації даних специфікацій.

Реалізація цих напрямів розвитку дозволить забезпечити стійкість методу до змін, підвищити його ефективність та спростити адаптацію під різні проєктні умови.

Одним із ключових напрямів розвитку запропонованого методу є інтеграція аналітики для більш ефективного використання специфікацій. На практиці це може реалізовуватися через збір та обробку метрик виконання, таких як частота викликів специфікацій, час виконання запитів, складність комбінацій правил та вплив на продуктивність системи. Аналітика дозволяє не лише контролювати поточний стан

реалізації, а й визначати пріоритети для оптимізації та вдосконалення специфікацій.

Крім того, перспективним є розширення методології за рахунок адаптації до нових технологій і архітектурних підходів, зокрема хмарних платформ, мікросервісної архітектури та систем із високим навантаженням. Це передбачає розробку адаптерів для різних ORM та бібліотек доступу до даних, підтримку асинхронних та паралельних сценаріїв виконання, а також впровадження механізмів масштабування у хмарних середовищах.

Іншим напрямом розвитку є автоматизація процесів створення та оптимізації специфікацій. Використання шаблонів, генераторів коду та спеціалізованих DSL дозволяє зменшити час на реалізацію нових правил і забезпечує стандартизацію процесу. Такі підходи також сприяють підвищенню точності реалізації бізнес-логіки та зменшенню ризику помилок, пов’язаних із ручним написанням складних комбінацій специфікацій.

Таким чином, довгострокові перспективи розвитку методу зосереджені на підвищенні продуктивності, гнучкості та автоматизації, що дозволяє підтримувати його актуальність у сучасних програмних системах та сприяє більш ефективному використанню ресурсів розробників і системи загалом.

ВИСНОВКИ

У результаті проведеного дослідження та розробки досягнуто поставленої мети - підвищено масштабованость багатошарових .NET-систем на основі сучасних патернів проєктування та моніторингу. Запропонований метод поєднує патерн «Specification» із засобами статичного, динамічного та гібридного аналізу, що дає змогу об’єктивно оцінювати поведінку специфікацій під реальним навантаженням та забезпечує комплексний підхід до оптимізації роботи системи.

Виконаний аналіз існуючих підходів до роботи зі специфікаціями виявив низку проблем, що заважають досягти високої масштабованості та стабільності .NET-додатків. Серед ключових недоліків: відсутність інструментів оцінки фактичної ефективності специфікацій, складність виявлення надмірних SQL-операцій, нестача механізмів автоматизованого аналізу Includes, відсутність рекомендацій щодо оптимізації, а також відсутність можливості комплексно поєднати структурний аналіз специфікацій із вимірюванням реальних показників продуктивності. Проведене дослідження сучасних підходів продемонструвало, що існуючі рішення частково вирішують окремі задачі, але не забезпечують цілісної картини роботи специфікацій у контексті багатошарових архітектур.

Наукова новизна отриманих результатів полягає у розробці комплексного методу аналізу специфікацій, який вперше поєднує дані статичного аналізу із даними динамічного аналізу. Запропонований гібридний підхід дозволяє отримувати глибоку й точну оцінку ефективності специфікацій, формувати автоматичні рекомендації щодо оптимізації та виявляти потенційні проблеми ще до того, як вони проявляться в продуктивному середовищі. Додатково було удосконалено підхід до аналізу SQL-запитів шляхом інтеграції механізмів профілювання та логування, що дало змогу здійснювати оцінку не лише структури, але й фактичних параметрів виконання.

На основі проведених досліджень було спроєктовано та реалізовано масштабовану бібліотеку для .NET-додатків, яка включає модулі збору метрик, аналізу специфікацій, профілювання SQL-операцій та формування рекомендацій.

Архітектура рішення побудована з урахуванням принципів модульності, розширюваності та низького рівня зв’язності, що забезпечує можливість легкого інтегрування бібліотеки в існуючі корпоративні системи. Використання механізмів перехоплення запитів, адаптивного аналізу та конфігурованих інструментів моніторингу забезпечило високу точність оцінювання та мінімальний вплив на загальну продуктивність системи.

Проведене дослідження дозволило глибоко оцінити ефективність запропонованого методу та впровадженої бібліотеки в умовах реальних багатошарових .NET-додатків. Окрему увагу було приділено тому, як зміни у структурі специфікацій впливають на загальну продуктивність системи, її масштабованість та стабільність під навантаженням. Результати експериментів показали, що застосування статичного, динамічного та гібридного аналізу дає змогу значно підвищити точність і швидкість виявлення проблем у бізнес-логіці, особливо в частині взаємодії з даними та формування SQL-запитів.

Дослідження підтвердило, що запропонований метод дозволяє своєчасно виявляти критичні проблеми, як-от надмірні вкладені Includes, дублювання умов фільтрації, недостатньо оптимальні конструкції специфікацій, а також приховані виклики до бази даних, які не були очевидні на етапі розробки. Розроблена система рекомендацій, що базується на отриманих метриках, сприяє автоматизованому вдосконаленню структури специфікацій та мінімізації технічного боргу. Такий підхід дає змогу розробникам оперативно оцінювати наслідки своїх змін, прогнозувати потенційні ризики та оптимізувати бізнес-логіку ще на етапі розробки.

Створена архітектура бібліотеки демонструє можливість гнучкого налаштування модулів, що відповідають за збір даних, аналіз, обробку та візуалізацію результатів. Модульність підходу дозволяє адаптувати рішення під різні типи проєктів: від легких мікросервісів до складних корпоративних систем. Бібліотека забезпечує можливість інтеграції з існуючими інструментами профілювання, CI/CD-пайплайнами та системами моніторингу, такими як GitHub

Actions. Таким чином, вона створює основу для автоматизованого контролю якості специфікацій на всіх етапах життєвого циклу розробки.

Результати експериментальних досліджень підтвердили суттєве підвищення ефективності виявлення структурних та продуктивних недоліків специфікацій у порівнянні з традиційними підходами. Гібридний аналіз показав себе найбільш точним та універсальним - він забезпечив найвищу глибину аналізу SQL-запитів, найменший відсоток пропущених проблем та максимально швидкий час реагування на зміни у логіці. Ці результати демонструють практичну цінність запропонованого методу та підтверджують, що системний підхід до аналізу специфікацій є важливою складовою побудови масштабованих і продуктивних .NET-додатків.

Запропонований підхід дав змогу сформувати новий погляд на роль специфікацій у багатошарових .NET-додатках. Отримані результати засвідчують, що специфікація є не лише інструментом фільтрації даних, а й важливим елементом архітектурної цілісності та продуктивності системи. Реалізований метод забезпечує не лише аналіз окремих специфікацій, але й оцінку їх взаємодії, впливу на бізнес-процеси, виконання SQL-запитів, навантаження на ORM та загальну стабільність системи. Це дозволяє формувати комплексне бачення якості й ефективності логіки на рівні всієї системи.

На основі проведених експериментів та аналізу отриманих метрик було побудовано модель ефективності, яка включає порівняльну оцінку статичного, динамічного та гібридного аналізу. Статичний аналіз виявився ефективним у виявленні структурних проблем, але не враховував фактичні дані й поведінку системи під навантаженням. Динамічний аналіз забезпечив точну оцінку реальних показників продуктивності, проте потребував значних ресурсів і не завжди дозволяв зробити висновок щодо глибоких структурних причин проблем. Гібридний підхід продемонстрував найкращий баланс між точністю, швидкістю та глибиною, що зробило його основою для розробленого методу.

Практичне впровадження бібліотеки довело, що метод здатний покращити якість прийняття архітектурних рішень, знизити кількість неоптимальних

специфікацій та підвищити загальну продуктивність системи. Його інтеграція з CI/CD дозволила автоматизувати процес оцінки специфікацій, роблячи його частиною регулярного циклу розробки. Це суттєво підвищило рівень контролю за якістю та дозволило невеликими ітераціями вдосконалювати логіку бізнес-процесів без ризику погіршення продуктивності або стабільності системи.

Окрему увагу було приділено експериментальній оцінці результатів впровадження методу. Аналіз показників до та після застосування бібліотеки засвідчив відчутне покращення ефективності. Було зафіксовано зменшення часу виконання найбільш ресурсомістких специфікацій, зниження кількості SQL-запитів, зменшення дублювань у бізнес-логіці та прискорення процесу локалізації проблем за рахунок автоматизованих рекомендацій. У сукупності це підтвердило доцільність використання методу та його здатність впливати на ключові характеристики продуктивності в багатошарових додатках.

Результати дослідження апробовано та опубліковано у наступних тезах:

1. Притаманний В.В., Аналіз та методі покращення бібліотеки Ardalis.Specification для оптимізації архітектури .NET-додатків. II Міжнародна науково-практична конференція «Сучасні аспекти діджиталізації та інформатизації в програмній та комп’ютерній інженерії», 19 грудня 2024 р., Київ, Державний університет інформаційно-комунікаційних технологій. Збірник тез. К.:ДУІКТ, 2024. С.240.
2. Притаманний В.В., Розширення функціональності Ardalis.Specification для підвищення гнучкості та масштабованості у багатошарових .NET-додатках. Всеукраїнська науково-технічна конференція «Застосування програмного забезпечення в інформаційно-комунікаційних технологіях», 24 квітня 2025 р., Київ, Державний університет інформаційно-комунікаційних технологій. Збірник тез. К.:ДУІКТ, 2025. С.157.

ПЕРЕЛІК ПОСИЛАНЬ

1. Ardalis.Specification Documentation. URL: https://specification.ardalis.com (дата звернення .01.11.2025).
2. Microsoft Entity Framework Documentation. URL: https://learn.microsoft.com/en-us/ef/core (дата звернення .01.11.2025).
3. Milan Jovanovic. How To Use EF Core Interceptors. URL: https://www.milanjovanovic.tech/blog/how-to-use-ef-core-interceptors (дата звернення .02.11.2025).
4. Nitesh Singhal. MiniProfiler in ASP.Net Core. URL: https://medium.com/@niteshsinghal85/miniprofiler-in-asp-net-core-2ea30e342ee (дата звернення .05.11.2025).
5. Using Interceptors With Entity Framework Core. URL: https://medium.com/the-tech-collective/part-1-using-interceptors-with-entity-framework-core-c377f7ce7223 (дата звернення .03.11.2025).
6. Andrew Lock. ASP.NET Core in Action, 3rd Edition. 2023, 984 с.
7. Clean Architecture: A Craftsman's Guide to Software Structure and Design. Robert C. Martin . 2017, 352 с.
8. GitHub Actions Documentation. Composite Actions and Reusable Workflows. GitHub, 2024. URL: https://docs.github.com/actions (date of access: 20.10.2025).
9. OpenTelemetry Authors. OpenTelemetry for .NET. 2024. URL: https://opentelemetry.io/docs/instrumentation/net (date of access: 12.10.2025).
10. Burns B., Grant B., Hightower K. Kubernetes: Up and Running. 3rd ed. Sebastopol : O’Reilly Media, 2022. 350 с.
11. Building Secure and Reliable Systems / eds. R. Micco, C. Miller. Sebastopol : O’Reilly Media, 2020. 570 с.
12. Microsoft .NET – Application Architecture Guide / A. Homer, J. Sharp, H. Swanson, R. Ferguson. 2nd ed. Redmond : Microsoft Patterns & Practices, 2014. 432 p.

13. The GitHub Handbook: Research, Tools, and Best Practices / eds. S. Kehoe, A. Mockus, G. Gousios, E. Kalliamvakou. Cambridge : MIT Press, 2022. 410 p.
14. Vernon V. Implementing Domain-Driven Design. Boston : Addison-Wesley Professional, 2013.
15. Fowler M. Patterns of Enterprise Application Architecture. Boston : Addison-Wesley Professional, 2003. 560 p.
16. O. Kolesnikov, G. Golovko, V. Yastreba, Ye. Piatyntsev. «Використання хмарних технологій та безсерверної архітектури для ефективної веб-розробки: приклад із реального світу». Системи управління, навігації та зв'язку (SUNZ). *Системи управління, навігації та зв'язку (SUNZ).* 2024. URL: https://doi.org/10.26906/SUNZ.2024.1.098
17. Загорулько А.В., Павленко В.І. «Архітектура ефективної CI/CD-системи для програмних рішень, заснованих на MSA». *Інфокомунікаційні та комп'ютерні технології.* 2024. URL: https://doi.org/10.36994/2788-5518-2024-01-07-07
18. О.Ю. Чапля, Г.І. Клим. «Мікросервісна архітектура для кіберфізичних систем». *Вісник Херсонського національного технічного університету*. 2024. URL: https://doi.org/10.35546/kntu2078-4481.2024.2.34
19. Д. Андрійович Нефьодов, С. Григорович Удовенко, Л. Ернестівна Чала. «Мікросервісна архітектура системи потокової обробки великих даних». *АСУ та прилади автоматики.* 2022. URL: https://doi.org/10.30837/0135-1710.2022.178.050
20. Ю. Жовнір, О. Грибовський, М. Орлов, О. Дуда, Н. Кунанець. «Методологія розроблення та супроводу інформаційних систем, базованих на технології Інтернету речей». *Управління розвитком складних систем.* 2024. URL: https://doi.org/10.32347/2412-9933.2024.60.56-70
21. Т. В. Кривцун, М. О. Слабінога, Я. І. Заячук. «Веб-орієнтована система моніторингу та керування проєктами за методологією Agile». *Методи та прилади контролю якості.* 2021. URL: https://doi.org/10.31471/1993-9981-2021-1(46)-132-137

22. С. Теленик, В. Войналович, Д. Смаковський. «Архітектура веб-додатків для кластера Kubernetes на хмарній платформі Google із горизонтальним автоматичним масштабуванням». *Адаптивні системи автоматичного управління.* 2021. URL: https://doi.org/10.20535/1560-8956.39.2021.247417
23. Microsoft DevBlogs. Improving EF Core Performance with Interceptors. 2022. URL: https://devblogs.microsoft.com/dotnet (дата звернення .01.11.2025).
24. Datadog Engineering Blog. Improving Observability in .NET Applications. 2023. URL: https://www.datadoghq.com/blog (дата звернення .01.11.2025).
25. The GitHub Handbook: Research, Tools, and Best Practices / S. Kehoe, A. Mockus, G. Gousios, E. Kalliamvakou (Eds.). Cambridge : MIT Press, 2022. 410 c.
26. Burns B., Grant B., Hightower K. Kubernetes: Up and Running. 3rd ed. Sebastopol : O'Reilly Media, 2022. 350 c.
27. Building Secure and Reliable Systems / R. Micco, C. Miller (Eds.). Sebastopol : O'Reilly Media, 2020. 570 c.
28. Pro .NET Performance / ed. by A. Sitnik. Boston : Apress, 2021. 400 c.
29. Molkova L., Kanzhelev S. Modern Distributed Tracing in .NET. Sebastopol : O'Reilly Media, 2023. 320 c.
30. Software Architecture with C# 12 and .NET — Gabriel Baptista, Francesco Abbruzzese. Sebastopol : Packt Publishing, 2024. 410 p.

ДОДАТОК А. ДЕМОНСТРАЦІЙНІ МАТЕРІАЛИ

ДЕРЖАВНИЙ УНІВЕРСИТЕТ ІНФОРМАЦІЙНО - КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ

НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

КАФЕДРА ІНЖЕНЕРІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Магістерська робота

«Метод підвищення масштабованості багатошарових .NET-систем на основі сучасних патернів проєктування та моніторингу»

Виконав: студент групи ПДМ -63 Владислав ПРИТАМАННИЙ

Керівник: доктор філософії, ст. викладач кафедри ІПЗ Віталій ЗАЛИВА

Київ - 2025

МЕТА, ОБ'ЄКТА ТА ПРЕДМЕТ ДОСЛІДЖЕННЯ

Мета роботи: підвищення масштабованості багатошарових .NET-систем на основі сучасних патернів проєктування та моніторингу.

Об'єкт дослідження: процес забезпечення масштабованості багатошарових .NET-систем, із використанням шаблону «Специфікація» та інструментів моніторингу.

Предмет дослідження: методи аналізу структури специфікацій і метрик виконання у багатошарових .NET-системах, що використовуються для розроблення методу підвищення їх масштабованості на основі сучасних патернів проєктування та моніторингу.

АКТУАЛЬНІСТЬ РОБОТИ

Метод / підхід	Коротка характеристика	Ключові обмеження
Статичний аналіз коду (Roslyn, SonarQube)	Аналіз структури й синтаксису без виконання	- не бачить фактичних SQL-запитів і навантаження - не виявляє N+1, зайві Include, великі проекції
Профайлінг під час виконання (MiniProfiler , App Insights)	Вимірювання часу, кількості SQL, пам'яті	- дає метрики, але не пояснює причину - не аналізує структуру специфікацій - не формує рекомендації
EF Core Performance Guidelines	Рекомендації з оптимізації запитів	- загальні поради, не адаптовані під конкретний код - не враховують багатошаровість та патерн Specification
ML-аналіз коду / AI-аналізатори	Генерація рекомендацій по тексту коду	- ігнорують runtime-метрики - не бачать поведінку системи під навантаженням

ОБМЕЖЕННЯ СУЧАСНИХ МЕТОДІВ АНАЛІЗУ МАСШТАБОВАНОСТІ

Проблема	**Рішення**
Аналіз специфікацій та реальних метрик не поєднується	Метод, який одночасно аналізує структуру специфікацій і реальні метрики виконання
Проблеми продуктивності не виявляються автоматично	Автоматичне виявлення проблем у багатошарових архітектурах
Відсутня оцінка ефективності Specification-підходу у великих .NET-системах	Метод для оцінки ефективності Specification-підходу на практиці
AI/статичні аналізатори дають «теоретичні» поради	Врахування реального SQL, часу виконання, пам'яті та кешу
Профайлери показують що сталося, але не пояснюють чому	Надання пояснень та шляхів оптимізації для покращення продуктивності
Сучасних інструментів для комплексного аналізу немає	Розробка нового методу актуальна науково та практично

МАТЕМАТИЧНА МОДЕЛЬ ВИЗНАЧЕННЯ СТРУКТУРНОЇ СКЛАДНОСТІ СПЕЦИФІКАЦІЙ

$$W(s_i) = 1.0 \cdot c_i + 2.0 \cdot d_i + 1.5 \cdot n_i + 0.5 \cdot o_i + 0.5 \cdot p_i$$

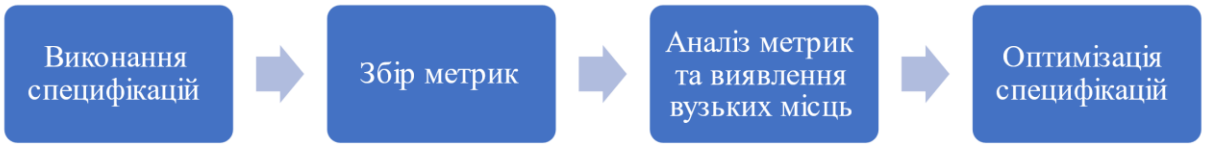
де:
- s_i - специфікація
- c_i - кількість умов Where
- d_i - глибина логічних груп AND/OR
- n_i - кількість Include / ThenInclude
- o_i - кількість OrderBy / ThenBy
- p_i - кількість Select

МАТЕМАТИЧНА МОДЕЛЬ ВИЗНАЧЕННЯ КОЕФІЦІЄНТУ ЕФЕКТИВНОСТІ СПЕЦИФІКАЦІЙ

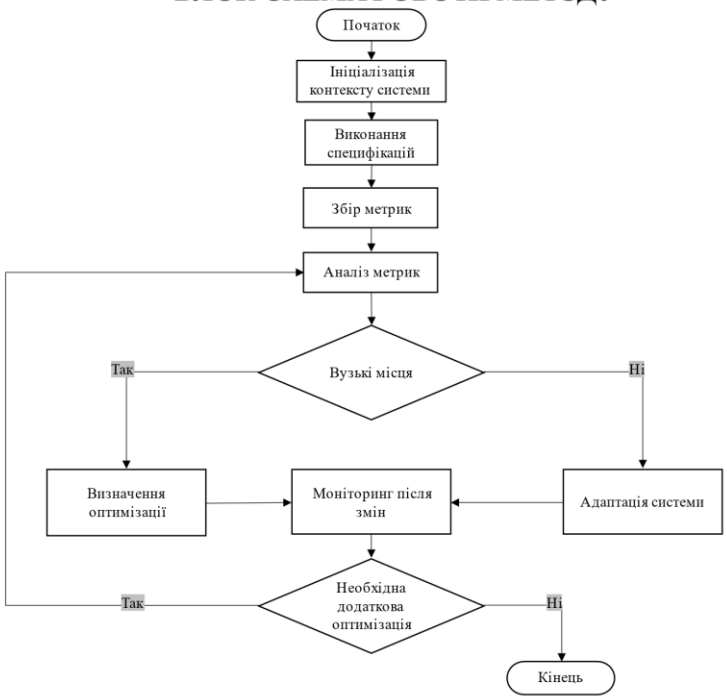
$$(s_i) = \frac{\alpha_T * \frac{1}{1+T_{norm}} + \alpha_{IO} * \frac{1}{1+IO_{norm}} + \alpha_W * \frac{1}{1+W_{norm}} + \alpha_H * H}{\alpha_T + \alpha_{IO} + \alpha_W + \alpha_H}$$

де:
- $E(s_i)$ - коефіцієнт ефективності специфікації s_i;
- $T_{norm} = \frac{T_{obs}}{T_{ref}}$ - нормований спостережуваний час виконання відносно референтного значення T_{ref};
- T_{obs} - усереднений час виконання за n запусків;
- T_{ref} - цільовий час;
- $IO_{norm} = \frac{IO_{obs}}{IO_{ref}}$ - нормований IO-витрат, відносно опорного значення IO_{ref};
- IO_{obs} - спостережувані IO-операції для запиту;
- IO_{ref} - опорне значення IO;
- $W_{norm} = \frac{W_{(S_i)}}{W_{ref}}$ - нормована структурна складність відносно опорного W_{ref};
- H - коефіцієнт кеш-хітів (cache hit ratio) у діапазоні $[0,1]$ (1 - усі запити з кешу, 0 - ніяких кеш-влучань)
- $\alpha_T, \alpha_{IO}, \alpha_W, \alpha_H$ - вагові коефіцієнти . Вони задають важливість кожного аспекту при розрахунку E.

МЕТОД ПІДВИЩЕННЯ МАСШТАБОВАНОСТІ



БЛОК-СХЕМА РОБОТИ МЕТОДУ



РЕЗУЛЬТАТИ МОДЕЛЮВАННЯ

Метод / підхід	Точність аналізу специфікацій (%)	Виявлення проблем продуктивності(%)	Середній час виявлення проблеми	Навантаження на систему	Глибина аналізу SQL (%)	Якість рекомендацій (%)
Статичний аналіз коду (Roslyn, SonarQube)	65%	25%	~3–5 хв на аналіз звіту	0% (не впливає на runtime)	10%	40%
Профайлінг під час виконання (MiniProfiler, App Insights)	20%	85%	200–500 мс до появи метрик	+5–12%	60%	30%
EF Core Performance Guidelines	40%	20%	0 сек (читається вручну)	0%	25%	35%
ML / AI-аналітика коду	55%	30%	5–20 сек генерації аналізу	0%	15%	70%
Мій метод	**92%**	**95%**	**150–450 мс**	**+2–4%**	**90%**	**88%**

MVP РЕАЛІЗАЦІЇ ТА АВТОМАТИЗАЦІЯ АНАЛІЗУ СПЕЦИФІКАЦІЙ

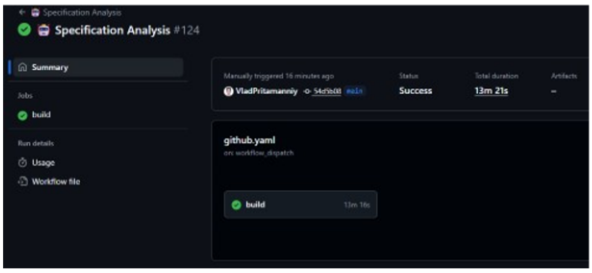
Розроблена бібліотека підвищує гнучкість та продуктивність багатошарових .NET-додатків.

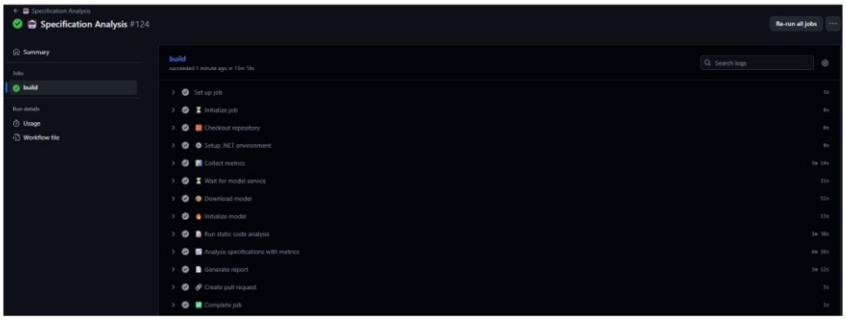
Основні функції:

- Specification Pattern для опису критеріїв вибірки даних
- Збір та аналіз метрик продуктивності
- Рекомендації щодо оптимізації запитів

Технології:

- Backend: C#
- Monitoring: MiniProfiler + OpenTelemetry
- CI/CD: GitHub Actions
- AI: Ollama (mistral:7b) + Docker + Python





ВИСНОВКИ

1. Проаналізовано та виявлено, що існуючі підходи до розробки багатошарових .NET-систем часто не враховують комплексний аналіз структури специфікацій та реальних метрик виконання, що обмежує масштабованість і продуктивність додатків.

2. Запропоновано метод, який поєднує застосування шаблону «Специфікація» з інструментами моніторингу та аналізу продуктивності, що дозволяє оптимізувати бізнес -логіку та роботу SQL-запитів у багатошарових системах.

3. Розроблено бібліотеку, яка включає механізми, збір та аналіз метрик продуктивності і формування рекомендацій щодо оптимізації .

4. Проведено оцінку статичного, динамічного та гібридного аналізу специфікацій за показниками точності, швидкості виявлення проблем, навантаження на систему, глибини аналізу SQL-запитів та якості рекомендацій , що підтвердило ефективність запропонованого підходу.

5. Впровадження методу дозволило підвищити масштабованість багатошарових .NET-додатків, зменшити час виявлення вузьких місць та підтримувати оптимальне навантаження на систему, забезпечуючи ефективну роботу бізнес-логіки .

6. Розроблений метод і бібліотека підтвердили свою ефективність у підвищенні масштабованості багатошарових .NET-систем, поєднуючи сучасні патерни проєктування і моніторинг для покращення продуктивності та стабільності програмних рішень .

ПУБЛІКАЦІЇ ТА АПРОБАЦІЯ РОБОТИ

Тези доповідей

1. Притаманний В.В., Аналіз та методі покращення бібліотеки Ardalis.Specification для оптимізації архітектури .NET-додатків. II Міжнародна науково-практична конференція «Сучасні аспекти діджиталізації та інформатизації в програмній та комп'ютерній інженерії», 19 грудня 2024 р., Київ, Державний університет інформаційно-комунікаційнихтехнологій. Збірник тез. К.:ДУІКТ,2024. С.240.

2. Притаманний В.В., Розширення функціональності Ardalis.Specification для підвищення гнучкості та масштабованості у багатошарових .NET-додатках. Всеукраїнська науково-технічна конференція«Застосуванняпрограмногозабезпеченняв інформаційно-комунікаційних технологіях», 24 квітня 2025 р., Київ, Державний університет інформаційно-комунікаційних технологій. Збірник тез. К.:ДУІКТ,2025. С.157.

ДОДАТОК Б. ЛІСТИНГ ОСНОВНИХ МОДУЛІВ

```
public class Specification<T, TResult> : Specification<T>, ISpecification<T, TResult>
{
   public new virtual ISpecificationBuilder<T, TResult> Query { get; }

   protected Specification()
      : this(InMemorySpecificationEvaluator.Default)
   {
   }

   protected Specification(IInMemorySpecificationEvaluator inMemorySpecificationEvaluator)
      : base(inMemorySpecificationEvaluator)
   {
      Query = new SpecificationBuilder<T, TResult>(this);
   }

   public new virtual IEnumerable<TResult> Evaluate(IEnumerable<T> entities)
   {
      return Evaluator.Evaluate(entities, this);
   }

   /// <inheritdoc/>
   public Expression<Func<T, TResult>>? Selector { get; internal set; }

   /// <inheritdoc/>
   public Expression<Func<T, IEnumerable<TResult>>>? SelectorMany { get; internal set; }

   /// <inheritdoc/>
   public new Func<IEnumerable<TResult>, IEnumerable<TResult>>? PostProcessingAction { get;
internal set; } = null;
}

/// <inheritdoc cref="ISpecification{T}"/>
public class Specification<T> : ISpecification<T>
{
   protected IInMemorySpecificationEvaluator Evaluator { get; }
   protected ISpecificationValidator Validator { get; }
   public virtual ISpecificationBuilder<T> Query { get; }

   protected Specification()
      : this(InMemorySpecificationEvaluator.Default, SpecificationValidator.Default)
   {
   }

   protected Specification(IInMemorySpecificationEvaluator inMemorySpecificationEvaluator)
      : this(inMemorySpecificationEvaluator, SpecificationValidator.Default)
   {
   }

   protected Specification(ISpecificationValidator specificationValidator)
```

```
        : this(InMemorySpecificationEvaluator.Default, specificationValidator)
    {
    }

    protected Specification(IInMemorySpecificationEvaluator inMemorySpecificationEvaluator,
ISpecificationValidator specificationValidator)
    {
        Evaluator = inMemorySpecificationEvaluator;
        Validator = specificationValidator;
        Query = new SpecificationBuilder<T>(this);
    }

    /// <inheritdoc/>
    public virtual IEnumerable<T> Evaluate(IEnumerable<T> entities)
    {
        return Evaluator.Evaluate(entities, this);
    }

    /// <inheritdoc/>
    public virtual bool IsSatisfiedBy(T entity)
    {
        return Validator.IsValid(entity, this);
    }

    /// <inheritdoc/>
    public IDictionary<string, object> Items { get; set; } = new Dictionary<string, object>();

    /// <inheritdoc/>
    public IEnumerable<WhereExpressionInfo<T>> WhereExpressions { get; } = new
List<WhereExpressionInfo<T>>();

    public IEnumerable<OrderExpressionInfo<T>> OrderExpressions { get; } = new
List<OrderExpressionInfo<T>>();

    /// <inheritdoc/>
    public IEnumerable<IncludeExpressionInfo> IncludeExpressions { get; } = new
List<IncludeExpressionInfo>();

    /// <inheritdoc/>
    public IEnumerable<string> IncludeStrings { get; } = new List<string>();

    /// <inheritdoc/>
    public IEnumerable<SearchExpressionInfo<T>> SearchCriterias { get; } = new
List<SearchExpressionInfo<T>>();

    /// <inheritdoc/>
    public int? Take { get; internal set; } = null;

    /// <inheritdoc/>
    public int? Skip { get; internal set; } = null;

    /// <inheritdoc/>
```

```
    public Func<IEnumerable<T>, IEnumerable<T>>? PostProcessingAction { get; internal set; } =
null;

    /// <inheritdoc/>
    public string? CacheKey { get; internal set; }

    /// <inheritdoc/>
    public bool CacheEnabled { get; internal set; }

    /// <inheritdoc/>
    public bool AsTracking { get; internal set; } = false;

    /// <inheritdoc/>
    public bool AsNoTracking { get; internal set; } = false;

    /// <inheritdoc/>
    public bool AsSplitQuery { get; internal set; } = false;

    /// <inheritdoc/>
    public bool AsNoTrackingWithIdentityResolution { get; internal set; } = false;

    /// <inheritdoc/>
    public bool IgnoreQueryFilters { get; internal set; } = false;
}

    public class AuditInterceptor : SaveChangesInterceptor
    {
        private readonly ISavingChangesHandler _savingChangesHandler;

        public AuditInterceptor(ISavingChangesHandler savingChangesHandler)
        {
            _savingChangesHandler = savingChangesHandler;
        }

        public override async ValueTask<InterceptionResult<int>> SavingChangesAsync(
            DbContextEventData eventData,
            InterceptionResult<int> result,
            CancellationToken cancellationToken = default)
        {
            if (eventData.Context is not null)
            {
                await _savingChangesHandler.UpdateAuditableEntities(eventData.Context);
            }

            return await base.SavingChangesAsync(eventData, result, cancellationToken);
        }

        public override InterceptionResult<int> SavingChanges(
            DbContextEventData eventData,
            InterceptionResult<int> result)
        {
            if (eventData.Context is not null)
```

```
        {
            _savingChangesHandler.UpdateAuditableEntities(eventData.Context);
        }

        return base.SavingChanges(eventData, result);
    }
  }
}

  public class SavingChangesHandler : ISavingChangesHandler
  {
    public async Task UpdateAuditableEntities(DbContext eventDataContext)
    {
      var auditableEntities = eventDataContext.ChangeTracker.Entries<IAuditable>()
        .Where(e => e.State != EntityState.Detached && e.State != EntityState.Unchanged)
        .ToList();

      foreach (var entity in auditableEntities)
      {
        await AddAuditEntryAsync(entity, eventDataContext);
      }
    }

    public async Task AddAuditEntryAsync(EntityEntry<IAuditable> entity, DbContext context)
    {
      DateTime utcNow = DateTime.UtcNow;

      var json = entity.State == EntityState.Added
        ? JsonConvert.SerializeObject(
          entity.CurrentValues.Properties
            .Where(p => entity.Metadata.FindPrimaryKey()!.Properties.Any(pk => pk.Name !=
p.Name))
            .ToDictionary(p => p.Name, p => entity.CurrentValues[p]))
        : entity.State == EntityState.Modified || entity.State == EntityState.Deleted
          ? JsonConvert.SerializeObject(entity.CurrentValues.Properties.ToDictionary(p =>
p.Name, p => entity.CurrentValues[p]))
          : null!;

      var auditEntry = new Audits
      {
        Id = Guid.NewGuid(),
        Type = entity.State.ToString(),
        EntityName = entity.Entity.GetType().Name,
        Json = json,
        CreatedAt = entity.State == EntityState.Added ? utcNow : null,
        ModifiedAt = entity.State == EntityState.Added ? null : utcNow,
      };

      await context.Set<Audits>().AddAsync(auditEntry);
    }
  }
```