**НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ**
**«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ**
**імені ІГОРЯ СІКОРСЬКОГО»**

**Факультет інформатики та обчислювальної техніки**

**Кафедра автоматики та управління в технічних системах**

«На правах рукопису»
УДК _004.62, 004.42_

До захисту допущено:
Завідувач кафедри
_____ Олександр РОЛІК
«___»_____2021р.

# Магістерська дисертація

**на здобуття ступеня магістра**

**за освітньо-науковою програмою «Інженерія програмного забезпечення комп'ютерних систем»**

**зі спеціальності 121 «Інженерія програмного забезпечення»**

**на тему: «Модель уніфікованої бібліотеки для представлення та обробки складнозв'язаних даних в об'єктно-орієнтованій парадигмі»**

Виконала:
студентка VI курсу, групи ІТ-91мн
Літвінова Наталія Олександрівна _____

Керівник:
доцент кафедри АУТС, к.т.н., доцент
Катін Павло Юрійович _____

Рецензент:
доцент кафедри ОТ, к.т.н., доцент
Волокита Артем Миколайович _____

Засвідчую, що у цій магістерській дисертації немає запозичень з праць інших авторів без відповідних посилань.
Студентка _____

Київ – 2021 року

**Національний технічний університет України**

**«Київський політехнічний інститут імені Ігоря Сікорського»**

**Факультет інформатики та обчислювальної техніки**

**Кафедра автоматики та управління в технічних системах**

Рівень вищої освіти – другий (магістерський)

Спеціальність – 121 «Інженерія програмного забезпечення»

Освітньо-наукова програма «Інженерія програмного забезпечення комп'ютерних систем»

ЗАТВЕРДЖУЮ

Завідувач кафедри

_____ Олександр РОЛІК

«___»_____2021 р.

**ЗАВДАННЯ**

**на магістерську дисертацію студенту**

**Літвінова Наталія Олександрівна**

1. Тема дисертації «Модель уніфікованої бібліотеки для представлення та обробки складнозв'язаних даних в об'єктно-орієнтованій парадигмі», науковий керівник дисертації Катін Павло Юрійович, доцент кафедри АУТС, к.т.н., затверджені наказом по університету від «12» березня 2021 р. № 809-с

2. Термін подання студентом дисертації 11.05.2021

3. Об'єкт дослідження складнозв'язані дані, зокрема ті, що можуть бути представлені у реляційній моделі

4. Предмет дослідження модель бібліотеки для представлення та обробки в об'єктно-орієнтованій парадигмі складнозв'язаних структурованих даних, що зберігаються в різних джерелах

5. Перелік завдань, які потрібно розробити: проаналізувати теоретичні матеріали, визначити вимоги та задачі до бібліотеки, розробити модель уніфікованої бібліотеки, спроектувати та реалізувати тестову базу даних та бібліотеку згідно розробленої моделі, перевірити за допомогою тестів роботу програми

6. Орієнтовний перелік графічного (ілюстративного) матеріалу: діаграма варіантів використання бібліотеки, діаграма компонентів базової моделі, діаграма компонентів тестової бібліотеки, діаграма розгортання, діаграми класів, діаграми

послідовності серіалізації та десеріалізації, схема формату даних, ER діаграма тестової бази даних

7. Орієнтовний перелік публікацій Способи захисту баз даних від SQL-ін'єкцій, Формат обміну даними для сутностей у реляційному представленні

9. Дата видачі завдання 01.02.2021

Календарний план

| № з/п | Назва етапів виконання магістерської дисертації | Термін виконання етапів магістерської дисертації | Примітка |
|---|---|---|---|
| 1 | Вибір та узгодження теми магістерської дисертації | 02.02.2021-04.02.2021 | |
| 2 | Аналіз теоретичних матеріалів та вивчення предметної області | 05.02.2021-18.02.2021 | |
| 3 | Розробка технічного завдання | 15.03.2021-18.03.2021 | |
| 4 | Проектування моделі уніфікованого рішення | 19.03.2021-25.03.2021 | |
| 5 | Вибір технологій реалізації тестової бібліотеки та бази даних | 26.03.2021-28.03.2021 | |
| 6 | Розробка структури тестової бази даних та проектування бібліотеки | 01.04.2021-04.04.2021 | |
| 7 | Реалізація, перевірка та налагодження програми | 05.04.2021-23.04.2021 | |
| 8 | Оформлення текстових та графічних матеріалів | 15.04.2021-04.05.2021 | |
| 9 | Передзахист магістерської дисертації | 05.05.2021 | |
| 10 | Доопрацювання пояснювальної записки та підготовка презентації | 06.05.2021-11.05.2021 | |
| 11 | Захист магістерської дисертації | 18.05.2021 | |

Студент Наталія ЛІТВІНОВА

Науковий керівник Павло КАТІН

# РЕФЕРАТ

Магістерська дисертація присвячена розробці та опису моделі уніфікованої бібліотеки для представлення та обробки складнозв'язаних структурованих даних у об'єктно-орієнтованій парадигмі.

Магістерська дисертація міститься на 107 сторінках тексту та включає 23 рисунки, 21 таблицю та 24 бібліографічні посилання. Вона складається з наступних розділів: перелік умовних позначень, скорочень і термінів, вступ, 6 розділів для основної частини, висновки, перелік посилань та 9 додатків.

Актуальність обраної теми пояснюється зростанням складності програмного забезпечення. Виникає необхідність підвищення ефективності його роботи та спрощення процесу розробки. Це можна вирішити за допомогою уніфікованого засобу для представлення, обробки та перетворення даних з різних джерел.

Метою роботи є спрощення та уніфікація представлення та обробки складнозв'язаних даних у об'єктно-орієнтованій парадигмі у програмних системах, а також зменшення обсягу даних під час обміну.

Об'єктом дослідження є складнозв'язані дані, зокрема ті, що можуть бути представлені у реляційній моделі.

Предметом дослідження є модель бібліотеки для представлення та обробки в об'єктно-орієнтованій парадигмі складнозв'язаних структурованих даних, що зберігаються в різних джерелах.

Під час виконання магістерської дисертації були застосовані такі теоретичні методи дослідження, як: індукція, аналіз та синтез, абстрагування та узагальнення.

Результати роботи можуть бути використані під час розробки програмних систем, що передбачають взаємодію з джерелами структурованих даних, незалежно від більшості факторів: архітектури, цільового середовища, рівня складності тощо.

Ключові слова: реляційна модель даних, ООП представлення даних, програмний обмін даними, формат серіалізації, взаємодія з джерелами даних, реляційні бази даних.

# SUMMARY

The master's thesis is devoted to the development and description of a unified software library model for the representation and processing of complex structured data in an object-oriented paradigm.

The master's thesis contains 107 text pages and includes 23 figures, 21 tables and 24 bibliographic references. It is made with the following sections: a list of symbols, abbreviations and terms, introduction, 6 sections for the main part, conclusions, references and 9 applications.

The relevance of the topic chosen is referable to the growing complexity of the software. There is a need to increase its efficiency and to simplify the development process. This can be solved with a unified tool for presenting, processing and converting data from different sources.

The purpose of the work is to simplify and unify the presentation and processing of complex data in an object-oriented paradigm in software systems, as well as to reduce the amount of data during the exchange.

The object of research is complex data, in particular those which can be represented in a relational model.

The subject of the study is the model of library for the representation and processing in an object-oriented paradigm of complex structured data stored in different sources.

Following theoretical research methods were used while realization of the dissertation: induction, analysis and synthesis, abstraction and generalization.

The results of the work can be used during the development of software systems that provide interaction with sources of structured data, regardless of most factors, such as architecture, target environment, level of complexity, etc.

Keywords: relational data model, OOP data representation, software data exchange, serialization format, interaction with data sources, relational databases.

# ЗМІСТ

# ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СКОРОЧЕНЬ І ТЕРМІНІВ

БД — база даних

ОО — об’єктно-орієнтований

ОС — операційна система

ПЗ — програмне забезпечення

СКБД — система керування базами даних

ADO — ActiveX Data Objects

API — Application Programming Interface

DLL — Dynamic Link Library

IDE — Integrated development environment

ODBC — Open Database Connectivity

SQL — Structured Query Language

UML — Unified Modeling Language

# ВСТУП

Сфера розробки програмного забезпечення набула широкого поширення в сучасному світі серед усіх напрямків бізнесу, особливо тих, що мають справу з великим потоком даних. На даний момент провідні компанії замовляють інформаційні системи для клієнтів, такі як CRM, програмне забезпечення для управління бізнес-процесами (ERP) та програми на основі робочих процесів. Яким би не було призначення застосунку програмне забезпечення передбачає роботу з великим обсягом даних.

Питання організації інформаційних процесів та передачі даних у наш час є важливим для настільних систем, розподілених систем, веб-застосунків та застосунків з мікросервісною архітектурою, що складають більшу частину ринку програмного забезпечення.

У роботі досліджуються два базових питання: пошук універсального рішення для представлення в об’єктно-орієнтованій парадигмі структурованих даних (в тому числі сутностей у реляційній моделі з різних джерел); організація роботи з даними та створення формату обміну даними для підвищення ефективності їх передачі.

На даний момент існує багато фреймворків та бібліотек, що надають можливість зв’язати програмні об’єкти з реляційними базами даних (ORM), проте вони не надають інтерфейс для взаємодії з джерелами іншого типу.

Метою роботи є спрощення та уніфікація представлення та обробки складнозв’язаних даних у об’єктно-орієнтованій парадигмі у програмних системах, а також зменшення обсягу даних під час обміну.

Об’єктом дослідження є складнозв’язані дані, зокрема ті, що можуть бути представлені у реляційній моделі.

У ролі предмета дослідження магістерської дисертації модель бібліотеки для представлення та обробки в об’єктно-орієнтованій парадигмі складнозв’язаних структурованих даних, що зберігаються в різних джерелах.

Під час виконання магістерської дисертації були застосовані переважно теоретичні методи дослідження: аналіз та синтез – для вивчення основних

складових логічних сутностей, їх взаємозв’язків та ролей у представленні окремого елемента у системі; індукція – для визначення загальних тенденцій (характеристик, відношень) складнозв’язаних структурованих даних, а також для розробки спільної моделі їх представлення й обробки у програмі; абстрагування та узагальнення – для відокремлення та опису загальних спільних ознак, визначених у предметі дослідження.

У рамках магістерської дисертації за допомогою уніфікованої мови моделювання та інших ілюстративних матеріалів надано опис програмної бібліотеки для роботи зі складнозв’язаними даними, що можуть зберігатися у різних джерелах. Під уніфікованістю мається на увазі єдиність структури та правил побудови бібліотеки незалежно від стеку технологій та мови програмування.

Універсальність рішення досягається завдяки гнучкій архітектурі, що базується на загально визнаних шаблонах проектування. Також для представлення працездатності запропонованого рішення була розроблена, відлагоджена та протестована тестова версія бібліотеки, написана за допомогою .NET Core та мови програмування C#. В якості тестового джерела даних для неї була спроєктована типова база даних, що знаходиться на віддаленому сервері PostgreSQL.

У загальному випадку представлене рішення може застосовуватися під час розробки програмної системи будь-якого рівня складності, оскільки архітектура дозволяє підтримку множини джерел даних та забезпечує зручний механізм опису складнозв’язаних об’єктів та доступ до їх метаданих.

Також у межах дисертації представляється розроблений формат обміну даними для сутностей з реляційним представленням, основною перевагою якого є мінімізація обсягу даних, що передаються.

Таким чином, наукову новизну даної роботи забезпечують два чинники: вперше одержано модель, за допомогою якої можна розробити уніфіковану бібліотеку для роботи зі зв’язаними реляційними даними, а також удосконалено процес обміну такими даними між програмними компонентами за допомогою спеціального нового формату серіалізації.

Основними задачами, що вирішуються в рамках цієї роботи, є наступні:

— об’єктно-орієнтоване представлення для складнозв’язаних сутностей з різних джерел даних;
— реалізація взаємодії з різноманітними джерелами даними за спільною схемою;
— здійснення обробки даних відповідних джерелу сутностей;
— покращення обміну даними між процесами шляхом зменшення обсягу даних, що передаються;
— підвищення ефективності обробки прийнятих даних.

Результати проведених у рамках магістерської дисертації досліджень було оприлюднено на VII Міжнародній науково-практичній конференції у збірнику «WORLD SCIENCE: PROBLEMS, PROSPECTS AND INNOVATIONS» у доповіді «Способи захисту баз даних від sql ін’єкцій для забезпечення якості програмного забезпечення» у 2021 році в м. Торонто, Канада [1].

Публікація результатів магістерської роботи здійснювалася у фаховому журналі України «Технічні науки та технології» №1(23), 2021, м. Чернігів, за темою «Підвищення ефективності обміну даними сутностей у реляційному представленні та їх обробки» [2]. Також було розглянуто доцільність застосування розробленого формату обміну даними у співаторстві з Альпертом М. І. у роботі «Програмно-апаратна інфраструктура наземної автономної платформи з елементами штучного інтелекту» у науковому журналі «Математичні машини і системи» №1 у 2021 році, м. Київ [3].

Загальна кількість опублікованих статей: 3.

# 1 ПОСТАНОВКА ЗАДАЧІ

З метою розробки моделі уніфікованої бібліотеки для представлення та обробки складнозв’язаних даних в об’єктно-орієнтованій парадигмі необхідно визначити коло завдань, що буде вирішувати сама бібліотека, та чітко визначити область її застосування. У свою чергу, під час постановки задачі важливо визначити вимоги до програмного забезпечення — необхідна передумова для розробки, оскільки це є одним з ключів для вирішення питань стосовно архітектури системи, що приймається на перших етапах проектування на основі технічного завдання.

## 1.1 Проблема уніфікованого представлення та обробки складнозв’язаних даних

Перш ніж розробити програмне забезпечення для будь-якого сегменту бізнеса незалежно від його сфери, спочатку проводиться дослідження відповідної йому предметної області. Під час описання будь-якої предметної області виділяються її головні сутності, що зв’язані між собою та зазвичай мають певну структуру. Ці сутності далі стають основоположними для моделювання інформаційних процесів у програмній системі. Тому дуже важливо забезпечити універсальний та ефективний механізм для програмної роботи з подібними даними.

Під універсальним мається на увазі те, що рішення має дозволяти, використовуючи єдиний інтерфейс, взаємодіяти з будь-яким сховищем даним незалежно від його розміщення, структури, реалізації тощо. Також для розробленої моделі бібліотеки справедливе інше трактування універсальності: програмне рішення на її основі може бути реалізоване незалежно від платформи та технологій програмування з мінімальними зміни.

В якості джерел даних для розробленої моделі можуть виступати розташовані локально або на віддаленому сервері баз даних, REST API-сервіси, звичайні файли тощо. Робота з ними абстрагується за допомогою спільного інтерфейсу програмної взаємодії.

Зокрема, для об’єктів, що відповідають сутностям, та списків таких об’єктів універсальним є метод їх завантаження з різних джерел, що абстрагується від реалізації будь-якого джерела, використовуючи згаданий вище інтерфейс. А завдяки розробленій системі абстракцій процеси серіалізації/десеріалізації об’єктів є повністю незалежними від формату, що застосовується. Також під час виконання програми для об’єктів є можливість змінювати джерело, до якого вони відносяться.

Для вирішення проблеми обробки самих сутностей, що представляють дані, та їх зв’язків в рамках представленої моделі пропонується використання так званих метаданих сутностей, тобто даних, які надають додаткову інформацію про самі дані. Інакше кажучи, метадані містять службову інформацію, що може містити правила організації зв’язків сутності або необхідні параметри, які вказують на прив’язку чи пріоритетність даних, обмеження для їх значень тощо.

Також, враховуючи незмінну тенденцію використання реляційних БД для збереження даних [4], в рамках розробленого на основі запропонованої моделі тестового програмного рішення було надано інтерфейс для взаємодії з SQL-джерелами та механізм для створення комплексних запитів на вибір даних, текст команди яких формуються автоматично приховано від користувача.

Варто згадати переваги динамічного завантаження бібліотек до програми, що застосовується у розробленому рішенні. За його допомогою можна під час роботи застосунку розширювати набір об’єктів для нових сутностей, довантажувати бібліотеки з реалізаціями постачальника даних для інших джерел тощо.

## 1.2 Питання неефективності обміну структурованими зв'язаними даними з використанням існуючих форматів серіалізації

Для вирішення багатьох задач сучасні системи проектуються не монолітами, а комплексом з декількох різних незалежних частин, які прийнято називати компонентами, або сервісами у мікросервісній архітектурі. У наш час навіть простим застосункам, як правило, доводиться звертатися до сторонніх сервісів для отримання даних через API. Дуже часто взаємодія між компонентами однієї системи

або навіть різними взаємопов’язаними системами відбувається за допомогою мережі, наприклад через HTTP, де дані передаються між сокетами [5]. Такий спосіб взаємодії використовує переважна більшість веб-застосунків.

Як відомо, при передачі даних між компонентами однією з головних проблем є великий обсяг даних, що передаються, особливо, якщо мова йде передачу об’єктів з баз даних.

Якщо вести мову про розповсюджені на даний час формати для передачі даних, то варто зауважити, що вони створювалися як універсальний засіб передачі даних різної структури між комп’ютерами. Отже, вони є неефективними для передачі великої кількості об’єктів, що мають спільну структуру.

У цій роботі буде представлено власний варіант вирішення питання підвищення ефективності передачі структурованих даних за допомогою нового двійкового формату серіалізації/десеріалізації даних, що дозволяє зменшити надлишковість даних, що передаються через мережу.

## 1.3 Функціональні вимоги до бібліотеки

Функціональні вимоги описують очікувану поведінку системи та її функціонал. Наприклад, організацію управління даними та їх обробку, а також дії, що може виконати користувач та інші специфічні функції, які має виконувати програмний продукт [6].

Оскільки дана бібліотека призначена для представлення та обробки складнозв’язаних даних, то основна частина функціональних вимог регламентує саме функціонал, необхідний розробникам для роботи з даними.

Таким чином, з точки зору інженера-програміста як користувача бібліотеки, до основних функціональних вимог можна віднести:

— представлення даних із джерел в об’єктно-орієнтованій парадигмі для взаємодії всередині програми, що передбачає визначення схеми даних об’єкта у відношенні до відповідної сутності у джерелі та відношень цього об’єкта з іншими об’єктами-сутностями;

— можливість додання нових даних до джерела, використовуючи програмні об’єкти, що представляють відповідні сутності;
— внесення змін до існуючих даних за допомогою об’єктів;
— видалення даних у джерелі, для якого розроблені відповідні класи;
— обмін даними між програмними компонентами;
— наявність можливості перегляду метаданих об’єкта, включаючи його зв’язки, під час виконання застосунку, тобто runtime.

Також з врахуванням того, що найпопулярнішим зовнішнім джерелом залишаються реляційні бази даних [4], було вирішено додати такі функціональні вимоги для роботи з SQL-джерелами:

— автоматизована побудова запитів на вибір даних об’єктів з БД;
— можливість виконання на сервері БД довільних команд;
— програмний інтерфейс для перегляду результатів виконання команд;
— забезпечення транзакційності при роботі з БД як джерелом даних.

## 1.4 Нефункціональні вимоги до бібліотеки

На відміну від функціональних вимог, які визначають що програмне забезпечення має робити, нефункціональні вимоги визначають яким воно повинно бути. Вони поділяються на три категорії [6]: вимоги до інтерфейсу (мається на увазі зовнішній вигляд системи для користувача); операційні вимоги; апаратні й програмні вимоги. Нижче будуть перелічені вимоги, що відносяться до останніх двох категорій, оскільки вимоги до інтерфейсу незастосовані у відношенні до бібліотеки.

Так, власне для самого програмного рішення головними нефункціональними вимогами є незалежність від платформ, а також наступні операційні вимоги:

— безпека та конфіденційність: розроблене програмне рішення має передбачати можливість обмеження доступу до будь-якої сторонньої інформації, а також забезпечувати захист джерела даних від атак, що можуть здійснюватися через саме програмне рішення;

— коректність: відповіді на будь-які запити від користувача повинні бути вірні та зрозумілі;
— безвідмовність: має гарантуватися стабільність роботи та захищеність від помилок;
— збереження даних: бібліотека повинна чітко інформувати про результати виконання збереження даних. Для всіх пов’язаних даних має гарантуватися цілісне збереження;
— правила перевірки: не повинно допускатися введення даних, що можуть спричинити некоректну або непередбачувану роботу чи порушити будь-яку із вище зазначених вимог.

Далі надається опис вимог, що додатково висуває розроблена тестова версія бібліотеки.

Апаратні та програмні вимоги:

— мінімальна версія платформи .NET Core – 3.1;
— рекомендований мінімум: процесор з частотою 1 ГГц;
— мінімальний обсяг ОЗП – 1 ГБ;
— PostgreSQL-сервер та користувач з правами на виконання функцій, вибірку, вставлення, зміну та видалення даних.

## 1.5 Висновки

У даному розділі магістерської дисертації було проаналізовано проблему уніфікованого представлення у об’єктно-орієнтованій парадигмі структурованих складнозв’язаних даних. А також розглянуто питання їх передачі та обробки при використанні існуючих форматів даних.

Було визначено основну задачу роботи – створення моделі уніфікованої бібліотеки для вирішення цієї проблеми. Ця модель описує ідею програмної бібліотеки, що може бути реалізована для різних ОС, платформ, сховищ даних, з використанням різних технологій (ОО мов програмування).

Також тут були детально описані функціональні та нефункціональні вимоги до бібліотеки.

Так, зваживши на основні потреби розробників та розповсюджені недоліки існуючих рішень, було зроблено висновок, що уніфікована бібліотека має задовольняти наступним критеріям:

— вміння роботи зі структурованими складнозв’язаними даними з різних джерел;
— надання базової функціональності для роботи з SQL-джерелами, а також механізм автоматизованої побудови запитів на вибір даних сутностей;
— забезпечення ефективної передачі даних між програмними компонентами шляхом зменшення обсягу даних та забезпечення їх багатопотокової обробки;
— коректність та безвідмовність роботи програмного продукту.

# 2 МОДЕЛЬ УНІФІКОВАНОЇ БІБЛІОТЕКИ ДЛЯ ПРЕДСТАВЛЕННЯ ТА ОБРОБКИ СКЛАДНОЗВ'ЯЗАНИХ СТРУКТУРОВАНИХ ДАНИХ

## 2.1 Узагальнений опис варіантів використання уніфікованої бібліотеки

У попередньому розділі були описані задачі, що вирішує бібліотека, модель якої розроблялася в рамках даної магістерської дисертації. Проте для розуміння її призначення, необхідно надати пояснення її місця в архітектурі цілісної промислової програмної системи та основні цілі використання бібліотеки в застосунку.

Якщо розглядати типову для програмних клієнт-серверних систем трирівневу архітектуру [7], зображену на рис. 2.1, то дана бібліотека застосовується на другому рівні для налагодження взаємодії між другим та третім рівнями.

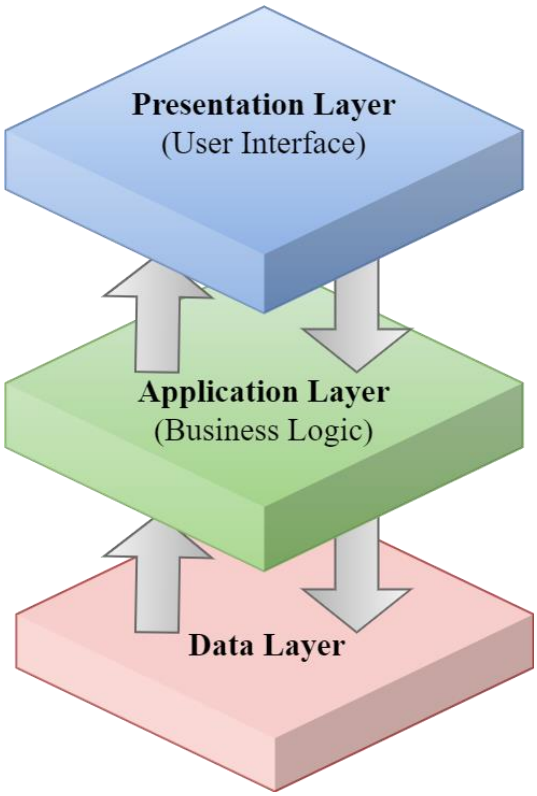


Рисунок 2.1 — Модель трирівневої архітектури програмних систем

Трирівнева архітектура — це усталена архітектура програмних застосунків, що організовує програми у три логічні та фізичні обчислювальні рівні.

До рівнів цієї архітектури входять:

— рівень презентації або користувальницький інтерфейс;
— рівень застосунку чи бізнес-логіки, де відбувається обробка даних;
— рівень даних, де зберігаються дані, пов’язані з предметною областю та роботою цільової системи, та відбувається керування ними.

Така архітектура забезпечує множину переваг у сфері програмування, а саме: підвищення показників випуску продукції, зменшення часу її розробки, покращення якості програмних продуктів. Головна перевага трирівневої архітектури полягає у тому, що кожен рівень може розроблятися одночасно окремими командами розробників, виконуватися на власній інфраструктурі і бути оновлений чи масштабований за необхідності, не впливаючи на інші рівні.

На сьогодні більшість трирівневих застосунків слідують останнім тенденціям модернізації з використанням власних хмарних технологій. До них відносяться контейнери, мікросервіси, а також перехід до хмари. Не зважаючи на плив течії тенденцій, актуальним залишається питання організації взаємодії між сусідніми рівнями. Як вже було згадано, у даній роботі розглядається вирішення питання взаємодії нижніх двох рівнів.

У трирівневій програмі вся комунікація проходить через рівень застосунку. Рівень презентації та рівень даних не можуть безпосередньо взаємодіяти між собою. На цьому рівні інформація, зібрана на рівні презентації, обробляється за допомогою бізнес-логіки, певного набору бізнес-правил. На програмному рівні можна додавати, видаляти або змінювати дані, що зберігаються на 3-ьому рівні. Рівень застосунку, як правило, взаємодіє з рівнем даних за допомогою викликів API.

Рівень даних, що також іноді називають рівнем бази даних чи рівнем доступу до даних, як правило, представляє реляційна система керування базами даних, така як PostgreSQL, MySQL, Oracle або Microsoft SQL Server. Але не обов’язково реляційна: це можуть бути  бази даних NoSQL, такі як Cassandra, CouchDB та MongoDB. Також на цьому рівні в якості джерел даних можуть застосовуватися прості файли (Excel, CSV, XML, JSON, TXT тощо), веб-сервіси (наприклад, AWS Redshift, Azure SQL DW) та визначені користувачем джерела, як REST API.

Зважаючи на таке різноманіття джерел даних для інформаційних систем на рівні застосунку виникає необхідність забезпечити взаємодію з різними джерелами даних. З метою уникнення розробки окремого програмного рішення під кожне джерело та забезпечення одного інтерфейсу для взаємодії з будь-яким джерелом даних з програми була розроблена модель уніфікованої бібліотеки для представлення та обробки даних в об’єктно-орієнтованій парадигмі.

В рамках реалізації своєї основної мети бібліотека надає множину варіантів використання, які зображено у додатку А.

## 2.2 Забезпечення універсальності роботи з джерелами даних

Розроблена модель бібліотеки гарантує універсальний механізм представлення структурованих даних з різноманітних джерел та єдиний інтерфейс взаємодії з джерелами. Відповідно до постановки задачі в розробленій моделі також присутня підтримка «схеми» сутності, тобто спеціальної структури, що зберігає метадані об’єкта стосовно організації відповідної сутності в самому джерелі й програмі.

Принцип реалізації універсальності роботи з джерелами даних у представленій моделі буде пояснено далі за допомогою діаграми компонентів уніфікованої бібліотеки, що можна знайти у додатку Б.

Так, першим компонентом на діаграмі є «Runtime Platform». Цей компонент співвідноситься з платформою виконання програмного застосунку. Основними інтерфейсами, що він надає є: інформація про бінарні модулі (бібліотеки), інформація про класи та їх властивості.

Переважно цю інформацію дозволяють отримати мови програмування за допомогою таких механізмів, як RTTI (динамічна ідентифікація типу) та рефлексія. Ці механізми доступні, зокрема, в таких мови, як Delphi, C++, C#, Java, Python тощо. Проте за відсутності можливості перегляду цієї інформації у процесі виконання застосунку при використанні виключно внутрішніх механізмів обраної технології можна власноруч реалізувати альтернативний механізм отримання.

Наступним компонентом бібліотеки є модуль «DataCore». Він включає такі підсистеми:

— EntitiesManager – програмний менеджер сутностей, що зберігає інформацію про присутні у джерелі даних реєстри сутностей та керує створенням та ініціалізацією програмних об’єктів, що відповідають сутностям;
— DataObjectSchema – схема певної сутності, що дозволяє отримати інформацію щодо членів даної сутності та метадані самої сутності;
— DataStorageProvider – постачальник об’єктів для взаємодії з різноманітними джерелами даних.

До функціональності EntitiesManager (див. додаток Б) також входить автоматичне динамічне визначення типів сутностей та схем даних з бінарних модулей, що передаються до менеджера. Цей менеджер зберігає схеми для кожного типу сутності, що дозволяє підвищити продуктивність роботи програми.

У програмі компонент DataObjectSchema (див. додаток Б) представляється якимось типом, що реалізує інтерфейс IDataObjectSchema. За допомогою одного типу схеми можна створити множину його екземплярів, кожен з яких містить інформацію про різні сутності. Проте за необхідності можна визначити інші інтерфейси, успадковані від базового IDataObjectSchema, та вказати вибраним сутностям схему якого інтерфейсу для них використовувати.

Кожен об’єкт сутності містить посилання на DataStorageProvider (див. додаток Б) – свій постачальник джерела даних. Тому можна створювати різні об’єкти однієї сутності, які насправді відносяться до відмінних джерел даних. При цьому об’єкт повністю абстрагується від свого джерела і взаємодіє з ним через спільний для всіх джерел даних інтерфейс.

Постачальник може містити джерело даних за замовчуванням, проте в метаданих кожного реєстру сутності можна зазначити конкретне джерело даних, до якого відносяться об’єкти відповідної сутності, що буде застосовуватися замість того, що за замовчуванням.

Крім того, DataCore (див. додаток Б) включає в себе інші елементи:

— DataEntity – абстрактний програмний тип, що в цілому специфікує сутності з джерела даних;

— DataObject – реалізація DataEntity, представляє екземпляр певного реєстру, тобто об’єкт, що відповідає сутності з джерела даних;

— DataList – реалізація DataEntity, представляє список, що може включати декілька екземплярів певного реєстру.

Кожна сутність використовує інтерфейс IDataObjectSchema для можливості перегляду власних метаданих та членів.

Для роботи зображеної на діаграмі бібліотеки потрібна реалізація інтерфейсу IDataStorage (див. додаток Б) – конкретного джерела даних. В свою чергу вона надає такі інтерфейси, як: IDataObject, IDataList, IEntitiesManager.

Підсумовуючи надану до даного моменту інформацію, уніфікована модель бібліотеки надає інфраструктуру для абстрагування сутностей даних від джерел, в яких вони розташовані. Також у ній передбачено спеціальні елементи для підвищення продуктивності роботи бібліотеки і зручності її використання: менеджер та схему.

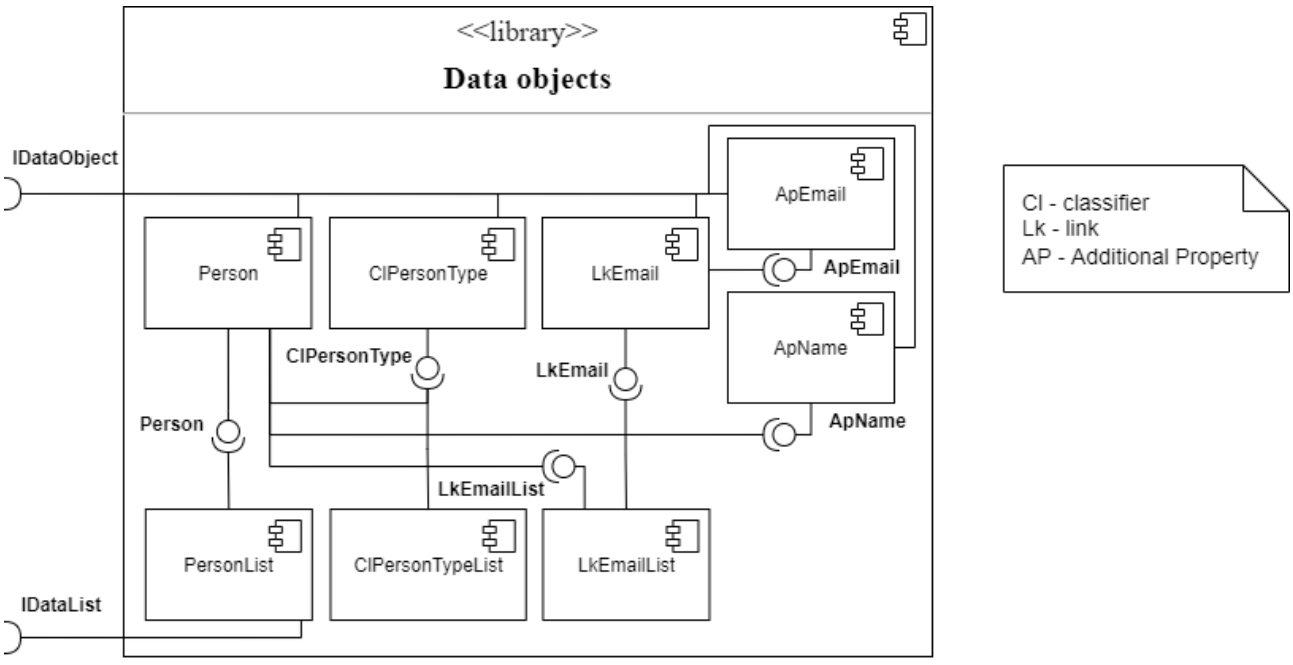


Рисунок 2.2 — Діаграма компонентів бібліотеки з прикладами об’єктів даних

Для наведення прикладу реалізації об’єктів даних на рисунку 2.2 зображено діаграму компонентів, що містить бібліотеку «Data Objects» з різними об’єктами даними та коментар з розшифровками абревіатур у назва її складових. Так, у цьому компоненті містяться типи, що відповідають сутностям, присутнім у якомусь джерелі даних.

Як видно з рисунку, класи, що описують одиничний екземпляр наслідують IDataObject. Для сутностей, що можуть використовуватися у групі, додатково розробляється клас відповідного списку, що наслідує інтерфейс IDataList. Обидва ці інтерфейси визначені у DataCore на діаграмі уніфікованої бібліотеки (див. додаток Б), опис якої було надано вище.

Для прикладу в Data Objects наведено кілька типів, пов’язаних з сутністю Особа (див. рисунок 2.2).

Клас Person представляє певну особу: це може бути як фізична особа, так і юридична, що визначається за допомогою пов’язаного об’єкта типу (ClPersonType). Для цієї сутності передбачено також використання списків. Для цього розроблений клас PersonList.

ClPersonType – клас для типу особи, тобто екземпляр цього класу являє собою об’єкт класифікації з відповідного систематизованого переліку; сам перелік чи його частина доступна через відповідний список ClPersonTypeList.

Клас ApEmail представляє таку додаткову властивість особи, як електронна адреса, а клас ApName – ім’я.

LkEmail – клас для зв’язування адреси та особи; оскільки в особи може бути декілька адрес, то присутній також клас-список LkEmailLis.

Зрозуміло, що реальне рішення включає значно більше сутностей і класів.

Перейдемо до розгляду компонентів, що є прикладами основоположних реалізацій бібліотеки. Вони зображені на діаграмі компонентів у додатку В.

Основну частину цієї діаграми займає приклад реалізації IDataStorage для реляційних баз даних. Як було згадано, реалізація об’єктів даних незалежна від реалізації джерела, об’єкт отримує посилання на своє джерело завдяки постачальнику джерел, що займається прив’язкою певного джерела до типу. Тому

на діаграмі реалізацій зображено, що інтерфейс IDataStorage (див. додаток Б) надається компоненту DataStorageProvider в DataCore (див. додаток В) реалізацією.

Розглядаючи бібліотеку DBGCore варто зазначити призначення її основних складових частин:

— DBDataStorage в загальному випадку є реалізацією IDataStorage для реляційних БД, проте в дійсності програмні джерела реалізують інтерфейс IDBDataStorage, що додатково до базового інтерфейсу містить об’єкт з’єднання до БД, що реалізує IDBGConnection;
— StdSqlBuilder – автоматизований конструктор запитів до БД на вибір даних;
— StdDBGConnection представляє собою абстракцію над рівнем безпосереднього доступу до БД;
— StdDBGCommand – абстракція команди до БД, що виконується в рамках з’єднання;
— StdDBGCommandParameter – параметр команди;
— StdDBGate – шлюз, що представляє собою абстракцію над роботою з БД і надає доступ до основних механізмів взаємодії: зв’язку, команди, параметрів тощо;
— DBGManager – програмний менеджер шлюзів до різних БД, вміє під час виконання програми автоматично знаходити типи, що визначають шлюзи, у бінарних модулях, які йому передаються.

Крім того, що StdDBGConnection (див. додаток В) зберігає усі можливі налаштування фізичного з’єднання та надає доступ до його основних методів, він також може керувати декількома з’єднаннями нижчого рівня, тобто містити так званий «пул» з’єднань. Це сприяє підвищенню продуктивності роботи застосунку.

Об’єкт з’єднання є невід’ємною частиною будь-якого об’єкта StdDBGCommand (див. додаток В) для його використання. Сама команда може бути параметризована, при цьому параметри задаються за допомогою IDBGCommandParameter (див. додаток В) для уникнення можливих уразнень, що можуть виникнути внаслідок SQL-ін’єкцій [1, 8, 9].

Усі описані компоненти є універсальними. Вони не містять реалізацій, які стосуються особливостей тої чи іншої СКБД. Для конкретизації технології, за допомогою якої відбувається взаємодія з БД, реалізується відповідна додаткова бібліотека. У розробленому тестовому рішення це бібліотека «DBGAdo» (див. додаток В), що забезпечує програмну взаємодію за допомогою ADO .NET.

Оскільки різні бази даних з точки зору API у певній мірі відрізняються між собою, то для роботи через ADO необхідно надати специфічні для конкретної БД реалізації інтерфейсів AdoConnection та AdoCommand (див. додаток В). На діаграмі показано, що у розробленому тестовому рішенні є дві бібліотеки з відповідними реалізаціями: бібліотека MSSql – для Microsoft SQL Server і PGSql – для PostgreSQL.

## 2.3 Забезпечення універсальності серіалізації та десеріалізації об'єктів даних

Як згадувалося у попередньому розділі, модель уніфікованої бібліотеки також вирішує проблему забезпечення універсального механізму серіалізації/десеріалізації об’єктів даних незалежно від формату передачі даних.

Для реалізації інтерфейсної єдності процесу серіалізації/десеріалізації у моделі передбачено застосування наступних компонент:

— DataCursorSource є контейнером для передачі, що містить дані одного або декількох об’єктів сутності та забезпечує ітеративну обробку даних різних об’єктів за допомогою спеціально розробленого DataCursor;

— DataCursor є курсором для переміщення між різними рядками даних в рамках одного контейнера, уможливлює багатопотокову обробку даних;

— DataRecord – запис даних однієї сутності у контейнері, що може містити декілька таких записів: по запису на кожну унікальну сутність.

Завдяки розробленим інтерфейсам об’єкт даних у програмі може серіалізуватися чи десеріалізуватися. При цьому об’єкту даних неважливо за допомогою якого формату це буде відбуватися, оскільки він працюватиме з абстракціями контейнера даних та запису. Тож до задач розробника входить лише написання реалізацій цих абстракцій для бажаних форматів передачі даних.

На рисунку 2.3 за допомогою діаграми послідовності зображено процес серіалізації одного об’єкта даних у контейнер. Для даного процесу об’єкту необхідний екземпляр самого контейнера даних, в який об’єкт додасть новий запис зі своїми даними. Схема відповідної йому сутності і так присутня в кожному об’єкті.

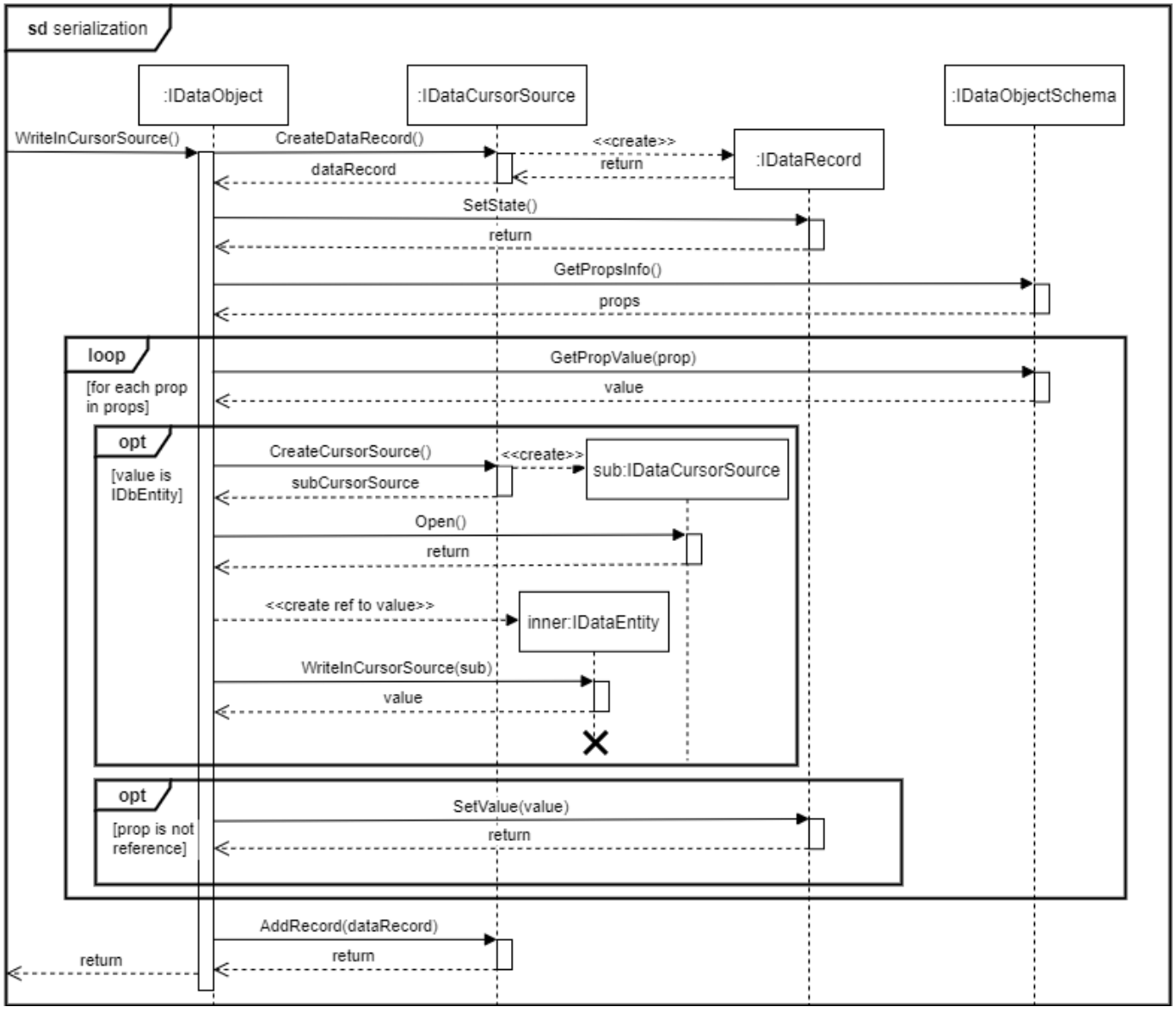


Рисунок 2.3 — Діаграма послідовності серіалізації об’єкта

Якщо вести мову не про одинарний об’єкт, а про список таких об’єктів, то процес серіалізації повторює описаний вище, але для кожного об’єкта у списку. Як результат, джерело складатиметься зі стількох записів даних, скільки об’єктів було представлено у списку на момент серіалізації.

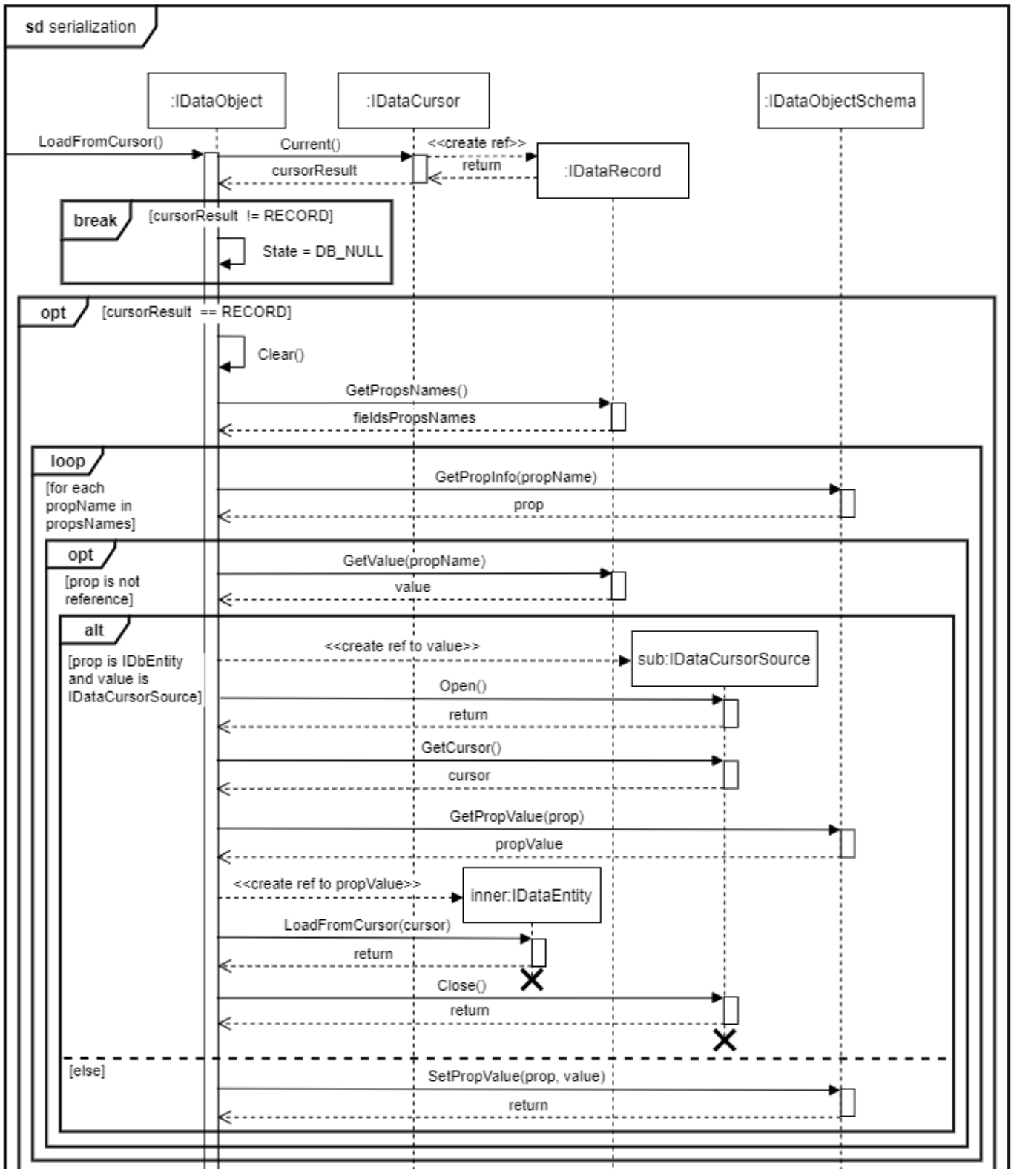


Рисунок 2.4 — Діаграма послідовності десеріалізації об’єкта

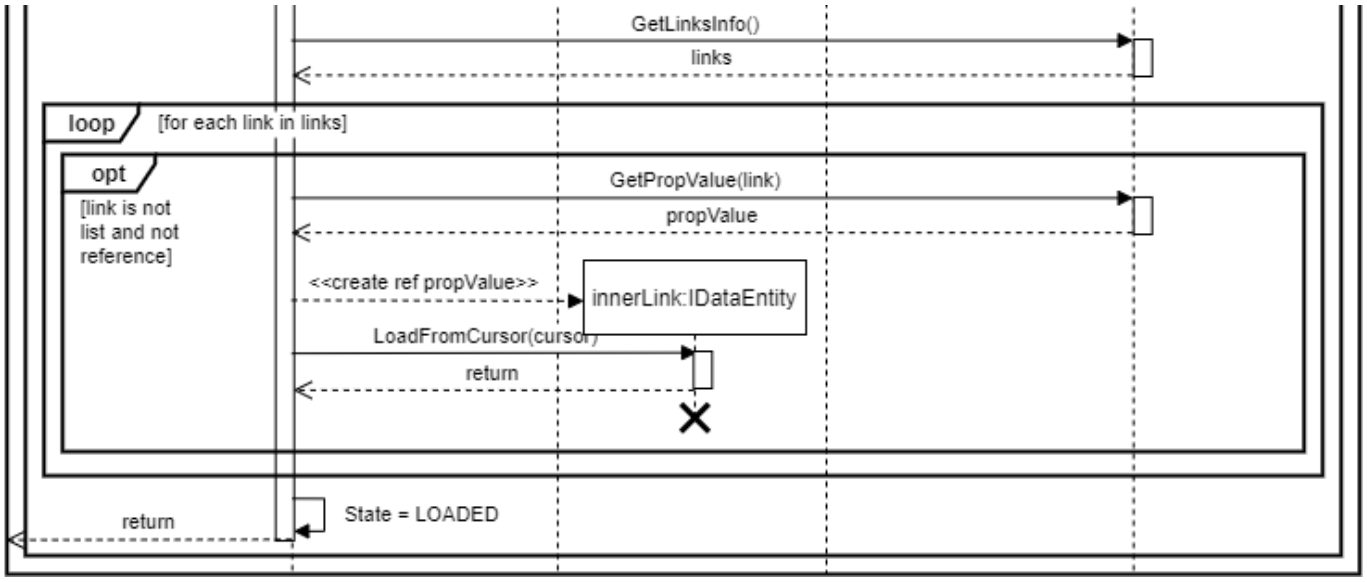


Рисунок 2.4, аркуш 2

Зворотнім процесом до серіалізації є десеріалізація, тобто завантаження об’єктом власних даних з джерела. Діаграма послідовності процесу десеріалізації зображена на рисунку 2.4.

Під час десеріалізації об’єкт працює з певним курсором, що вказує на певну позицію запису в джерелі, аналогічно з серіалізацією у ході роботи також поступають звернення до схеми відповідної сутності.

Для списку даний процес повторюється: поки курсор вказує на існуючий у джерелі запис даних, для кожного запису створюється об’єкт даних, що завантажується з нього, та відбувається перехід далі за допомогою курсора.

## 2.4 Висновки

В даному розділі було надано опис механізмів, завдяки яким модель забезпечує уніфікацію бібліотеки. Також було розроблено дві діаграми компонентів для пояснення структури бібліотеки, а саме: діаграма компонентів для моделі уніфікованої бібліотеки, а також діаграма з прикладами основоположних реалізацій.

Основною перевагою розробленої моделі є забезпечення універсальності роботи з джерелами даних: процес взаємодії з об’єктами, що представляють певну

сутність у джерелі, є однаковим незалежно від типу джерела.

Також у роботі передбачено універсальність серіалізації та десеріалізації об’єктів даних. Ці процеси відбуваються незалежно від деталей реалізації формату даних, що можливо завдяки їх абстрагуванню.

На діаграмі компонентів уніфікованої бібліотеки зображено основні елементи моделі, що забезпечують уніфіковану роботу з даними всередині програми. Ці дані можуть зберігатися в різних сховищах.

Діаграма реалізацій, в свою чергу, містить приклад виконання бібліотеки згідно з моделлю. На ній зображено можливі реалізації описаних у моделі інтерфейсів та принцип взаємодії компонентів між собою.

У підсумок, варто зазначити, що модель призначена для розробки уніфікованих бібліотек, що застосовуються на рівні бізнес-логіки. У системі така бібліотека виконує роль «зв’язуючої ланки» для середнього рівня і рівня даних.

# 3 ВИБІР ТА ОБГРУНТУВАННЯ ТЕХНОЛОГІЙ ДЛЯ РЕАЛІЗАЦІЇ

Для перевірки працездатності розробленої моделі уніфікованої бібліотеки для представлення та обробки складнозв’язаних даних в об’єктно-орієнтованій парадигмі було реалізовано тестову бібліотеку на її основі. У даному розділі наведена інформація щодо основних вибраних технологій реалізації тестового програмного рішення.

## 3.1 Цільова платформа

Виконана в рамках даної магістерської дисертації тестова бібліотека розроблена за допомогою програмної платформи .NET (раніше відомої як .NET Core) [10]. На відміну від свої попередників, дана платформа має відкритий код, є безкоштовною та найголовніше — кроссплатформеною. Це означає можливість запуску розробленого на її основі комп’ютерного програмного забезпечення на різних операційних системах: Windows, Linux, macOS. .NET повністю підтримує C# та F# (також C++/CLI для Windows станом на 3.1), а також Visual Basic .NET.

Ця технологія представляє собою всебічну і узгоджену модель програмування для побудови як багатофункціональних бібліотек, та і застосунків, що надає привабливий інтерфейс користувача, прозорі і безпечні засобами зв'язку, а також можливість створення різноманітних бізнес-процесів. .NET підтримує наступні кроссплатформені сценарії: веб-застосунки на ASP.NET Core, застосунки командного рядка, бібліотеки та застосунки Universal Windows Platform. Також дана платформа підтримує Xamarin для розробки застосунків під мобільні пристрої, Unity для розробки комп’ютерних ігор, Azure для розробки застосунків з використанням хмарних технологій, а також такі технології для розробки настільних застосунків, як: Windows Forms та Windows Presentation Foundation (WPF).

Важливою особливістю платформи .NET є модульність: кожен її компонент оновлюється через менеджер пакетів NuGet. Окремий застосунок може працювати з різними модулями платформи і не залежить від єдиного оновлення. Отже, на

противагу .NET Framework, що оновлювався одразу цілком з використанням Центру оновлення Windows, обрана платформа дозволяє окремо оновлювати обрані модулі.

Двома основними компонентами .NET є CoreCLR та CoreFX, які можна співвіднести з Common Language Runtime (CLR) та Framework Class Library (FCL) у реалізації Common Language Infrastructure (CLI) у .NET Framework.

У якості реалізації CLI Virtual Execution System (VES), CoreCLR є повноцінним робочим середовищем та віртуальною машиною для керованого виконання програм CLI і включає компілятор часу виконання (англ. just-in-time, або скорочено JIT), під назвою RyuJIT.

У якості реалізації CLI основоположних Standard Libraries, CoreFX розділяє підмножину API .NET Framework, однак вона також постачається зі своїми API, що не є частиною .NET Framework.

## 3.2 Мова програмування та технологія тестування

Для розробки програмного забезпечення під платформу .NET була обрана така високорівнева мова програмування, як C# [11]. C# є об’єктно-орієнтованою мовою зі строгою типізацією, розроблена під егідою Microsoft.

Не зважаючи на те, що ця мова програмування перейняла багато від своїх попередників, вона спираючись на практику їхнього використання, виключає деякі моделі, що зарекомендували себе як проблематичні при розробці програмних систем. Так, мова C#, на відміну від C++, виключає можливість множинного успадкування класів.

Прийнято виділяти наступні найважливіші переваги, що надає дана мова:

- автоматичне керування пам'яттю;
- можливість створення багатопотокових застосунків;
- підтримка узагальнень, властивостей класів, просторів імен;
- метапрограмування за допомогою атрибутів;
- LINQ, лямбда вирази та замикання, а також підтримка функціонального програмування;

— розширені можливості обробки виняткових ситуацій;
— механізм абстракцій, наслідування, поліморфізм, інкапсуляція;
— вказівники на функції-члени класів, делегати;
— велика стандартна бібліотека класів, зокрема для роботи з файлами та колекціями.

В якості технології, що використовується для проведення тестування розробленої бібліотеки, було обрано відомий фреймворк з відкритим кодом сімейства xUnit, створений спеціально для платформи .NET — NUnit [12]. Він представляє собою модульну систему тестування для всіх мов даної платформи.

Розробка тестів на основі цього фреймворку в основному передбачає використання атрибутів та тверджень. Вони доступні у різноманітних варіаціях за допомогою статичних методів класу Assert.

Якщо твердження не виконується, тобто отриманий результат не відповідає очікуваному, це означає, що була порушена запланована логіка виконання через невірну роботу коду, що перевіряється. У такому випадку виклик методу не повертається, замість цього система повідомляє про помилку. Якщо у тесті міститься кілька тверджень, будь-яке, що слідує за невдалим, не буде виконано.

NUnit володіє наступними перевагами:

— тести можна запускати з консольного процесу, а в Visual Studio за допомогою Тестового Адаптера або через сторонні процеси;
— тести можна запускати паралельно;
— наявна потужна підтримка тестів на основі даних;
— підтримує кілька платформ, включаючи .NET Core, Xamarin Mobile, Compact Framework та Silverlight;
— кожен тест може бути доданий до однієї або декількох категорій, щоб забезпечити вибіркове тестування.

## 3.3 Система керування базами даних як зовнішнє джерело

У якості системи керування базами даних (СКБД) для тестової бібліотеки було обрано таку реляційну систему, як PostgreSQL [13]. Вона розробляється спільнотою, має вільну ліцензію та відкритий код. Вибір реляційної СКБД пояснюється тим, що БД цього типу найкраще підходить для збереження структурованих даних, що можуть мати складні зв’язки.

PostgreSQL працює на всіх провідних операційних системах і задовольняє набору властивостей, що гарантують надійну роботу транзакцій бази даних (ACID): атомарність, узгодженість, ізольованість, довговічність.

За статистикою станом на 2021 рік вона входить в топ 5 найпопулярніших баз даних по всьому світу та є єдиною серед свої аналогів, що має позитивну динаміку протягом останніх місяців [14].

У табл. 3.1 наведені порівняльні характеристики таких популярних СКБД, як: SQL Server, Oracle та PostgreSQL.

Таблиця 3.1 — Функціональне порівняння популярних СКБД

| Характеристика | Microsoft SQL Server | Oracle | PostgreSQL |
|---|---|---|---|
| Адміністративне керування | Добре | Відмінно | Добре |
| Графічні інструменти | Відмінно | Добре | Відмінно |
| Механізм даних | Добре | Відмінно | Відмінно |
| Одночасний доступ декількох користувачів | Добре | Відмінно | Відмінно |
| Повнотекстовий пошук | Добре | Відмінно | Відмінно |
| Єдина реєстрація | Добре | Добре | Відмінно |
| Можливості програмування | Задовільно | Відмінно | Відмінно |
| Процедури, що зберігаються, та тригери | Добре | Відмінно | Відмінно |

Продовження таблиці 3.1

| | | | |
|---|---|---|---|
| Вбудована мова програмування | Задовільно | Відмінно | Відмінно |
| Побудова БД | Добре | Відмінно | Відмінно |
| Підтримка ОО парадигми | Задовільно | Відмінно | Відмінно |
| Організація сховищ даних і підготовка звітів | Відмінно | Відмінно | Відмінно |

Таким чином, вибір PostgreSQL аргументований тим, що ця СКБД надає значні функціональні можливості та загалом не поступається своїм конкурентам.

Нижче наведено невичерпний перелік різноманітних можливостей PostgreSQL зведених за категоріями.

Типи даних:

— примітиви: ціле, числове, рядок, булеве значення;
— структуровані: дата/час, масив, діапазон, UUID;
— документ: JSON/JSONB, XML, ключ-значення (Hstore);
— геометрія: Точка, Лінія, Коло, Багатокутник;
— налаштування: Комбіновані, Спеціальні типи.

Механізми для забезпечення цілісності даних:

— UNIQUE (унікальний), NOT NULL (обов’язкове значення);
— первинні ключі;
— зовнішні ключі;
— обмеження виключення;
— явні блокування, рекомендаційні блокування.

Можливості для паралельного виконання та підвищення продуктивність:

— індексація: B-tree, композитна, за виразами, часткова;
— розширене індексування: GiST, SP-Gist, KNN Gist, GIN, BRIN, індекси покриття, фільтри Bloom;
— складний планувальник/оптимізатор запитів, сканування;
— транзакції, вкладені транзакції (за допомогою точок збереження);

— багатоверсійний паралельний контроль (MVCC);
— паралелізація запитів читання та побудова індексів B-дерева;
— розбиття таблиці;
— усі рівні ізоляції транзакцій, визначені стандартом SQL, включаючи серіалізацію;
— JIT-компіляція виразів.

Забезпечення надійності та відновлення можливе завдяки таким засобам:

— ведення журналу попереднього запису (WAL);
— реплікація: асинхронна, синхронна, логічна;
— точкове відновлення (PITR), активні режими очікування;
— табличні простори.

Безпеку в PostgreSQL забезпечують:

— аутентифікація: GSSAPI, SSPI, LDAP, SCRAM-SHA-256, Certificate тощо;
— надійна система контролю доступу;
— безпека на рівні стовпців і рядків;
— багатофакторна автентифікація за допомогою сертифікатів та додаткового методу.

Завдяки наступним можливостям у PostgreSQL можлива розширюваність:

— збережені функції та процедури;
— процедурні мови: PL/PGSQL, Perl, Python та багато інших;
— вирази пошуку за шляхом SQL/JSON;
— сторонні обгортки даних: підключення до інших баз даних або потоків за допомогою стандартного інтерфейсу SQL;
— налаштування інтерфейсу зберігання для таблиць;
— багато розширень, що надають додаткові функціональні можливості, включаючи PostGIS.

Для інтернаціоналізації та пошуку тексту можна використовувати:

— підтримку міжнародних наборів символів;
— «нечутливість» до регістру та наголосів;
— повнотекстовий пошук.

Також варто зазначити, що PostgreSQL має API та конектори для більшості мов програмування, бібліотеки для мов платформи .NET, а також забезпечує підтримку для ODBC за допомогою ODBC-драйвера psqlODBC. Згідно з інформацією розміщеною на офіційному сайті [13] PostgreSQL є високо масштабованою як за кількістю даних, якими вона може управляти, так і за кількістю одночасних користувачів. У виробничих середовищах є активні кластери PostgreSQL, що управляють багатьма терабайтами даних, і спеціалізовані системи, що керують петабайтами.

## 3.4 Висновки

В даному розділі було надано опис та обґрунтування технології, що були застосовані під час реалізації тестової бібліотеки на основі запропонованої моделі уніфікованої бібліотеки для представлення та обробки складнозв’язаних даних в об’єктно-орієнтованій парадигмі.

Так, в якості цільової платформи було обрано останнє рішення від Microsoft — .NET. Ця платформа має відкритий код та є кроссплатформеною, що дозволяє розробнику системи використовувати дану бібліотеку незалежно від ОС, на якій він працює. В рамках .NET підтримується три мови програмування, для реалізації уніфікованої бібліотеки перевага було віддана високорівневій об’єктно-орієнтованій мові програмування C#. В якості засобу для тестування розробленого рішення було обрано NUnit — фреймворк для модульного тестування ПЗ на платформі .NET.

Також у даному розділі зазначається, що у бібліотеці передбачено взаємодію з такою СКБД, як PostgreSQL, де дані зберігаються на віддаленому сервері у базах даних, які виступають у якості джерела даних. Зокрема, ця СКБД надає вільну ліцензію та необхідні засоби, а також володіє технічними характеристиками та функціональними можливостями, здатними задовольнити потреби тестового рішення.

# 4 БІБЛІОТЕКА ДЛЯ ПРЕДСТАВЛЕННЯ ТА ОБРОБКИ ДАНИХ НА ОСНОВІ УНІФІКОВАНОЇ МОДЕЛІ

## 4.1 Загальні відомості

Розроблена бібліотека є універсальною для застосунків, що розробляються для операційних систем, що підтримуються платформою .NET Core 3.1.

У розробленому програмному рішенні протестовано взаємодію тестової програми з такою СКБД, як PostgreSQL, в якості зовнішнього джерела даних.

Для роботи з БД бібліотека використовує Npgsql — постачальник даних ADO.NET з відкритим кодом для PostgreSQL [15]. Він дозволяє програмам, написаним на C#, Visual Basic та F#, отримувати доступ до сервера баз даних PostgreSQL. Npgsql є максимально сумісним з ADO.NET, а наданий ним API майже ідентичний до інших драйверів баз даних на .NET. Ця бібліотека є безкоштовною та повністю написана на C#.

Параметри підключення до баз даних передаються спеціальним класам, що відповідають за підключення. У готовому програмному рішенні, що буде включати розроблену бібліотеку, розміщення цих параметрів не обмежується. Рекомендується вказувати налаштування у спеціальних конфігураційних файлаї. Вони, як правило, розміщуються поряд з виконуваним файлом. Такі файли можуть бути без зайвих складнощів змінені адміністратором, при цьому не треба перебудовувати рішення.

До складу бібліотеки входить модуль, що містить реалізацію спеціально розробленого унікального формату даних для підвищення ефективності серіалізації/десеріалізації об’єктів даних. У поточній версії в цьому модулі передбачено фрагментарне використання JSON, для цього також підключаються бібліотеки Newtonsoft, що визнана найпопулярнішою для .NET [16].

Тож, для функціонування бібліотеки необхідні такі сторонні пакети: Npgsql.dll для роботи з PostgreSQL та Newtonsoft.Json, Newtonsoft.Json.Bson для JSON.

Загальний розмір файлів з вихідним кодом — близько 400 КБ. В той час сам проект був розроблений за допомогою інтегрованого середовища розробки програмного забезпечення Microsoft Visual Studio, універсальне рішення складається

з 3-х власних бібліотек. Діаграма розгортання універсального рішення зображена на рисунку 4.1 нижче.

Ще один проект був розроблений для тестування різних частин бібліотеки.

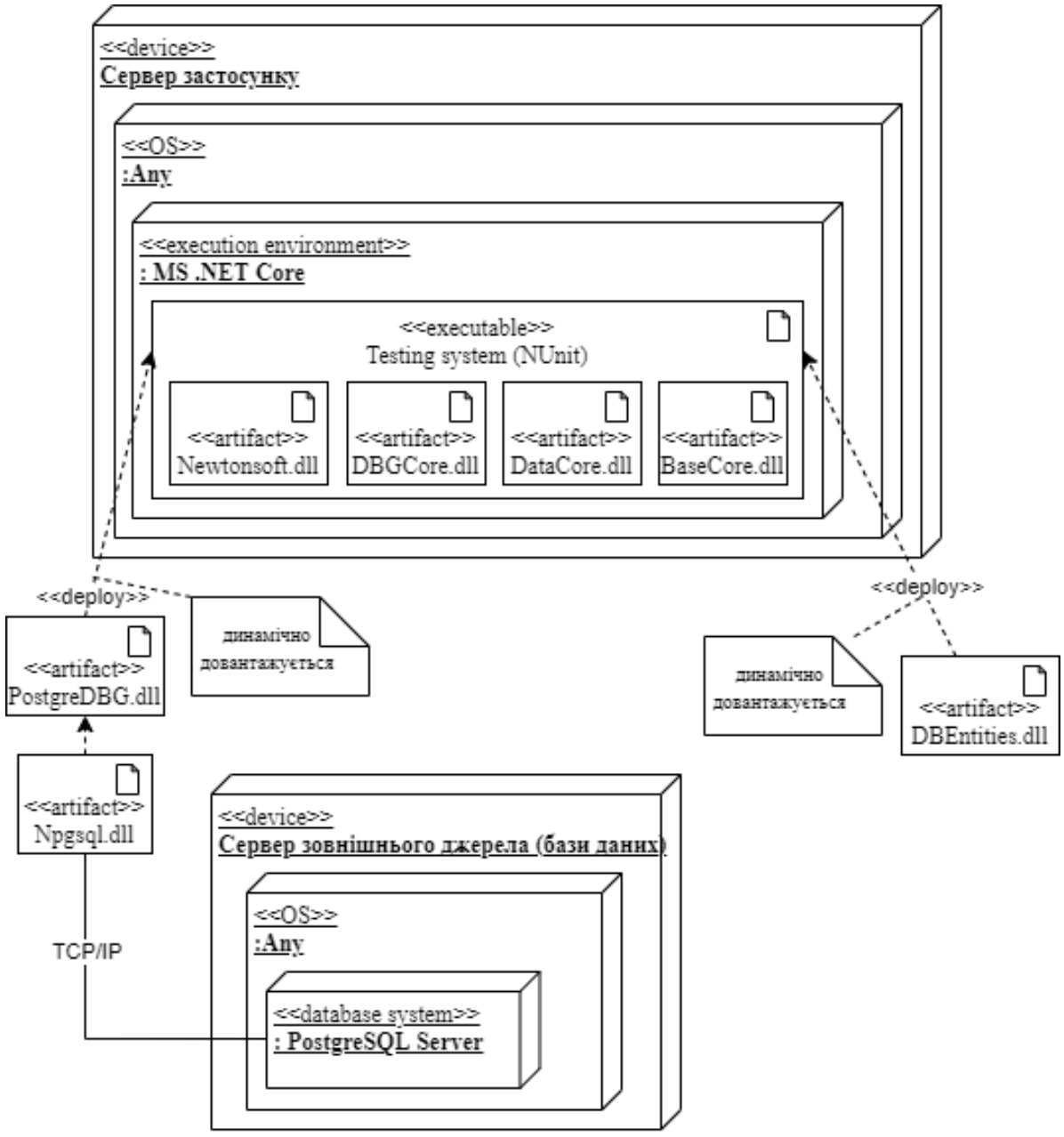


Рисунок 4.1 — Діаграма розгортання розробленого рішення

Логічну структуру бібліотеки наведено у додатку Г, що містить UML-діаграми, які зображують схему класів. Узагальнений опис основних компонентів був наведений у розділі 2 цієї роботи. У цьому розділі увага буде приділятися суттєвим деталям реалізаціям та принципам організації роботи бібліотеки.

## 4.2 Опис архітектури

Не зважаючи на те, що загальноприйнятого визначення поняття «архітектура програмного забезпечення» не існує, зазвичай під цим терміном (англ. software architecture) прийнято вважати спосіб структурування програмної або обчислювальної системи, абстракція елементів системи на певній фазі її роботи [17].

Основним задачами, що розглядаються під час розробки архітектури ПЗ є:

— вибір структурних елементів та їх інтерфейсів, що складають основу системи;
— організація поведінки в рамках співпраці структурних елементів та протоколи взаємодії;
— з'єднання обраних елементів у масштабніші системи.

На цьому етапі приймаються важливі рішення щодо самої системи. Під час дослідження архітектури системи намагаються визначити як найкраще розбити систему на частини, як ці частини визначають та взаємодіють одна з одною, як між ними передається інформація, як ці частини розвиваються поодинці тощо.

Під час проєктування уніфікованої бібліотеки застосовувався об’єктний підхід. В межах цього підходу, на основі об'єктно-орієнтованого моделювання за допомогою UML, надається можливість декомпозиції системи, визначення вимог й об’єктної моделі предметної області, шо дозволяє враховувати аспекти, специфічні для системи зважаючи на поставлені до неї задачі, так само, як особливості використання потенційного застосунку користувачами. Такий шлях дозволяє створювати сценарії виконання системи, відображати взаємодію між елементами системи, послідовності переходу потоку управління тощо.

Бібліотека дозволяє спроєктувати на її основі модульну, компоненто-базовану систему, що може бути динамічно розширена.

Перевагою розробки на основі компонентів є розмежування функціональної відповідальності між компонентами системи. За такого підходу компоненти є слабозв’язаними незалежними взаємозамінними функціональними частинами системи, які можуть бути використані багатократно.

Функціональна розширюваність системи забезпечується завдяки можливості динамічного завантаження до бібліотеки бінарних модулей. Такі скомпільовані програмні модулі ще носять назву плагінів і зазвичай виконуються у вигляді динамічних бібліотек, що використовують сервіси, що надає основна програма. Остання в свою чергу незалежно оперує плагінами і забезпечує можливість їх додання та оновлення без внесення змін в основний застосунок.

Такий підхід до побудови системи забезпечує ряд переваг, серед яких зменшення розмір основного застосунку, спрощення розробки та додання нових програмних модулей до комплексної системи.

## 4.3 Особливості реалізації ключових елементів

Ключовими елементами бібліотеки є класи, що відповідають за роботу з об’єктами даних та класи для взаємодії з базами даних. Окремо тут також буде описано деякі особливості реалізації контейнера даних для власного формату даних та структура самого формату.

### 4.3.1 Формат серіалізації та пов’язані реалізація

Загальна структура поточної версії формату серіалізації об’єктів зображена на рисунку 4.2. Як видно, структуру складають сегменти підпису, метаданих та даних.

Можна побачити, що на початку кожного логічного блока у сегменті обов’язково зазначається його тип (на рисунку він позначений як «Signature type»), що ідентифікує призначення відповідного блока. Так, наразі виділено чотири типи логічно виділених блоків:

— DS (Data Source) означає початок нового джерела з даними, що представляє собою контейнер для сутностей однієї структури (схеми);

— HR (Header Row) — опис структури об’єктів даного джерела;

— DR (Data Row) — значення одного об’єкта;

— ZZ сигналізує про кінець конкретного потоку джерела з даними.

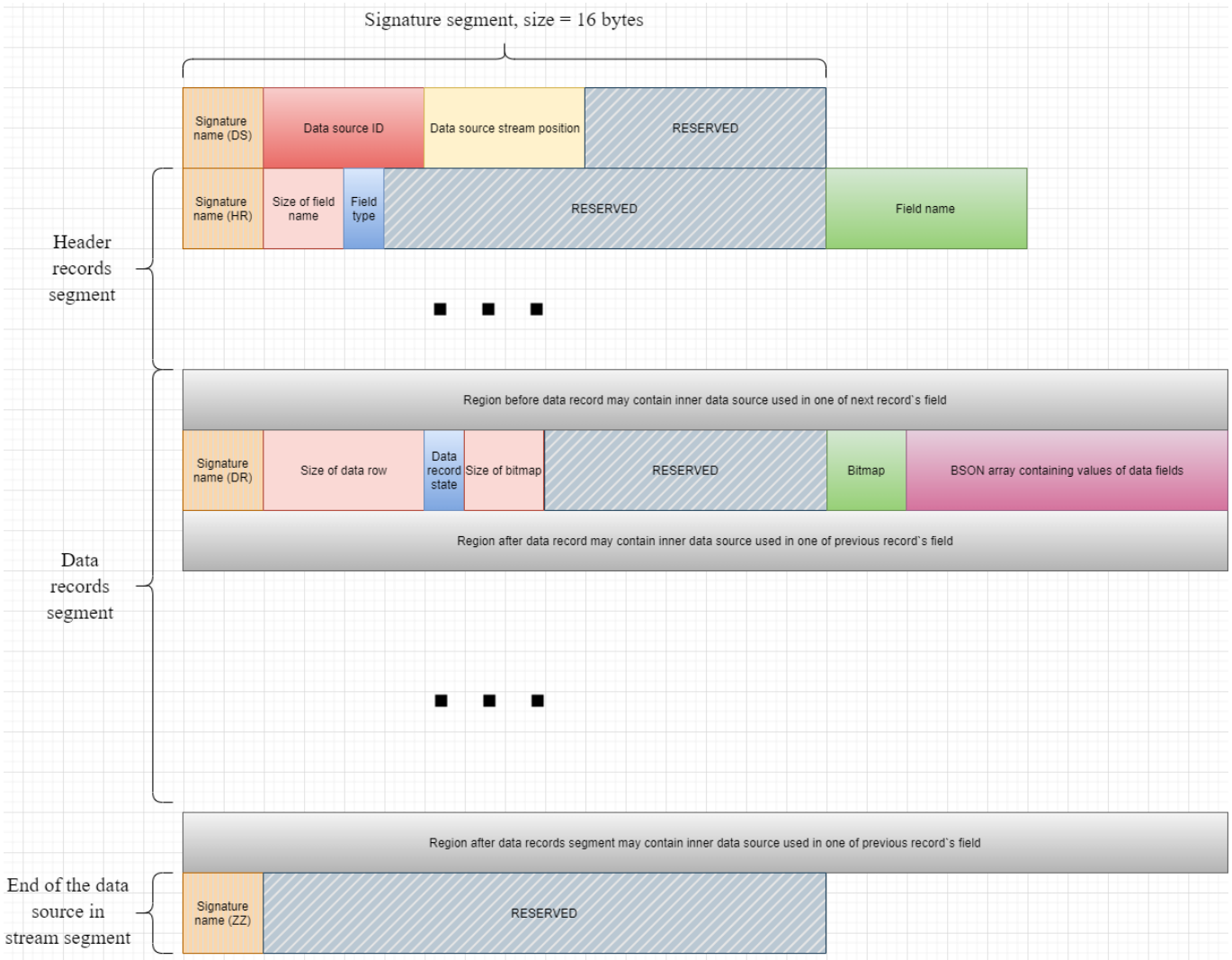


Рисунок 4.2 — Структура формату даних [2]

Варто зазначити, що кожне джерело даних зберігає в собі схему відповідної сутності, що описується за допомогою заголовків, які будемо називати терміном «header record», а також самі дані – значення властивостей об’єктів певного типу, для яких введемо термін «data record».

Маємо, що на відміну від таких популярних форматів, як XML та JSON, де назви властивостей об’єктів повторюються для кожного об’єкту, у даному форматі опис властивостей подібних сутностей зазначається лише один раз, а безпосередньо самі дані по кожному окремому об’єкту – сегмент data record-ів – слідують один за одним одразу за сегментом header record-ів. Саме це допомагає значно зменшити обсяг даних, а також пам’ять і час передачі пакетів, якщо мова йде про

«спілкування» у мережі між різними сервісами. По суті, ці показники є найбільш критичними для передачі даних через HTTP.

На початку потоку, що відповідає джерелу, розміщено сегмент підпис (на рисунку позначений як «Signature segment») — це 16-байтові блоки, що містять опис відповідної одиниці. Залежно від типу блоку тут може розміщуватися різноманітна службова інформація: ідентифікатор, інформація про тип, розмір та інше. Підпис «веде» будь-який блок даних або метаданих, на рисунку він зображений перший згори.

Якщо рухатися вниз по рисунку, далі представлено сегмент header record-ів. Як вже було згадано, він містить опис схеми конкретного типу (сутності); кожен рядок у сегменті метаданих представляє специфікацію поля даних, що відповідає властивості відповідної реляційної сутності. У підписі header record-а міститься назва поля даних та тип даних, за допомогою якого можна визначити, чи ця властивість представляє собою скалярне(просте) значення, чи є посиланням на інший набір даних (джерело), за ним — власне ім’я описаної у даному рядку властивості об’єкта.

Після цього, далі у форматі, розміщено сегмент data record-ів, що включає в себе довільну кількість рядків, що містять безпосередньо значення полів, описаних у метаданих, тобто значення властивостей об’єктів обраної сутності.

Так, один рядом даних (data record) складається з таких елементів:

у підписі

— загальний розмір даних об’єкта, тобто розмір bson-масиву, розміщеному в кінці рядку;

— розмір bitmap — спеціальної структури, що описує присутні у об’єкта властивості зі схеми;

за підписом

— сам bitmap: один біт відповідає за якесь одне поле, описане у схемі джерела (0 означає відсутність значення, 1 – присутність), дозволяється не писати замикаючі нулі;

— bson-масив, у якому відповідно до зазначено у bitmap порядку слідують значення властивостей.

Біля data record-ів можна спостерігати видовжені сірі блоки. Вони означають, що у відповідних місцях можливі вкладені набори даних інших сутностей, тобто джерела для пов'язаних об'єктів (для нескалярних властивостей об'єкта).

Заштриховані на рисунку блоки сірого кольору з написом «RESERVED» означають, що це місце у підписі блока наразі зарезервоване (визначена кількість байт) і може бути використане для майбутніх потреб.

Якщо вести мову про самі дані об'єктів, то у поточній версії формату вони розташовані у спеціальному JSON-масиві, що має двійковий форматі. Цей масив містить значення властивостей об'єкта в строгому порядку, заданому метаданими. З метою зменшення обсягу передачі даних кількість полів даних, описаних у bitmap-структурі, і відповідно кількість розташованих у масиві значень полів, може бути меншою, ніж це заявлено у метаданих, що дозволяє позбутися необхідності передавати порожні або неіснуючі значення.

Іншим засобом для покращення ефективності роботи застосунків є розроблений метод обробки серіалізованих відповідно до даного формату даних. Він надає можливість обробки даних одночасно декількома потоками, що у перспективі здатне значно пришвидшити роботу багатопотокових систем з використанням такого формату. Для пояснення принципу роботи розробленого методу розглянемо, що відбувається з заповненим відповідно до вищеописаного методу потоком даних, що передається мережею.

Так, дані з потоку зчитуються по логічним блокам та поступово записуються у відповідний об'єкт для взаємодії у коді, назвемо його відповідно до введених раніше термінів Контейнер.

Контейнер містить зв'язаний список, де зберігає усі завантажені у нього дані сутностей. Щойно з потоку було зчитано новий логічний блок з даними та передано Контейнеру, Контейнер за допомогою System.Threading.Monitor [18] блокує свій список для додання нових даних (Enter). Коли список готовий, Джерело повідомляє усім потокам, які зчитують з нього дані, що можна продовжувати роботу (Pulse) та

розблоковує список (Exit). Відповідно усі потоки, що очікували нових даних (Wait), «прокидаються» та здійснюють далі свою роботу.

Для механізму паралельного зчитування було розроблено Курсор, що вже згадувався у пункті 3 розділу 2. Він повертається Контейнером при ініціюванні читання та використовується для переміщення по даним, запам’ятовує останню позицію у списку даних для кожної задачі-читача. Один курсор може використовуватися декількома потоками, які між собою розділяють задачу читання.

### 4.3.2 Робота з об’єктами даних

Для роботи з об’єктами даних у бібліотеці передбачено набір основоположних класів, що розташовані у модулі DataCore. Відповідну діаграму класів наведено у додатку Г.

Найцікавішими з точки зору реалізації є класи, що відносяться до об’єктів даних: StdDBEntity, StdDBObject, StdDBList.

Для визначення стану об’єкта даних (DBEntityState у додатку Г) визначено перелік можливих станів:

- NOT_LOADED – об’єкт не був завантажений;
- DB_NULL – об’єкт не був завантажений, проте такий об’єкт відсутній у джерелі;
- LOADED – усі основні дані об’єкта завантажені;
- DELETED – об’єкт помічений на видалення, проте ще не збережений;
- CHANGED – об’єкт має зміни, ще не збережений.

Ці стани також використовуються при збереженні об’єкта. За їх допомогою визначається, яку діє потрібно виконати: вставка, зміна чи видалення.

Новим об’єктом вважається об’єкт, що має невизначений ідентифікатор.

Для відслідковування змін StdDBObject (див. додаток Г) має список, що містить імена змінених властивостей. Список вважається зміненим, коли змінився його зміст чи був змінений будь-який об’єкт у ньому, що список відслідковує за допомогою події StateChanged об’єкту.

Збереження об’єкта відбувається за умови, що його стан CHANGED або DELETED. Алгоритм збереження передбачає три ключові етапи: спочатку об’єкт зберігає пов’язані з ним об’єкти, від яких він сам залежить (ці об’єкти можуть змінити його зв’язуючі властивості); далі зберігає свої основні дані (деякі властивості можуть отримати нові значення від джерела, а власні зв’язуючі властивості можуть бути встановлені залежним об’єктам); після чого зберігаються залежні об’єкти.

Для видалення використовується метод MarkDeleted(), що встановлює стан об’єкта в DELETED. Якщо об’єкт належав якому списку, то цей список реагує на подію зміни стану об’єкта та вилучає його з переліку своїх об’єктів та поміщає його у внутрішній список trashObjects, об’єкти з якого при збереженні списку будуть видалені з БД.

Під час збереження списку у циклі для кожного його елемент, а також елементів списку trashObjects, виконується збереження відповідно до алгоритму описаного вище. Тобто кожен об’єкт вирішує як саме він має модифікувати джерело даних.

Завантаження об’єкту можливе, якщо його стан NOT_LOADED або явно вказано перезавантажити об’єкт. При завантаженні об’єкт передає управління програмній абстракції над джерелом, що в свою чергу формує контейнер з даними для об’єкту та отримує у нього курсор. Цей курсор передається об’єкту, що десеріалізується, використовуючи його. Процес десеріалізації був описаний у 2.3, але особливістю завантаження об’єктів з джерелом є додаткове завантаження даних пов’язаних об’єктів окремо, якщо такі відсутні у курсорі.

Для специфікації класу, що є класом даних і представляє певну таблицю з БД, над визначенням класу ставиться атрибут DBRR (від RR – reestr) або його нащадок DBObject. У них зазначається до якої таблиці БД відноситься цей клас, а також можна визначити джерело даних відмінне від джерела, визначеного по замовчуванню у постачальнику. Атрибут DBObject дозволяє додатково зазначити найменування функцій у БД, що використовуються для видалення, змінення чи вставлення запису.

Атрибут DBRR може бути використаний не тільки на клас, але й на збірку. Це зручний спосіб зазначення в одному місці даних усіх присутніх реєстрів. DBEManager може сканувати встановлені на збірку атрибути та публічні класи, що містяться в ній, при завантаженні в систему нового бінарного модуля. Він реєструє визначені в атрибутах реєстри та нащадків інтерфейсу IDBEntity, що позначені відповідним атрибутом. Для реєстрів DBEManager створює, завантажує та зберігає відповідну схему. Також при скануванні збірника цей менеджер реєструє визначені реалізації інтерфейсів схем об’єктів.

Для опису структури об’єкту та співвіднесення його членів до об’єктів БД використовуються наступні атрибути:

— DBField – проста властивість, що відповідає колонці з зазначеною назвою у відповідній таблиці (якщо в атрибуті визначено, що є ця властивість розраховується БД, то їй встановиться значення, повернене джерелом після збереження);
— DBUniqueIdentifier – властивість, що є ідентифікуючою, атрибуту передається значення, яке буде встановлене для нових об’єктів;
— DBRefProp використовується для властивості, що вказує на об’єкт іншого типу, який в свою чергу містить властивість, яка має бути встановлена відповідно до даних об’єкта, який визначає цей атрибут;
— DBRefProvValue – нащадок DBRefProp, дозволяє вказати визначене у атрибуті значення для властивості іншого об’єкта;
— DBRefPropMeta – нащадок DBRefProp, дозволяє вказати конкретне значення з мета даних за назвою;
— DBLink використовується для зв’язування з об’єктом іншого типу, зазначається вид зв’язку з переліку DBLinkType, імена властивостей, які використовуються для ідентифікації зв’язку, опціонально обов’язковість зв’язку для цілісності об’єкта та ім’я властивості для встановлення посилання, на об’єкт, що визначає зв’язок, іншому об’єкту;
— DbForeignKey – нащадок DBLink, для якого визначено обов’язковість.

Зв’язок між об’єктами може бути двох видів: IN_LINK, коли об’єкт, що визначає зв’язок залежить від пов’язаного об’єкта; OUT_LINK для зворотних ситуацій, коли пов’язаний об’єкт залежить від об’єкту, який визначає зв’язок.

Для властивостей об’єкта, шо представляють пов’язані з ним об’єкти, в атрибуті DBLink в обов’язковому порядку вказується категорія пріоритетності завантаження з переліку PropCategory:

— MAIN – основні дані без яких об’єкт не є цілісним, отримуються з джерела при будь-якому запиті на завантаження;
— LINKED_OBJECTS – пов’язані об’єкти, що не є обов’язковими для використання основного об’єкта;
— LINKED_LISTS – пов’язані списки об’єктів, також не обов’язкові до завантаження для роботи з об’єктом.

Це дозволяє зменшити кількість даних, що завантажуються з джерела у разі, коли в них нема необхідності.

За допомогою цих категорій також можна вказати бажаний «ступінь» наповненості об’єкта при завантаженні. Цей перелік помічений стандартним атрибутом System.Flags, тому при завантаженні можна використати спеціальну опцію переліку ALL_MINIMIZED = MAIN | LINKED_OBJECTS | LINKED_LISTS, що дозволяє завантажити усі можливі дані для об’єкта.

Таким чином, об’єкт може бути завантажений з мінімальним набором даних – ініціалізовані усі його властивості з категорією MAIN. За необхідності до об’єкта можна дозавантажити необов’язкові пов’язані об’єкти чи списки.

У додатку Д в якості прикладу опису структури та взаємозалежностей об’єкту наведено вихідний код класу, що представляє сутність Особа, пов’язана з ОПІВ.

### 4.3.3 Робота з базами даних

У тестовій бібліотеці у модулі DBGCore містяться класи, призначені для роботи з базами даних у якості джерела. Діаграма для головних елементів цього модуля представлена у додатку В.

Для підключення шлюзу для роботи з конкретним типом СКБД у DBGManager реєструється тип, що реалізує інтерфейс IDBGate (див. додаток Д). За допомогою шлюзів можна створити з'єднання з БД. Це з'єднання можна встановити у властивість програмній абстракції над БД або використати для виконання команд.

Об'єкт з'єднання пов'язаний з об'єктом БД, який він отримує з шлюзу. До об'єкту БД прив'язується спільна інформація, наприклад, кеші. БД ідентифікує ім'я сервера, на якому вона розташована, та ім'я самої БД на сервері.

Стандартна реалізація IDBGConnection (див. додаток Д) є абстракцією над множиною з'єднань рівнем нижче, але для виконання команд всередині транзакції необхідно, щоб усі вони виконувалися з використанням одного спільного з'єднання нижчого рівня. Тому було створено окремий інтерфейс IDBGTransactionConnection та його реалізацію. Транзакційне з'єднання отримується у звичайного з'єднання.

## 4.4 Висновки

Отже, у даному розділі був наданий стислий опис розробленої на основі запропонованої моделі бібліотеки для представлення та обробки складнозв'язаних даних в ОО парадигмі.

Для роботи бібліотеки необхідні наступні сторонні компоненти: PostgreSQL-сервер, Npgsql – бібліотека для роботи з СКБД PostgreSQL, а також бібліотеки Newtonsoft.Json, Newtonsoft.Json.Bson для роботи з JSON.

Бібліотека спроєктована таким чином, щоб уможливити створення модульних, компонентно-орієнтованих та динамічно розширюваних систем на її основі.

Додатково до опису головних компонентів бібліотеки, у третьому підрозділі було детальніше розглянуто особливості реалізації ключових елементів бібліотеки. Серед них: формат серіалізації та пов'язані з ним класи для багатопотокової обробки даних; класи для взаємодії з базами даних; класи для роботи з об'єктами даних. Для них у загальних рисах було описано принципи роботи, механізми взаємодії класів між собою та їх призначення.

# 5 ТЕСТУВАННЯ БІБЛІОТЕКИ ДЛЯ ПРЕДСТАВЛЕННЯ ТА ОБРОБКИ ДАНИХ НА ОСНОВІ МОДЕЛІ

## 5.1 Проєктування тестової бази даних

Тестове рішення передбачає роботу з реляційною базою даних в якості джерела даних. У більшості систем джерелом інформації служить реляційна база даних, розташована на спеціально відведеній машині – сервері. Приймачами інформації у системі можуть бути як власне самі користувачі, так і компоненти системи.

Основні інформаційні процеси слідують з інформаційних потреб користувачів і визначаються переважно функціональними та нефункціональними вимоги, описаними у розділі 1.

Щодо інформаційних процесів, визначених функціональними вимогами до системи, передбачені наступні:

— збереження даних всередині програми та у БД;
— передача даних між БД та програмою, а також між компонентами системи;
— додавання нових записів до таблиць БД;
— видалення чи змінення існуючих у таблицях записів;
— пошук даних, формування вибірки з однієї або кількох таблиць із застосуванням умов фільтрації та сортування;
— можливість виконання довільних команд на сервері БД;
— обробка та перетворення інформації.

Також нефункціональні вимоги визначають додаткові правила для організації інформаційних процесів:

— інформація, що пов’язана з декількома сутностями, не може бути видалена поки вона є необхідною для якоїсь з пов’язаних сутностей (що переважно досягається завдяки зовнішнім ключам);
— змінення інформації має однаково впливати на усі пов’язані об'єкти (це можуть забезпечувати тригери з умовами на операції оновлення чи видалення);

— нова інформація має перевірятися для уникнення нелогічних зв’язків та некоректних даних; відповіді на будь-які запити від користувача повинні бути вірні та зрозумілі;
— має гарантуватися цілісне збереження даних для основного об’єкту разом з усіма пов’язаними;
— має гарантуватися стабільність роботи та захищеність від помилок;
— ПЗ має забезпечувати захист джерела даних від атак, що можуть здійснюватися з його використанням;
— зміни даних сутності та її зв’язків, що містяться у БД, можуть бути «підтягнуті» у програмі;
— бібліотека повинна чітко інформувати про результати виконання збереження даних.

Зміст БД дуже відрізняється від системи до системи, адже ПЗ пишеться для різних предметних областей й має різне призначення, відповідно інформація, що «рухається» системою – майже унікальна.

Для демонстрації можливостей розробленої бібліотеки, що передбачає роботу з реляційними БД, була обрана така предметна область, як інтелектуальна власність. Хоча цей приклад є нетиповий, він дозволяє легко зрозуміти зв’язки між основними сутностями, що робить «поріг входження» до цієї предметної відносно низьким, та водночас включає багато деталей, що дозволяє охопити більший обсяг перевірки з практичної точки зору.

Загалом до складу розробленої бази даних входять 73 таблиць та збережені процедури, які використовуються для вставлення, видалення та змінення записів.

Процес вибірки даних автоматизований в межах бібліотеки завдяки конструктору select-запитів [19].

Серед усіх таблиць основними являються 21, вони представляють окремі сутності, що відповідають об’єктам обраної для розгляду предметної області.

Найголовнішими серед них можна виділити наступні:

— Документ: згідно з предметною областю це може бути документ, що стосується діловодства по певній заявці, наказ на призначення працівника на посаду тощо;
— ОПІВ – об’єкт права інтелектуальної власності, тобто винахід, корисна модель, промисловий зразок, знак для товарів і послуг чи інтегральна мікросхема;
— Особа, пов’язана з ОПІВ: це може бути заявник, винахідник чи власник;
— Патентний повірений;
— Контрагент;
— Модель – структуроване представлення зв’язаних елементів системи, виконане у вигляді ієрархії; може використовуватися для класифікації чи, наприклад, для орієнтування підлеглості у організаціях зі складною структурою;
— Співробітник;
— Відділ;
— Посада;
— Користувач системи;
— Лог – запис дії користувача у системі.

Решта таблиць не представляють окремих сутностей предметної області, а слугують для правильного проєктування БД. Узагальнено їх можна розділити на наступні категорії:

— класифікатори – систематизовані зібрання однорідних найменувань (класифікаційних груп або об'єктів класифікації) та їх кодових позначень;
— таблиці для зв’язування (головних сутностей між собою чи головної сутності з додатковим реквізитами);
— таблиці, що містять додаткові властивості для доповнення головних сутностей довільною кількістю необов’язкових реквізитів.

У БД також включено службові таблиці для зберігання інформації необхідної для організації зв’язків між таблицями або роботи ПЗ.

В цьому пункті для стислого опису протестованих аспектів до розгляду було прийнято рішення обрати сукупність таблиць, пов’язаних з сутністю Особа, оскільки вона неодмінно присутня в будь-якій системі. Відповідна ER діаграма, що містить частину спроєктованої БД, наведена у додатку Ж.

Для зручності подання інформації у тексті будуть наводитися фрагменти цієї діаграми.

Центральною таблицею у БД є Person. Сама по собі вона не представляє жодну окрему сутність, але є базовою для будь-якої сутності, що представляє людину. Це може бути користувач, співробітник, особа, пов’язана з ОПІВ тощо. Адже одна людина може бути присутня в системі у декількох ролях, тому важливо, щоб система вміла ідентифікувати людину як єдине ціле, а не в ролі окремих осіб. Така організація таблиць у БД дозволяє одночасно уникнути дублювання даних та відокремити різні ролі, в яких може перебувати людина.

У той же час в таблицях для осіб, пов’язаних з ОПІВ, передбачено дублювання записів для однієї людини. Присутність окремих записів для тієї самої людини в різних ролях виправдана специфікою принципів ведення діловодства у сфері інтелектуальної власності: змінення якоїсь інформації щодо людини відбувається по конкретній заявці, а не в цілому для цієї людини.

Що мається на увазі найкраще пояснити на прикладі. Припустимо жінка подавала заявки на 3 об’єкти інтелектуальної власності. Після одруження жінка змінила прізвище та бажає змінити цю інформацію для останньої заявки (наприклад тому, що свідоцтво дійсне на цей час лише для цього об’єкта). Тоді у матеріалах щодо цього об’єкта буде зазначена нове прізвище, а в інших – прізвище до одруження.

Загальні принципи побудови БД засвідчують наступне правило: таблиці, що відповідають особам, містять мінімальний набір колонок для даних, що є невід’ємною частиною певної сутності. Усі необов’язкові дані, або так звані додаткові властивості (англ. additional property) розташовуються у додаткових таблицях. Назва цих таблиць формується за наступним шаблоном: ap_[характер

даних], де ap – абревіатура від additional property. Наприклад, ap_address – таблиця для збереження адрес, ap_birthdate – для дат народження, тощо.

Така схема дозволяє додавати додаткові дані до основних сутностей, не нарощуючи і не засмічуючи головні таблиці. За розробником залишається вибір, як саме організувати прив'язку додаткових даних до головних сутностей.

### 5.1.1 Опис таблиць для організації відношень

Нижче розглянуто два шляхи, за допомогою яких можна зв'язати особу з її додатковими властивостями.

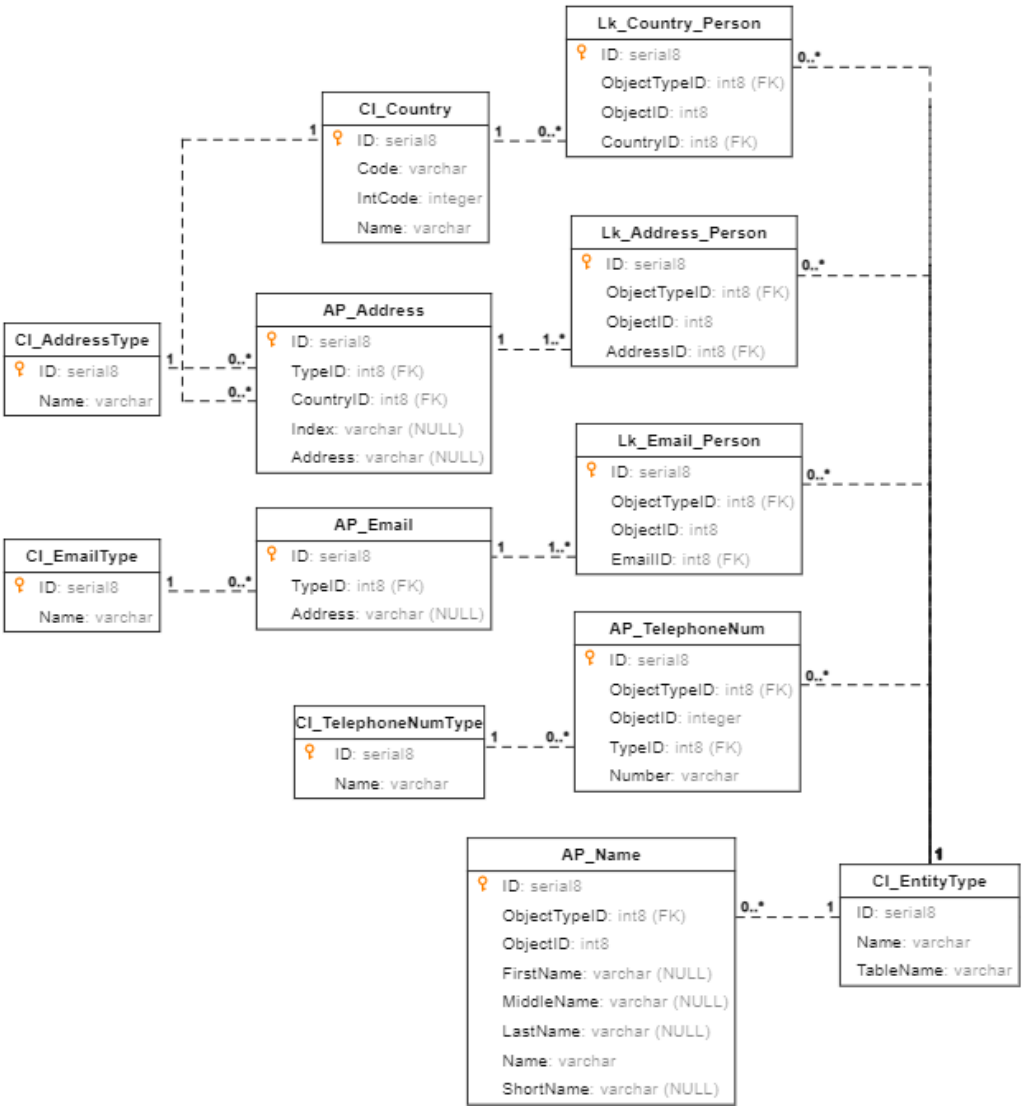


Рисунок 5.1 — Фрагмент ER діаграми для додаткових властивостей Особи

На рисунку 5.1 наведено фрагмент ER діаграми з додатку Ж. Спочатку варто звернути увагу на таблицю Cl_EntityType: вона слугує системним класифікатором сутностей БД, містить записи про усі присутні у БД головні таблиці. За її допомогою організовуються різного роду зв’язки між записами різних таблиць.

Таким чином, один з можливих варіант зв’язування додаткової властивості із записом особи можна побачити у таблиці AP_Name. Структуру цієї таблиці представлено на рисунку 5.2.

| | | Name | Data type | Length/Precision | Scale | Not NULL? | Primary key? |
|---|---|---|---|---|---|---|---|
| ✎ | 🗑 | ID | integer | | | Yes | No |
| ✎ | 🗑 | ObjectTypeID | integer | | | Yes | No |
| ✎ | 🗑 | ObjectID | integer | | | Yes | No |
| ✎ | 🗑 | FirstName | character varying | | | No | No |
| ✎ | 🗑 | MiddleName | character varying | | | No | No |
| ✎ | 🗑 | LastName | character varying | | | No | No |
| ✎ | 🗑 | Name | character varying | | | Yes | No |
| ✎ | 🗑 | ShortName | character varying | | | No | No |

Рисунок 5.2 — Колонки таблиці AP_Name

У цій таблиці колонка ObjectTypeID є частиною зовнішнього ключа, що посилається на ID таблиці Cl_EntityType, тобто значення поля вказує до якої сутності відносяться додаткові дані, зазначені у цьому рядку. Налаштуваннями даного зовнішнього ключа передбачено наступне: дія на редагування та на видалення – CASCADE.

Припустимо, необхідно зв’язати винахідника з його іменем. Тоді до таблиці AP_Name вноситься запис з даними особи (id особи у ObjectID та код таблиці, що відповідає сутності особи з класифікатора сутностей у ObjectTypeID) та заповнюється структура, що описує власне самі додаткові дані (решта колонок). А

при видаленні головної сутності необхідно видаляти й пов’язані з нею таким чином додаткові.

Така методологія зв’язування підходить для зв’язку «один до багатьох»: один телефонний номер може бути прив’язаний лише до однієї особи, але одна особа може мати декілька номерів.

Іншим прикладом організації зв’язування є зв’язування (англ. linking) через допоміжну таблицю. Імена таких таблиць у розробленій БД формуються за наступною схемою: Lk_[найменування головної таблиці]_[ найменування додаткової таблиці], де Lk – скорочення від link. Цей вид зв’язування може використовуватися для організації відношення «багато до багатьох».

Оскільки за однією адресою можуть проживати декілька людей, то за такої схеми достатньо створити один запис з реквізитами адреси у таблиці AP_Address та прив’язати його до декількох людей, створюючи по запису для кожної людини у допоміжній таблиці зв’язування Lk_Address_Person.

На рисунку 5.3 наведено опис колонок допоміжної таблиці Lk_Address_Person.

| | | Name | Data type | Length/Precision | Scale | Not NULL? | Primary key? |
|---|---|---|---|---|---|---|---|
| ✎ | 🗑 | ID | integer | | | Yes | No |
| ✎ | 🗑 | ObjectTypeID | integer | | | Yes | No |
| ✎ | 🗑 | ObjectID | integer | | | Yes | No |
| ✎ | 🗑 | AddressID | integer | | | Yes | No |

Рисунок 5.3 — Колонки таблиці Lk_Address_Person

Як видно, при такій організації зв’язку колонки, що відповідають за специфікацію атрибутів зв’язку, присутні у самих пов’язуючих таблиці. При цьому в таблиці AP_Address відсутні дані, що вказують на зв’язок з головною сутністю, на противагу попередньо описаному методу.

Варто зазначити, що цей спосіб, на відміну від попереднього, під час видалення головної сутності потребує реалізації більш складної логіки. При

використанні цього способу необхідно видаляти запис у додатковій табличці лише тоді, коли з ним не пов’язана більше жодна інша головна сутність. Це можна реалізувати на стороні сервера БД за допомогою тригерів на видалення.

Отже, на даному етапі було продемонстровано 2 способи організації зв’язків між сутностями. В обох з них для зв’язування, окрім ідентифікаторів записів з таблиць, що зв’язуються, використовується ще й допоміжне поле зі значення із визначеного переліку. Таке зв’язування має бути врахованим у програмному рішенні.

Звісно, у спроєктованій БД існують таблиці, для яких організація відношень відбувається без допоміжних полів.

Розглянемо приклад з додатку Ж (таблиця Log та User). Вони пов’язані відношенням «один до багатьох»: колонка UserID таблиці Log посилається на id таблиці User.

Прикладом реалізації відношення «багато до багатьох» є зв’язок між таблицями Worker та Department, виконаний за типовою схемою за допомогою допоміжної таблиці-зв’язку [20], що зображена на рисунку 5.4.

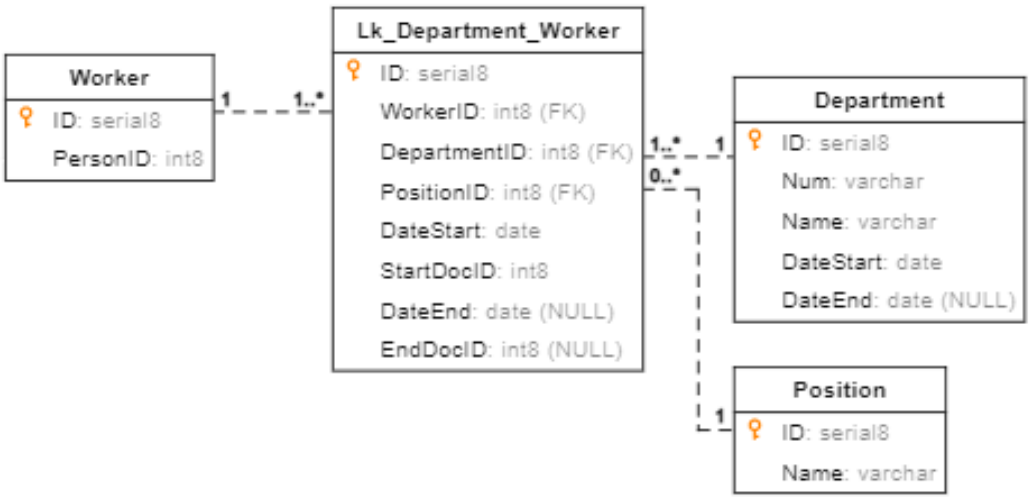


Рисунок 5.4 — Фрагмент ER діаграми для типової реалізації відношення «багато до багатьох» між сутностями Працівника та Підрозділу

Ще одним видом складних зв’язків, представлення та організацію яких, варто продумати та перевірити у бібліотеці є рекурсивний зв’язок. Таблиці у тестовій базі, на яких можна продемонструвати приклад такого зв’язку наведено на рисунку 5.5.

Таблиця Model призначена для зберігання опису ієрархічних моделей, до складу яких можуть бути включені будь-які з головних сутностей. Самі елементи моделі містяться у табличці Lk_Model.

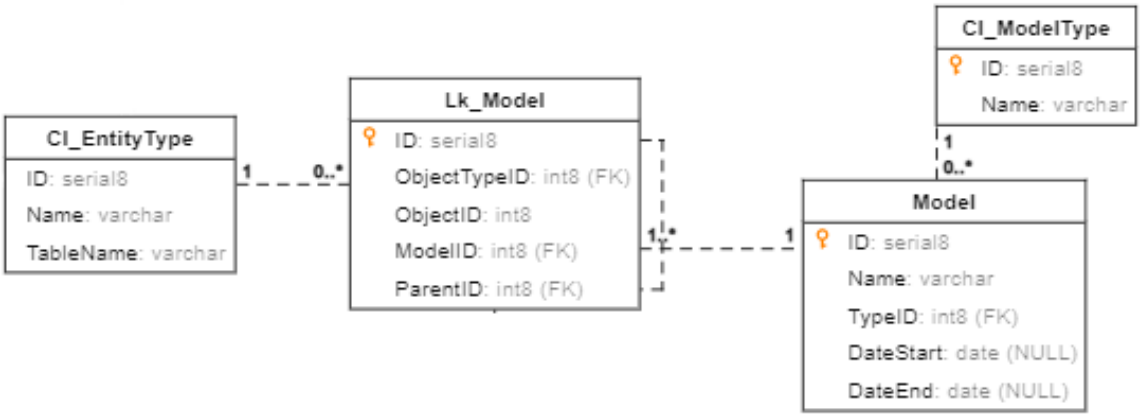


Рисунок 5.5 — Фрагмент ER діаграми для рекурсивних зв’язків

Кожен запис в цій таблиці представляє собою ієрархічну одиницю і містить посилання на батьківський запис з цієї ж таблиці. Для кореневого елемента, що знаходяться на горі ієрархії значення даного поля – NULL.

Ієрархічні посилання організовані за допомогою зовнішнього ключа для ParentID, що вказує на ID батьківського запису цієї ж таблиці. Для цього ключа була зазначена дія CASCADE на змінення та на видалення.

### 5.1.2 Застосування збережених процедур

Для зменшення використання обчислювальних потужностей на стороні клієнта, які розробники не можуть контролювати, прийнятою практикою є перенесення частини бізнес-логіки на віддалений сервер БД [21]. На ньому, наприклад, можуть бути розташовані підготовлені SQL-запити для модифікації або

отримання даних відповідно до відомих популярних інформаційних потреб користувачів.

Для цього в БД зберігаються процедури чи підготовлені функції, що можуть приймати вхідні параметри та здійснювати перетворення або вибір необхідних даних, а також встановлювати значення для параметрів на вихід. Отримана таким чином інформація повертається застосунку, що викликав цю процедуру, і відображається у ньому.

У подібних функціях, що зберігаються та виконується БД, досить зручно реалізовувати деяку бізнес-логіку системи. Такий спосіб надає можливість підвищити швидкість доступу до даних та спростити модифікації логіки порівняно з тим, коли вона зберігається у скомпільованих бінарних модулях програми.

Для побудови бізнес-логіки таким способом необхідно передбачити уніфіковані методи для основних операцій з даними. За допомогою цих методів дані можуть бути змінені як з бізнес-логіки системи, реалізованої на стороні БД, так і з вихідного коду програми. Також такі функції можуть знадобитися і для організації взаємодії зі сторонніми системами.

Якщо вести мову про застосування функцій БД у розробленому програмному рішенні, то було вирішено для кожної таблиці створити функції на вставлення нових записів, змінення та видалення існуючих. Варто зазначити, що у бібліотеці передбачено механізм для спрощення створення усіх цих функцій, проте не зважаючи на те, що усі ці запити можна сформувати та виконати у програмі.

## 5.2 Огляд фреймворку для тестування

У пункті 2 розділу 3 наводився загальний опис обраного фреймворку для тестування розробленої бібліотеки – NUint. Він є інтегрованим до IDE Visual Studio, що використовувалася під час розробки. Це дозволяє запускати обрані тести, використовуючи графічний інтерфейс середовища, переглядати у ньому результати виконання тестів. Самі тести за замовчуванням групуються за тестовим класом на вкладці «Оглядач тестів» (англ. Test Explorer) та можуть бути відсортовані в рамках

класу за визначеними практичними властивостями.

Серед множини технічних можливостей даного фреймворку у даному розділі будуть описані лише ті з них, що використовувалися під час написання тестів до основної функціональності тестової версії бібліотеки.

Аналогічно з розробленою бібліотекою, NUint використовує атрибути для декларування тестових класів та тестових методів всередині класу. Варто згадати, що у C# будь-який атрибут є класом, успадкованим від базового класу Attribute. У випадку NUnit усі атрибути, що використовуються при написанні коду тестів, є нащадками класу NUnitAttribute. Цей клас в свою чергу наслідує вище зазначений, базовий для атрибутів у C#, клас.

Атрибут не впливає на клас, значення його полів і властивостей і на виконання методів класу, якщо в тілі самого методу не передбачено використання інформації, що зберігається в атрибуті. Більш того, значення атрибута неможливо змінювати в процесі виконання коду. Значення властивостей атрибута (якщо такі визначені) зберігаються у вигляді констант у скомпільованому модулі.

Нижче надано перелік атрибутів, що використовувалися при написанні тестів для бібліотеки:

— [TestFixture] позначає клас, що містить модульні тести;
— [Test] вказує, що метод являється методом тесту, тобто таким, що може бути визваний з активного процесу виконання тестів NUnit; замість цього атрибуту може використовуватися один або декілька [TestCase], яким за допомого властивостей можна вказати налаштування та аргументи функції-тесту;
— [OneTimeSetUp] зазначає метод для підготовки середовища перед запуском тестів, такий метод виконується один раз перед запуском усіх тестів класу; альтернативою до цього атрибута є [SetUp], помічений ним метод відрізняється тим, що буде виконуватися щоразу перед кожним тестом;
— [OneTimeTearDown] метод з таким атрибутом виконується один раз після усіх тестів класу для того, щоб повернути середовище у початковий стан та звільнити усі ресурси, якщо такі були відкриті; аналогічно до

попереднього атрибуту, існує [TearDown], що виконується після виконання кожного тесту.

Як можна зрозуміти з існуючих атрибутів, фреймворк підтримує методологію написання тестів стандартної структури xUnit, що складається з 4 етапів:

— налаштування (англ. setup): об’єкт, що тестується, а в деяких випадках і середовище, що його оточує, приводяться у необхідний стан для перевірки поведінки самого об’єкта;
— виконання (англ. execution): запуск послідовності дій для перевірки об’єкту;
— перевірка (англ. verify): на даному етапі виконується безпосередньо перевірка, чи об'єкт, що тестується, повів себе саме так, як очікується;
— завершення (англ. teardown): повернення системи у той стан, який був до проведення тестування.

## 5.3 Методика тестування

До життєвого циклу тестування програмного забезпечення як стратегії ефективного тестування входять 6 етапів, зображені на рисунку 5.6.


Рисунок 5.6 — Етапи тестування

Кожен етап має так звані критерії входу та виходу, що визначають, коли етап починається та коли може завершитись. Деякі з етапів можуть виконуватися одночасно, тоді як інші вимагають спочатку завершення попереднього етапу.

На першому етапі аналізуються вимоги до системи для визначення того, що саме необхідно перевірити. Для розробленої бібліотеки це: представлення та обробка даних, взаємодія з БД та серіалізація, десеріалізація об’єктів за допомогою власного формату даних.

Другий етап планування випробувань передбачає розробку тестової стратегії, що як правило відображається у спеціальному документі, що містить назву «План випробувань» (англ. Test Plan).

Для тестування бібліотеки було складено такий план випробувань.

Випробування завантаження:

— основних даних для всіх таблиць: як одного об’єкта, так і списку об’єктів (за замовчуванням по 100 елементів);
— разом основних даних самої сутності та пов’язаних з ней (далі усіх даних): для об’єктів та списків усіх таблиць;
— даних сутностей, що були попередньо вставлені до БД під час підготовки тесту за допомогою SQL-запитів (по завершенню ці дані видаляються);
— усіх даних із сортуванням за властивостями основної сутності Документ;
— усіх даних із сортуванням за властивостями пов’язаних з Документом сутностей;
— усіх даних, що відповідають умові, складеної за властивостями основної сутності або пов’язаних (на прикладі декількох реєстрів).

Випробування збереження та видалення:

— основних даних на прикладі сутності ОПІВ;
— усіх даних включно та зв’язаних додаткових властивостей на прикладі Особи, пов’язаної з ОПІВ.

Випробування змінення:

— основних даних на прикладі сутності Користувач;
— усіх даних на прикладі сутності Контрагент.

Також до плану входять випробування багатопокового завантаження даних з БД та випробування кешування як для окремого об’єкта, так і для списків на прикладі двох таблиць-класифікаторів.

Останнім випробуванням є перевірка серіалізації та десеріалізації:

— основних даних сутностей для всіх таблиць: як одного об’єкта, так і списку об’єктів;
— усіх даних на прикладі сутності Користувач: як окремий об’єкт, так і список.

Під час тестування для об’єктів перевіряється їх стан та значення властивостей, а для списків, крім цього, ще кількість завантажених до списку об’єктів.

Під час вставки, змінення чи видалення сутностей також створюється ще один об’єкту того самого типу. Він завантажується за ідентифікатором того, що перевіряється і для нього повторюється процедура перевірки. Це зроблено для достовірності.

Пункт плану «Завантаження даних сутностей, що були попередньо вставлені до БД» гарантує перевірку роботи представлення сутностей навіть за умови, що до тестування база даних не містила жодного запису: через брак записів попередні тести вважалися би успішними і тому не можливо було б лише завдяки їм довести працездатність.

Стисла схема перевірки кешування наступна.

Спочатку для бази даних за допомогою властивості «вмикається» кешування. В якості аргументів тестовому методу передаються 2 типи сутностей (важливо, щоб ці типи були помічені атрибутом DBCachableDBO).

Для першого типу за наступною схемою перевіряється одиночний об’єкт:

— створюється об’єкт-прототип і завантажується з БД;
— перевіряється його стан, щоб впевнитись, що це не DB_NULL;
— створюється новий об’єкт-копія, якому встановлюється ідентифікатор як у об’єкта-прототипу, та завантажується з кешу;
— перевіряється стан об’єкта, він має бути LOADED.

Для другого типу перевіряється кешування повністю усієї таблиці:

— створюється список-прототип, який далі повністю завантажується з БД;
— перевіряється його стан, щоб впевнитись, що це не DB_NULL, а також, що кількість елементів списку дорівнює загальній кількості запитів у таблиці;
— в циклі для кожного об’єкту-елемента списку створюється новий об’єкт з ідентифікатором, як у елемента списку на поточній ітерації, та завантажується з кешу, перевіряється, що його стан LOADED;
— створюється новий список-копія, якому встановлюється ідентифікатор як у списку-прототипу, та завантажується з кешу;
— перевіряється стан, він має бути LOADED, а також кількість завантажених елементів всередині списку.

Вкінці тесту опція кешування «вимикається».

На третьому етапі формулюються конкретні набори тестів, для яких вказується входи, умови виконання, послідовність дій для перевірки та очікуваний результат. Рекомендується розробляти набори таким чином, щоб «покрити» якнайбільший відсоток коду, що перевіряється, тобто щоб були пройдені усі можливі гілки.

На даний момент набір тестів у програмному рішенні складається з 20 одиниць: 4 тестових метода для перевірки процесу серіалізації та десеріалізації, 16 – роботи з даними та БД.

На етапі під назвою «Підготовка середовища» відбувається налаштування та «розгортання» зовнішнього програмного середовища. Не варто плутати цей етап з етапом налаштування у життєвому циклі тесту: тут відбувається підготовка інфраструктури для проведення тестування, а не підготовка самої системи.

Назва передостаннього етапу каже сама за себе: тут запускаються тести та переглядається їх результат.

На останньому етапі, як правило, створюється звіт з результатів тестування, в якому узагальнюється весь процес тестування та надається порівняння між очікуваними результатами та фактичними. До кожного тесту наводиться опис його цілей, витрачений час, загальні витрати. В решті-решт, вказується загальне покриття, якого вдалося досягти тестуванням, та виявлені дефекти.

## 5.4 Результати тестування

За допомогою вбудованого у Visual Studio Оглядача тестів можна запустити модульні тести та переглянути результати випробувань. Як видно з рисунку 5.7, усі тести були виконані успішно, без помилок. Це означає, що перевірка засвідчила коректну роботу системи.

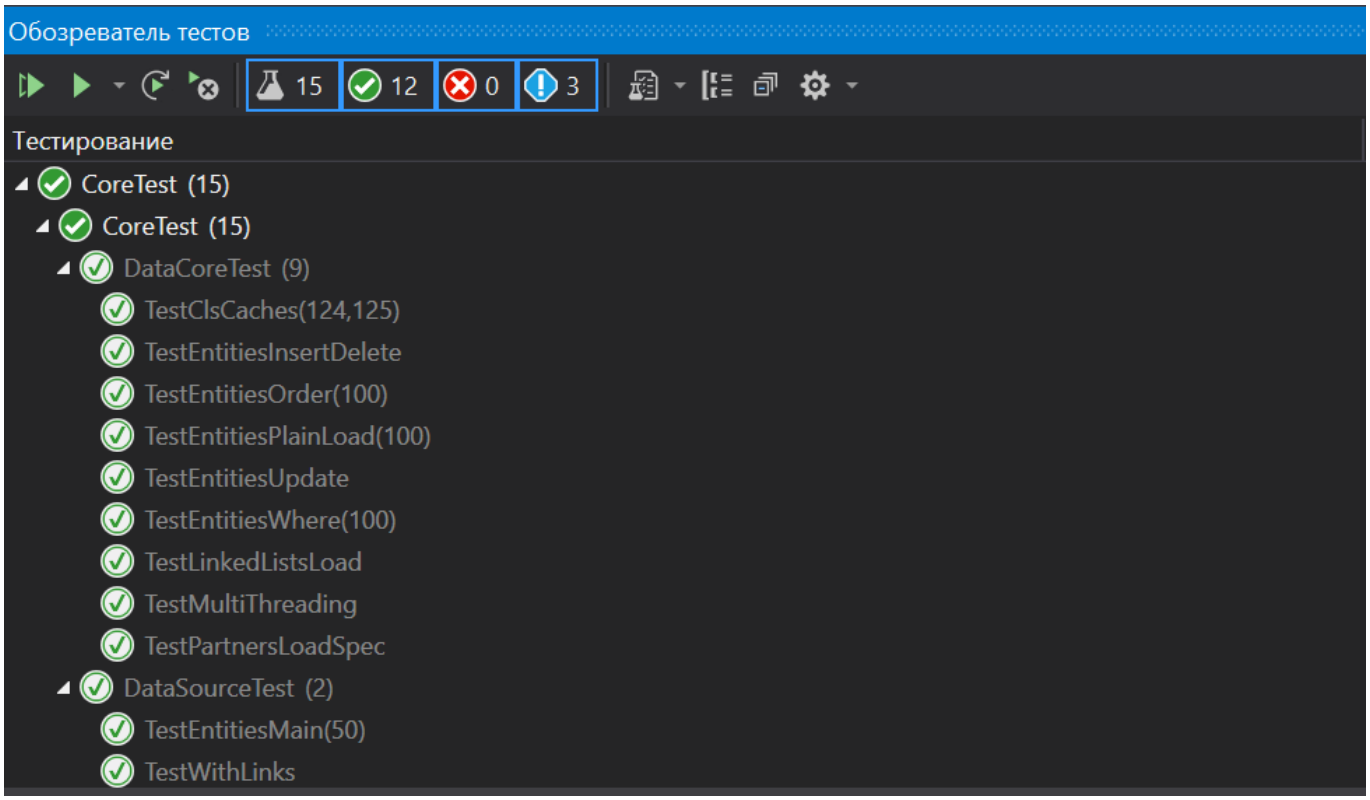


Рисунок 5.7 — Результати тестування

Середовище розробки Visual Studio має вбудований засіб для автоматичного аналізу покритого тестами вихідного коду, але воно доступне лише в Enterprise версії. Існує багато сторонніх засобів для аналізу покриття коду тестами та формування відповідних звітностей, проте більшість є комерційними. Серед безкоштовних засобів у вільному доступі найкращим з точки зору візуалізації є Fine Code Coverage.

Fine Code Coverage – безкоштовне розширення до Visual Studio, що дозволяє візуалізувати охоплення модулів тестовим кодом навіть у Community версії [22].

Воно підтримує як проекти .NET Core, так і проекти .NET Framework.

Розширення збирає дані по завершенню виконання процесу випробувань та представляє результат у однойменній вкладці, де розміщені основні відомості щодо протестованого коду: статистика покриття різних частин програмного рішення, узагальнюючі характеристики, можливі місця ризику. Також Fine Code Coverage візуально відмічає для кожного файлу з вихідним кодом протестовані і непротестовані частини всередині середовища розробки.

Статистика покриття коду, отримана за допомогою цього розширення, відображена на рисунку 5.8.

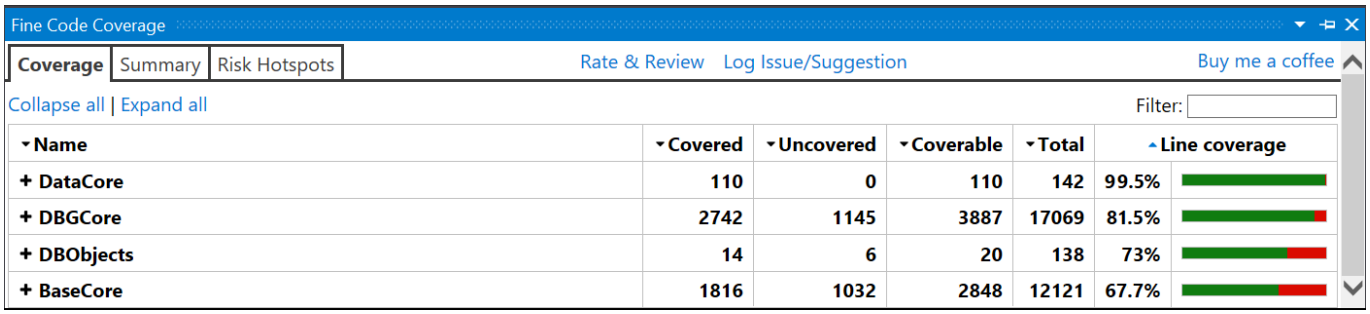


Рисунок 5.8 — Покриття коду тестами

У ході виконання тесту на завантаження, що перевіряє усі зареєстровані в менеджері сутності, здійснюються заміри часу для порівняння ефективності завантаження даних пов’язаних сутностей у запиті разом з даними головної сутностями та окремими запитами. Використовуючи отримані дані, був проведений порівняльний аналіз ефективності застосування різних спосіб для різних сутностей, результати якого описані у наступному підпункті.

### 5.4.1 Результати порівняння часу при різних способах завантаження об’єктів

Порівняльні результати для різних видів сутностей представлені за допомогою гістограм, де по осі ординат розташовані імена типів, що відповідають таблицям у БД, (в дужках зазначено скільки об’єктів завантажувалось з загальної кількості записів у БД), а вісь абсцис – чисельна шкала для результатів замірів, що

вимірюються в Ticks – спеціальна одиниця виміру часу, що співвідноситься до секунди як 1 до 10 000 000.


Рисунок 5.9 — Гістограма для порівняння часу завантаження даних сутностей з таблиць-класифікаторів

На рисунку 5.9 показано діаграму з результатами замірів по методам завантаження для типів, що відповідають класифікаторам. Був перевірений як одиничний об’єкт, так і список об’єктів. З рисунку 5.9 можна зробити висновок, що на простих класифікаційних об’єктах другий спосіб завантаження (синій колір) виграшу не дає. Це прогнозовано, оскільки для завантаження класифікаційного об’єкта не треба завантажувати пов’язані сутності.

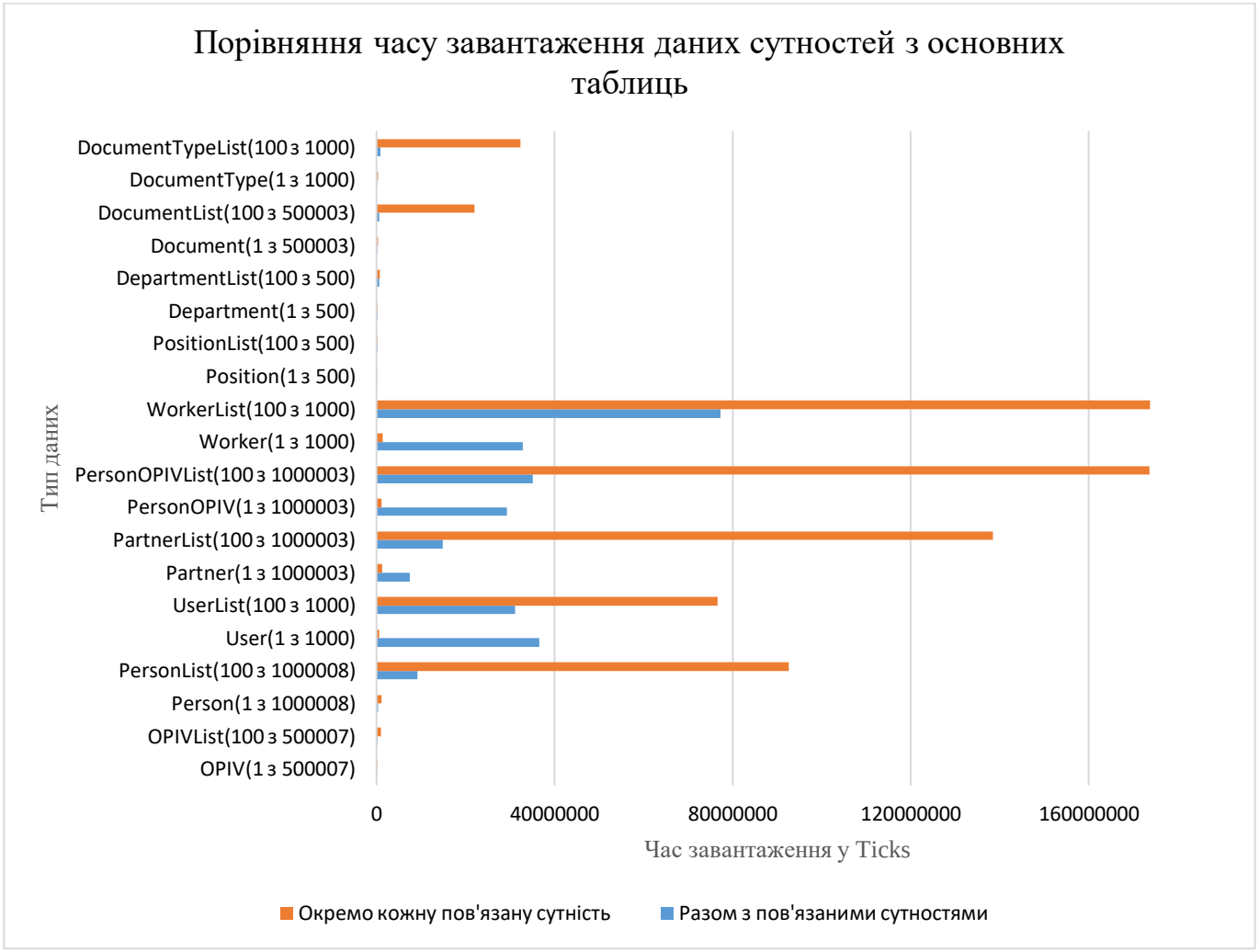


Рисунок 5.10 — Гістограма для порівняння часу завантаження даних сутностей з основних таблиць

Рисунок 5.10 показує гістограму з замірами для типів-списків, що відповідають таблицям основних сутностей. Вони представляють собою складнозв’язані об’єкти, тому використання запиту на отримання всіх даних передбачає значне пришвидшення процесу завантаження.

Результати тестування, наведені на рис. 5.10, підтверджують висунуте припущення. Для основних сутностей спосіб завантаження усіх даних за допомогою одного запиту дає значний виграш порівняно з завантаження окремими запитами даних основної сутності та пов’язаних з нею сутностей.

У рисунках 5.11 – 5.12 наводяться результати тестування для типів, що відповідають допоміжним таблицям.


Рисунок 5.11 — Гістограма для порівняння часу завантаження даних сутностей з допоміжних таблиць

На рисунку 5.11 яскраво відображена перевага завантаження єдиним запитом для списків об’єктів з таблиць, що пов’язують інші таблиці між собою. Такий спосіб дає можливість зменшити час завантаження в рази, адже для таких об’єктів обов’язково додатково необхідно завантажити два інші об’єкти, які він зв’язує. Зрозуміло, що ефективніше надіслати один запит та обробити його результати, ніж три запити.

Розглянемо детальніше решту результатів. На рисунку 5.12 зображено результати завантаження одиничних об’єктів.


Рисунок 5.12 — Гістограма для порівняння часу завантаження даних сутностей з допоміжних таблиць (лише одиничні об’єкти)

Можна побачити, що для таких об’єктів, по описаним раніше причинам, 2-ий спосіб виявляється набагато швидше, в той час як для простих об’єктів, якими являються додаткові властивості, обидва способи дають приблизно той самий результат.

5.4.2 Результати порівняння обсягу даних, що передаються, при застосуванні різних форматів

У рамках дисертації визначено задачу зменшення обсягу даних, що передаються, при обміні об’єктами. Для цього був розроблений новий формат даних, описаний у розділі 4, пункт 2.1.

З метою дослідження ефективності його застосування було проведено порівняльний аналіз обсягу даних, що міститься у потоці після серіалізації об’єктів за допомогою розробленого формату та типового формату JSON.

Обмеження на час виконання операцій не накладалися: прийнято припущення, що час виконання при проведенні експериментів відрізняється у незначній мірі.

Вхідними даними до дослідження послужили списки однотипних об’єктів даних: список додаткових властивостей, список зв’язуючих об’єктів та кілька списків об’єктів, що представляють основні сутності.

Хід проведення випробування розкриває послідовність дій, наведена далі.

У тестовій програмі створюється та завантажується з джерела список об’єктів відповідного типу. Перед серіалізацією списку для кожного формату створюється MemoryStream. До потоку за допомогою реалізацій IDataCursorSource (для власного формату та типового) записуються дані об’єкта. Для кожного з двох отриманих потоків заміряється його довжина (у байтах), тобто обсяг даних у потоці.

Цей алгоритм повторюється для списків розміром від 1 до 1000 з проміжними значеннями кратними 20 для формування результуючої вибірки з результатами.

В результаті проведення випробувань були отримані вибірки результатів довжин потоку для розробленого формату та типового формату JSON. Вони були завантажені до програми Microsoft Office Excel для проведення обробки та аналізу результатів.

Проведене дослідження показало, що обсяг даних при серіалізації списків в розробленому форматі та JSON відрізняється. За допомогою формули 1 для кожного розміру списку одного типу було розраховано відносну економію обсягу даних, що передаються, при використанні розробленого формату порівняно з JSON.

$$100 - \left( \frac{Size_1 * 100}{Size_2} \right), \tag{5.1}$$

де $Size_1$ – довжина потоку в байтах для розробленого формату;

$Size_2$ – довжина потоку в байтах для типового формату JSON.

На основі отриманої вибірки для кожного типу списку були сформовані діаграми, що показані на рисунках 5.13 – 5.17.

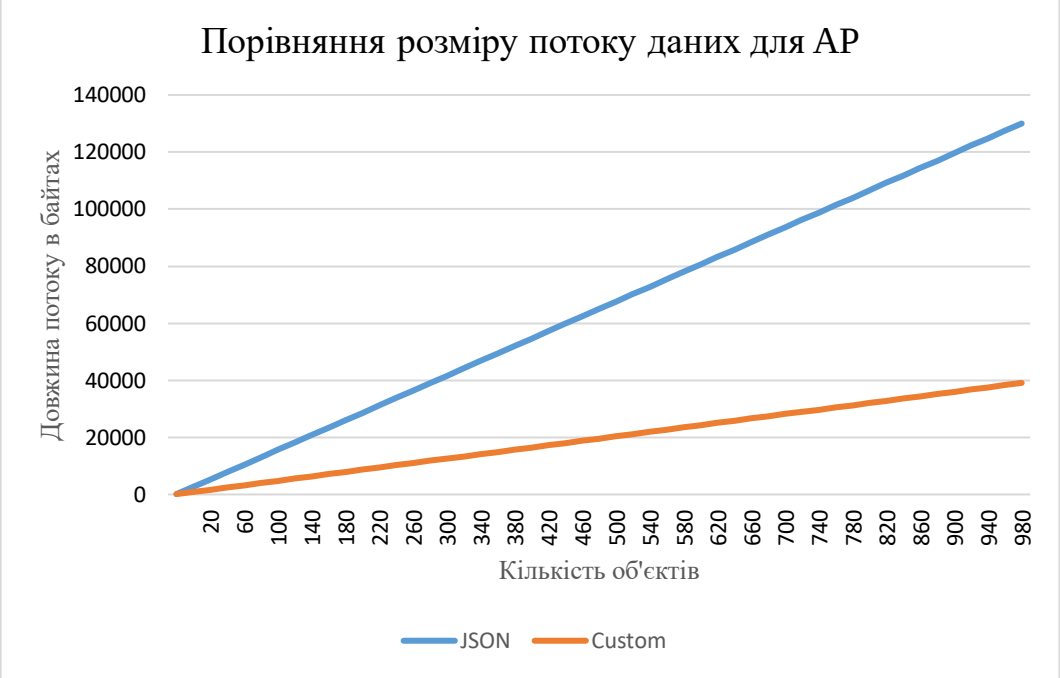


Рисунок 5.13 — Діаграма для порівняння обсягу даних при серіалізації AP_BirthDateList

На рисунку 5.13 показано результати для найпростішого об’єкту на прикладі типу AP_BirthDate – додаткової властивості, що зберігає дату народження. Результати для розробленого формату показані лінією червоного кольору, а для JSON – синього.

Можна зробити висновок, що для списку з одного об’єкта обсяг даних у потоці з використанням розробленого формату менший на 4,4% порівняно з типовим форматом JSON.

Для списку з 20 елементів перевага розробленого формату складає близько 66.5%, а для списків розмір від 40 до 1000 виграш становить 68-70%.

Далі наведено результати аналізу для списку об’єктів, що слугують для організації відношень (рисунок 5.14). Випробування проводилися на прикладі списку об’єктів класу Lk_DocumentType_PropertyType.


Рисунок 5.14 — Діаграма для порівняння обсягу даних при серіалізації Lk_DocumentType_PropertyTypeList

На рисунку 5.14 графік з результатами (кількості байт у потоці) для розробленого формату зображено лінією червоного кольору, а для JSON – синього (на наступних діаграмах аналогічно).

Згідно з розрахованими для цього типу в Excel значеннями виграшу по формулі 1, розроблений формат дає можливість зекономити значний обсяг даних.

Для списку з одного елемента на 19%.

При розмірі списку і 20-1000 елементів потік даних, отриманий при серіалізації об’єктів у нашому форматі, менше на 39-41%.

У порівнянні з попереднім випробуванням у даному випадку спостерігається менший відрив між показниками для різних форматів при серіалізації списків з багатьма об’єктами. Проте вищість розробленого формату не піддається сумніву: загальний обсяг даних внаслідок серіалізації 1000 елементів вдалося зменшити на 40%.

Далі наводяться результати досліджень для типів, що відповідають основним сутностям: ОПІВ, Особа та Користувач.

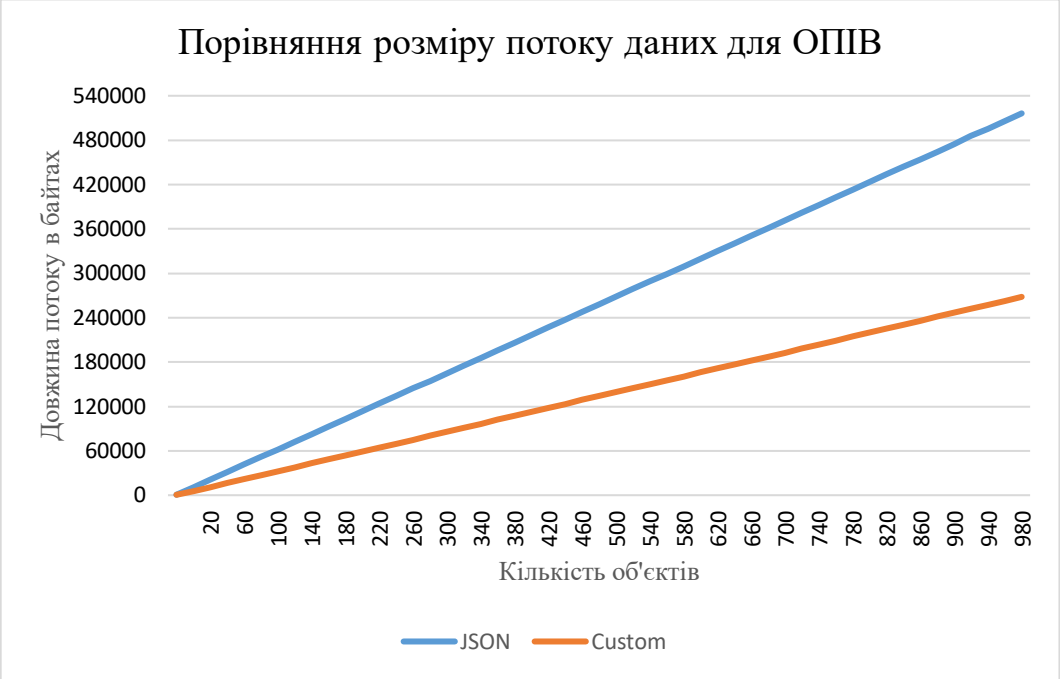


Рисунок 5.15 — Діаграма для порівняння обсягу даних при серіалізації OPIVList

Як можна побачити, використання розробленого формату дозволяє зекономити від 47 до 49% обсягу при передачі списку ОПІВ розміром від 20 до 1000 елементів відповідно. Схожі показники ефективності були отримані при серіалізації списків користувачів (рисунок 5.16): економія в середньому складає 46-47%.

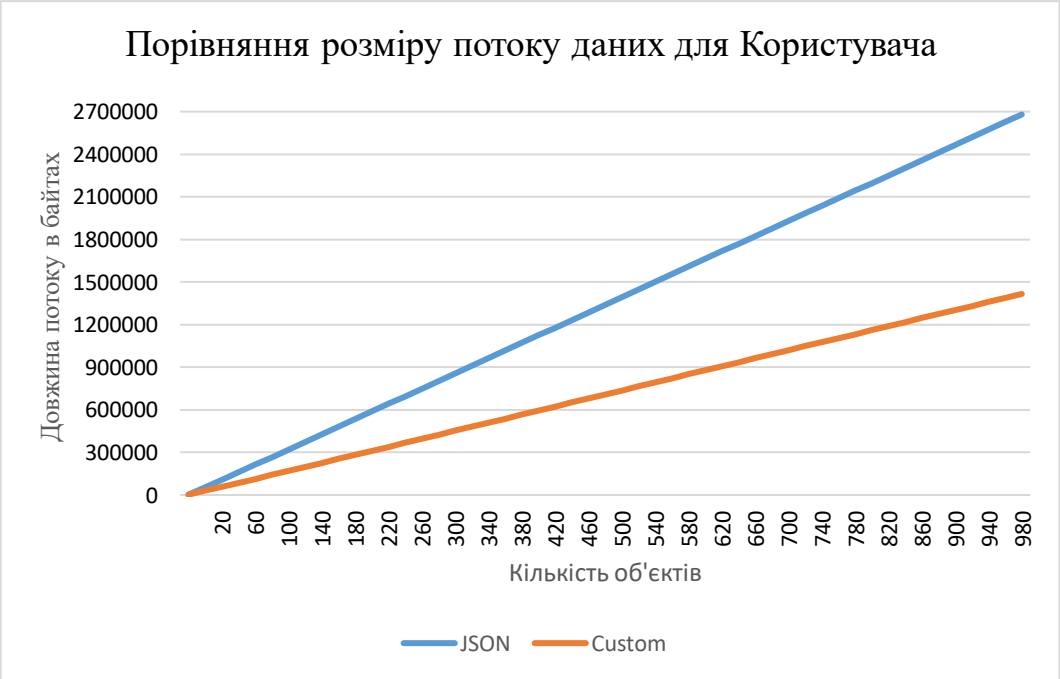


Рисунок 5.16 — Діаграма для порівняння обсягу даних при серіалізації UserList

Ще одним типом, для якого здійснювалося порівняння результатів, є Особа. Порівняльна діаграма наведена на рисунку 5.17. Якщо вести мову про списки розміром від 30 до 1000 елементів, то виграш при використанні нового формату досягає 44-45%.

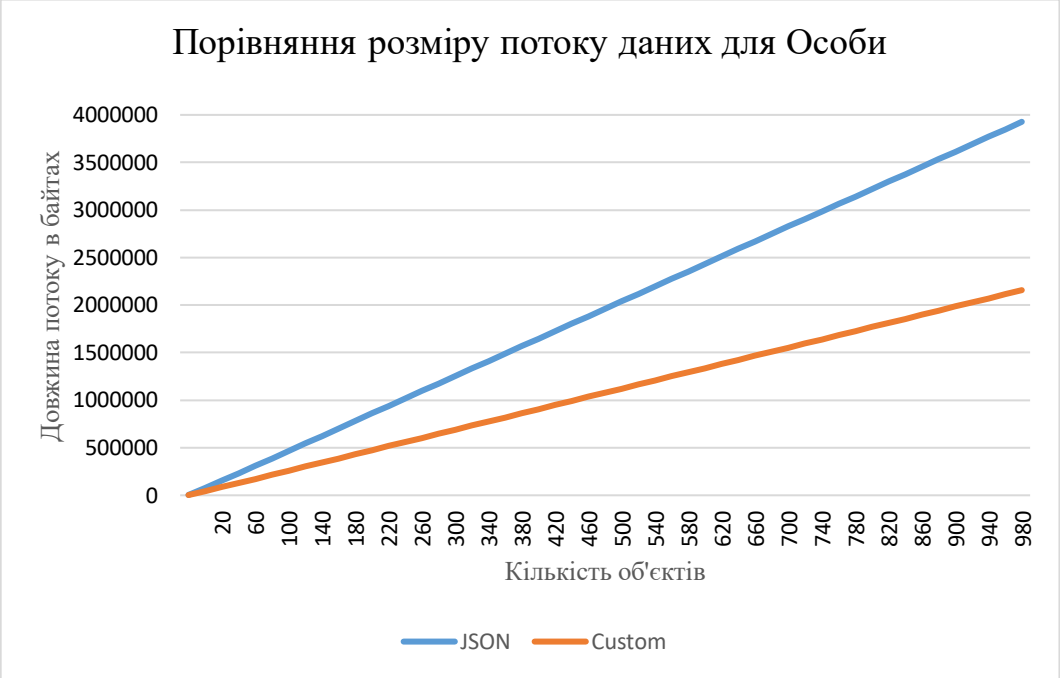


Рисунок 5.17 — Діаграма для порівняння обсягу даних при серіалізації PersonList

Як видно з останніх діаграм загальний виграш на списках комплексних об’єктів є меншим, ніж для списків простих об’єктів.

Підсумовуючи, варто зазначити, що хоча відсоток виграшу при застосуванні розробленого формату різниться в залежності від типу об’єктів, що серіалізуються, середні показники його ефективності для множини однотипних об’єктів є досить високими порівняно з типовим рішення на основі JSON. Темпи збільшення обсягу даних у потоці при збільшенні обсягу списку є стабільно меншими.

## 5.5 Висновки

У даному розділі була описана структура БД для тестової бібліотеки на основі моделі, а також спосіб організації даних у ній.

Також наводився стислий опис технічних можливостей фреймворку для тестування, що застосовувалися для перевірки основної функціональності бібліотеки. Були описані тестові випадки, що перевірялися під час випробувань, та методика тестування.

Під час тестування основними процесами, що перевірялися, були процеси серіалізації та десеріалізації, представлення даних та робота з ними, кешування, а також робота з БД. Усі тести були успішно пройдені. Показано, що вони покривають переважну більшість вихідного коду.

В кінці розділу викладені результати порівняльного аналізу ефективності застосування різних способів завантаження даних всередині бібліотеки. Проведене дослідження засвідчило, що за допомогою єдиного запиту разом даних головної та пов’язаних сутностей, можна у значній мірі пришвидшити завантаження комплексного об’єкту.

Також було проведено дослідження обсягу даних, що передаються, при використанні різних форматів даних під час серіалізації списків об’єктів. Розраховано, що за допомогою розробленого формату обсяг даних при обміні об’єктами між програмними компонентами можна скоротити в середньому на 45% на великих списках.

# 6 СЦЕНАРІЇ ВИКОРИСТАННЯ БІБЛІОТЕКИ

## 6.1 Діаграма прецедентів

Діаграма прецедентів, що іноді також називається діаграмою варіантів використання, відноситься до канонічних UML діаграм. Відповідна модель повинна відповідати на питання про те, що повинен робити програмний продукт в процесі свого функціонування, уникаючи того, як він повинен це робити. На даний момент серед розробників прийнято вважати, що основне призначення діаграми варіантів використання полягає в специфікації функціональних вимог до проектованої системи.

Діаграма представляється у вигляді графа, що складається з множини акторів (передбачені користувачі ПЗ) та прецедентів (варіантів використання) обмежених прямокутником системи. Усі об’єкти на діаграмі пов’язані між собою за допомогою асоціацій між акторами та прецедентами, відношень серед прецедентів та відношень узагальнення між акторами [23, 24]. Діаграми прецедентів відображають елементи моделі варіантів використання.

Іншими словами, кожен варіант використання є функціональною можливістю та визначає деякий набір дій, що виконує система при діалозі з певним актором. Інформація про те, яким саме чином буде реалізована взаємодія акторів з системою навмисно приховується.

Діаграма прецедентів для розробленого програмного рішення наведена у додатку А. Як можна побачити, на ній зображено двох акторів: розробника програмної системи, в якій використовується дана бібліотека, та користувача такої системи, а також 19 варіантів використання, деякі з яких є розширеннями, включення або частковими випадками інших прецедентів.

Нижче наведено опис кожного з варіантів використання за допомогою таблиць, в яких містяться наступні поля: назва діяльності; унікальний ідентифікатор (далі ID); короткий опис; тригер; попередні умови для виконання; умови після виконання; основний потік розвитку; альтернативний потік розвитку (за наявності); можливі помилки (за наявності).

Спочатку за допомогою таблиць 6.1 – 6.12 наводиться опис діяльностей, що відповідають прецедентам стосовно роботи з об’єктами.

Перший варіант використання — «Отримати об'єкт з джерела», що є узагальненням для таких варіантів використання, як «Завантажити об'єкт з результатів запиту» та «Завантажити об'єкт з відповідного кешу». Опис діяльності для цього варіанту використання представлений у таблиці 6.1.

Варто зауважити, що дана діяльність може виконуватися двома шляхами:

1) завантаження відповідного об’єкта відбувається з результатів виконання запиту (головний потік розвитку);
2) об’єкт завантажується з кешу всередині програми, якщо відповідний кеш містить дані стосовно нього (альтернативний потік).

Таблиця 6.1 — Опис діяльності «Отримати об'єкт з джерела»

| Назва | Отримати об'єкт з джерела |
|---|---|
| ID | C_01 |
| Опис | Формується готовий для подальшої програмної взаємодії об’єкт, що відповідає сутності, яка міститься у джерелі. «Наповненість» об’єкта можна налаштувати. Будь-який об’єкт може вибиратися за унікальним ідентифікатором або за множиною інших умов, допустимих схемою об’єкта.<br>При отриманні об’єкта за ідентифікатором його завантаження можливо з кеша, якщо «кешування» відповідного типу «увімкнено». |
| Актори | Користувач, розробник |
| Тригер | Виклик методу Task LoadAsync(PropCategory category, bool reload) у IDBObject |

Продовження таблиці 6.1

| | |
|---|---|
| Післяумови | По закінченню виконання задачі:<br>— властивості об’єкта, що відповідають переданій в якості аргумента функції категорії, ініціалізовані у значення, що відповідають значенням відповідної сутності у джерелі;<br>— інші пов’язані сутності, тип взаємозв’язку яких зазначений у аргументі-категорії, в свою чергу також завантажені відповідно до даної категорії |
| Основний потік розвитку | 1. Ініційовано завантаження об’єкту, вказавши категорію для специфікації набору даних та встановивши флаг перевантаження<br>2. Виклик на завантаження даних був переданий відповідному сховищу даних (див. табл. 6.2)<br>3. Об’єкт завантажується з результатів, повернених внаслідок роботи сховища (можливе звернення до сторонніх ресурсів). Якщо у сховищі доступна сутність за таким ідентифікатором стан – LOADED, інакше – DB_NULL |
| Альтернативний потік розвитку | 1. Ініційовано отримання завантаженого раніше об’єкту, тип якого кешується<br>2. Виклик на завантаження даних був переданий відповідному кешу (див. табл. 6.3)<br>3. Якщо відповідний об’єкт в кеші має достатньо даних, то сам об’єкт ініціалізується з нього, стан – LOADED. Інакше – потік продовжує свій розвиток з пункту 2 основного потоку |
| Можливі помилки | Відсутність зв’язку з зовнішнім джерелом даних, невідповідність опису типу та сутності |

Таблиця 6.2 містить інформацію про діяльність, що відповідає варіанту використання «Завантажити об'єкт з результатів запиту». Як вже зазначалося, цей варіант використання є один з підмножини прецедентів узагальненого варіанту використання, описаного вище.

Таблиця 6.2 — Опис діяльності «Завантажити об'єкт з результатів запиту»

| | |
|---|---|
| Назва | Завантажити об'єкт з результатів запиту |
| ID | C_02 |
| Опис | Формується готовий для подальшої взаємодії у програмі об’єкт, що відповідає сутності, яка міститься у зовнішньому джерелі. «Наповненість» об’єкта та запит для його завантаження можна налаштувати за допомогою аргументів функції.<br>Налаштування SQL-запиту для реляційних БД можливе використовуючи спеціальний об’єкт IRootTableRef (див. розділ 4). Серед важливих в рамках даної специфікації характеристик, варто виділити можливість, що дозволяє вказати чи включати дані щодо пов’язаних сутностей до цього ж запиту (за допомогою join), інакше — для кожної пов’язаної сутності формується окремий запит на завантаження з налаштуваннями головної сутності. |
| Актори | Користувач, розробник |
| Тригер | Виклик функції LoadFromDB(PropCategory category, IRootTableRef tableRef, bool reload) у IDBObject |
| Післяумови | Аналогічно до зазначених у табл. 6.1 |
| Основний потік розвитку | 1. Ініційовано завантаження нового об’єкту або об’єкту з поміткою «перезавантажити» |

Продовження таблиці 6.2

| | |
|---|---|
| Основний потік розвитку | 2. Виклик на завантаження даних був переданий відповідному сховищу даних, що відповідає за взаємодію з БД<br>3. Об’єкт завантажується з результатів, повернених внаслідок звернення сховища до БД. Якщо у сховищі доступна сутність, що відповідає зазначеним у запиті умовами, стан – LOADED, інакше – DB_NULL |
| Альтернативний потік розвитку | 1. Ініційовано завантаження об’єкту, що вже був колись отриманий, передавши останнім аргументом функції false<br>2. Перезавантаження об’єктом своїх даних не відбулося, об’єкт не змінюється |
| Можливі помилки | Аналогічно до описаних у табл. 6.1 |

Далі у таблиці 6.3 наведено опис такого варіанту використання, як «Завантажити об'єкт з відповідного кешу». Цей варіант використання, подібно до попереднього, є один з можливих шляхів вже описаного прецеденту «Отримати об'єкт з зовнішнього джерела», а також прецеденту під назвою «Отримати список об'єктів з зовнішнього джерела».

Оскільки механізм діяльності з точки зору окремої сутності та списку сутностей має певні відмінності, у полях, де інформація відрізняється, буде наведено опис з необхідними уточненнями.

Таблиця 6.3 — Опис діяльності «Завантажити об'єкт з відповідного кешу»

| | |
|---|---|
| Назва | Завантажити об'єкт з відповідного кешу |
| ID | C_03 |
| Опис | Формується готовий для подальшої програмної взаємодії об’єкт, що відповідає сутності, яка зберігається у зовнішньому джерелі. |

Продовження таблиці 6.3

| | |
|---|---|
| Опис | Такий спосіб завантаження можливий за наступних умов:<br>1) кеш для відповідного типу «увімкнений»<br>2) здійснюється завантаження однієї сутності за ідентифікатором |
| Актори | Користувач, розробник |
| Тригер | Виклик методу bool LoadFromCache() у IDBObject |
| Післяумови | За умови, що тип кешується та у кеші є необхідні дані, післяумови співпадають із зазначеними у табл. 6.1. Інакше – об’єкт не змінюється |
| Основний потік розвитку | 1. Ініційовано завантаження об’єкту з зазначеним ідентифікатором<br>2. Об’єкт отримує свої дані з відповідного об’єкту в кеші<br>3. Пов’язані об’єкти завантажуються за стандартною процедурою (табл. 6.1), що, зокрема, також включає в себе можливість використання кешу<br>Стан об’єкту – LOADED |
| Альтернативний потік розвитку | 1. Ініційовано завантаження об’єкту без ідентифікатора<br>2. Завантаження об’єктом своїх даних не відбулося, об’єкт не змінюється |
| Можливі помилки | Відсутні |

Діяльність для варіанта використання «Отримати список об'єктів з джерела» здійснюється в рамках тієї ж ідеології, що й діяльність найпершого розглянутого у даному розділі прецеденту: прецедент для списку багаторазово включає в себе прецедент для одного об’єкту. Тому характеристика цієї діяльності, представлена у таблиці 6.4, подібна до характеристики суміжної діяльності «Отримати об'єкт з зовнішнього джерела», представленої у таблиці 6.1.

Також з цим прецедентом пов’язані за допомогою відношення «узагальнення» такі прецеденти, як «Завантажити список з результатів запиту» та «Завантажити список з відповідного кешу», діяльність для яких описана в таблицях 6.5 – 6.6.

Таблиця 6.4 — Опис діяльності «Отримати список об'єктів з джерела»

| Назва | Отримати список об'єктів з джерела |
|---|---|
| ID | C_04 |
| Опис | Формується готовий для подальшої взаємодії у програмі список об’єктів одного типу, що містяться у джерелі. «Наповненість» усіх об’єктів загалом можна налаштувати.<br>Список, що представляє собою вибірку, можна отримати за множиною умов, допустимих схемою відповідного типу. Завантаження списку, що містить усі об’єкти, також можливо за допомогою кеша при дотриманні певних умов (див табл. 6.6). |
| Актори | Користувач, розробник |
| Тригер | Виклик методу Task LoadAsync(PropCategory category, bool reload) у IDBList |
| Післяумови | Післяумови, зазначені у табл. 6.1, справедливі й для даної діяльності у проекції на список об’єктів |
| Основний потік розвитку | 1. Ініційовано завантаження списку об’єктів<br>2. Виклик на завантаження даних був переданий відповідному сховищу даних (див. табл. 6.2)<br>3. Усі об’єкти завантажуються з результатів, повернених внаслідок роботи сховища (можливе звернення до сторонніх ресурсів), стан списку та об’єктів – LOADED |

Продовження таблиці 6.4

| | |
|---|---|
| Альтернативний потік розвитку | 1. Ініційовано отримання списку об’єктів типу, кеш для якого містить усі сутності<br>2. Виклик на завантаження даних був переданий відповідному кешу (див. табл. 6.6)<br>3. Об’єкти для яких у кеші достатньо даних ініціалізуються з відповідників повернених кешем, для інших – потік продовжує свій розвиток з пункту 2 основного потоку<br>4. Стан списку та об’єктів – LOADED |
| Можливі помилки | Відсутність зв’язку з зовнішнім джерелом даних, невідповідність опису типу та сутності |

Далі надано опис одного зі спеціального випадку для попереднього варіанту використання — «Завантажити список з результатів запиту». Він включає в себе описаний вище варіант використання «Завантажити об’єкт з результатів запиту». Відповідна діяльність представлена у табл. 6.5. Вона за описом схожа до діяльності включеного прецеденту.

Таблиця 6.5 — Опис діяльності «Завантажити список з результатів запиту»

| | |
|---|---|
| Назва | Завантажити список з довільного запиту |
| ID | C_05 |
| Опис | Формується готовий для подальшої взаємодії у програмі об’єкт, що представляє список однотипних сутностей, які містяться у зовнішньому джерелі. «Наповненість» об’єктів зі списку та запит для його завантаження можна налаштувати за допомогою аргументів функції аналогічно, як описано у табл. 6.2. |
| Актори | Користувач, розробник |

Продовження таблиці 6.5

| | |
|---|---|
| Тригер | Виклик функції LoadFromDB(PropCategory category, IRootTableRef tableRef, bool reload) у IDBList |
| Післяумови | Аналогічно до зазначених у табл. 6.4 |
| Основний потік розвитку | 1. Ініційовано завантаження нового списку або списку з поміткою «перезавантажити»<br>2. Аналогічно кроку 2 з табл. 6.2<br>3. З повернених сховищем результатів по черзі завантажуються об’єкти до списку. Якщо у сховищі відсутні сутності за запитом, стан – DB_NULL, інакше – LOADED |
| Альтернативний потік розвитку | Аналогічно до альтернативного потоку, зазначеного у табл. 6.2, але в проекції на список об’єктів |
| Можливі помилки | Аналогічно до описаних у табл. 6.4 |

Варіант використання «Завантажити список з відповідного кешу» є ще одним спеціальним випадком варіанту «Отримати список об'єктів з джерела». Основний потік розвитку відповідної діяльності представлений у табл. 6.6. Його основний потік розвитку включає в себе завантаження з кешу для об’єкту.

Таблиця 6.6 — Опис діяльності «Завантажити список з відповідного кешу»

| | |
|---|---|
| Назва | Завантажити список з відповідного кешу |
| ID | C_06 |
| Опис | Формується готовий для подальшої програмної взаємодії об’єкт, що відповідає списку сутностей, які зберігаються у зовнішньому джерелі. |

Продовження таблиці 6.6

| | |
|---|---|
| Опис | Такий спосіб завантаження можливий за наступних умов:<br>1) кеш для відповідного типу «увімкнений»<br>2) здійснюється завантаження списку сутностей для типу, усі сутності якого були повністю завантажені до кешу раніше. |
| Актори | Користувач, розробник |
| Тригер | Виклик методу bool LoadFromCache() у IDBList |
| Післяумови | Аналогічно як зазначено у табл. 6.4 |
| Основний потік розвитку | 1. Ініційовано завантаження списку об’єктів для типу, що повністю міститься в кеші<br>2. Усі об’єкти всередині списку отримують свої дані з відповідних об’єктів у кеші<br>3. Пов’язані об’єкти завантажуються за стандартною процедурою (табл. 6.1), що, зокрема, також включає в себе можливість використання кешу<br>Стан списку та об’єктів всередині нього – LOADED |
| Альтернативний потік розвитку | 1. Ініційовано завантаження списку для типу, для якого не всі сутності якого наявні у відповідному кеші<br>2. Завантаження даних до списку не відбулося, об’єкт списку не змінюється |

Далі, у таблицях 6.7 – 6.12, будуть описані діяльності прецедентів, для яких тригером є виклик методу збереження Task Store(). Внаслідок успішного виконання методу зміни об’єктів будуть перенесені до джерела, тобто відбувається синхронізація даних від програми до джерела.

Першим таким варіантом використання є «Додати об'єкт до джерела», відповідну діяльність якого описано у таблиці 6.7.

Таблиця 6.7 — Опис діяльності «Додати об'єкт до джерела»

| | |
|---|---|
| Назва | Додати об'єкт до джерела |
| ID | C_07 |
| Опис | Збереження об’єкта, що містить нові дані, в джерелі |
| Актори | Користувач, розробник |
| Тригер | Виклик методу Task Store() у IDBObject, що має стан CHANGED (набутий після змін замість NOT_LOADED) та «нульовий» ідентифікатор |
| Післяумови | До джерела даних будуть додані дані, що містяться у об’єкті, а також нові або змінені дані пов’язаних сутностей |
| Основний потік розвитку | 1. Актор ініціював збереження нового об’єкту<br>2. Виклик на збереження даних був переданий відповідному сховищу даних<br>3. В об’єкті разом з пов’язаними об’єктами ініційовані наявні властивості, значення яких визначаються джерелом, (так звані calculated властивості), стан – LOADED |
| Альтернативний потік розвитку | 1. Актор ініціював збереження об’єкту, що не може бути доданий внаслідок прописаних реалізацією умов (наприклад, у користувача недостатньо прав)<br>2. Об’єкт не зберігається і не змінюється |
| Можливі помилки | Додатково до помилок зазначених у табл. 6.4, можливе виникнення непередбачуваних виключень на стороні джерела під час запиту на збереження даних у ньому |

У наступній таблиці надано пояснення такого варіанту використання, як «Додати список об'єктів до джерела». Діяльність для цього варіанта використання здійснюється аналогічно до діяльності для прецеденту «Додати об'єкт до джерела»

через багаторазове включення. Характеристика цієї діяльності, представлена у таблиці 6.8, подібна до характеристики суміжної діяльності, представленої вище.

Таблиця 6.8 — Опис діяльності «Додати список об'єктів до джерела»

| | |
|---|---|
| Назва | Додати список об'єктів до джерела |
| ID | C_08 |
| Опис | Збереження списку об’єктів з новими даними у джерелі |
| Актори | Користувач, розробник |
| Тригер | Виклик методу Task Store() у IDBList, що має об’єкти, у яких стан CHANGED та «нульовий» ідентифікатор |
| Післяумови | Післяумови зазначені у табл. 6.7 справедливі й для даної діяльності у проекції на список об’єктів |
| Основний потік розвитку | 1. Актор ініціював збереження списку, що містить нові об’єкти<br>2. Виклик на збереження даних був переданий відповідному сховищу даних<br>3. Для кожного нового об’єкта та пов’язаних об’єктів ініційовані наявні властивості, значення яких визначаються джерелом (так звані calculated властивості, до яких відноситься ідентифікатор), стан – LOADED. Для об’єктів зі списку, що не є новими, подальший розвиток залежить від їх стану (див. інші варіанти збереження в таблицях 6.9, 6.11)<br>Стан списку – LOADED |
| Альтернативний потік розвитку | 1. Актор ініціював збереження списку, що включає об’єкти, які не можуть бути додані внаслідок прописаних реалізацією умов (наприклад, недостатньо прав) |

Продовження таблиці 6.8

| | |
|---|---|
| Альтернативний потік розвитку | 2. Такі об’єкти не зберігається і не змінюється<br>Стан списку – LOADED |
| Можливі помилки | Аналогічно до описаних у табл. 6.7 |

У таблиці 6.9 детально розглянуто ще одну діяльність пов’язану зі збереженням даних — «Змінити об'єкт у джерелі».

Таблиця 6.9 — Опис діяльності «Змінити об'єкт у джерелі»

| | |
|---|---|
| Назва | Змінити об'єкт у джерелі |
| ID | C_09 |
| Опис | Збереження об’єкту, що містить змінені дані, у джерелі |
| Актори | Користувач, розробник |
| Тригер | Виклик методу Task Store() у IDBObject, що має стан CHANGED та «ненульовий» ідентифікатор |
| Післяумови | До джерела будуть внесені зміни відповідно до змінених даних, що містяться у об’єкті (включаючи нові або змінені дані пов’язаних сутностей) |
| Основний потік розвитку | 1. Актор ініціював збереження існуючого об’єкту<br>2. Виклик на збереження даних був переданий відповідному сховищу даних<br>3. Визначені calculated властивості об’єкта та пов’язаних з ним об’єктів встановленні в значення, повернені джерелом після внесення змін, стан – LOADED |
| Альтернативний потік розвитку | 1. Актор ініціював збереження об’єкту, що не може бути збережений внаслідок прописаних реалізацією умов (наприклад, недостатньо прав)<br>2. Об’єкт не зберігається і не змінюється |
| Можливі помилки | Аналогічно до описаних у табл. 6.7 |

Далі розглядається варіант використання «Змінити список об'єктів у джерелі». Відповідна для нього діяльність здійснюється за допомогою багаторазового включення діяльності аналогічного прецеденту для одного об’єкта — «Змінити об'єкт у джерелі». Це пояснює подібність опису діяльності з таблиці 6.10 до опису суміжної діяльності (табл. 6.9).

Таблиця 6.10 — Опис діяльності «Змінити список об'єктів у джерелі»

| | |
|---|---|
| Назва | Змінити список об'єктів у джерелі |
| ID | C_10 |
| Опис | Збереження списку об’єктів, що містять змінені дані, у джерелі |
| Актори | Користувач, розробник |
| Тригер | Виклик методу Task Store() у IDBList, що має стан CHANGED та об’єкти, у яких стан CHANGED та «ненульовий» ідентифікатор |
| Післяумови | До джерела будуть внесені зміни відповідно до змінених або доданих нових даних, що містяться у об’єкті (включаючи нові або змінені дані пов’язаних сутностей) |
| Основний потік розвитку | 1. Актор ініціював збереження списку, що містить існуючі об’єкти<br>2. Виклик на збереження даних був переданий відповідному сховищу даних<br>3. Визначені calculated властивості змінених об’єктів та пов’язаних з ним об’єктів встановленні в значення, повернені джерелом після внесення змін, стан – LOADED. Для об’єктів зі списку, що є доданими або видаленими, подальший розвиток залежить від їх стану (див. інші варіанти збереження 6.7, 6.11)<br>Стан списку – LOADED |

Продовження таблиці 6.10

| | |
|---|---|
| Альтернативний потік розвитку | 1. Актор ініціював збереження списку, що містить об’єкти, які не можуть бути збережені внаслідок прописаних реалізацією умов (наприклад, недостатньо прав)<br>2. Такі об’єкти не зберігається і не змінюється. Потік розвитку для інших об’єктів залежить від їх стану (див. інші варіанти збереження 6.7, 6.11). Стан списку – LOADED |
| Можливі помилки | Аналогічно до описаних у табл. 6.7 |

Діяльність «Видалити об'єкт у джерелі» використовується для видалення даних певної сутності та дочірніх сутностей з джерела. Вона описана у таблиці 6.11.

Таблиця 6.11 — Опис діяльності «Видалити об'єкт у джерелі»

| | |
|---|---|
| Назва | Видалити об'єкт у джерелі |
| ID | C_11 |
| Опис | Збереження видаленого об’єкту, тобто видалення даних, пов’язаних з сутністю, з джерела |
| Актори | Користувач, розробник |
| Тригер | Виклик методу Task Store() у IDBObject, що має стан DELETED та «ненульовий» ідентифікатор |
| Післяумови | Дані щодо видалених сутностей були видалені з джерела |
| Основний потік розвитку | 1. Завантажений об’єкт помічається на видалення за допомогою функції MarkDeleted()<br>2. Здійснюється виклик на збереження даних відповідному сховищу даних |

Продовження таблиці 6.11

| | |
|---|---|
| Основний потік розвитку | 3. Властивості об’єкта та його дочірніх об’єктів ініціалізовані у значення за замовчуванням, стан цих об’єктів – DB_NULL |
| Можливі помилки | Аналогічно до описаних у табл. 6.7 |

У таблиці 6.12 визначена діяльність «Видалити список об'єктів у джерелі». Вона схожа за призначенням до описаної у попередній таблиці діяльності, але має відмінний принцип роботи.

Таблиця 6.12 — Опис діяльності «Видалити список об'єктів у джерелі»

| | |
|---|---|
| Назва | Видалити список об'єктів у джерелі |
| ID | C_12 |
| Опис | Збереження списку, що містить видалені об’єкти, тобто видалення цих даних з джерела |
| Актори | Користувач, розробник |
| Тригер | Виклик функції Store() у IDBList, що має стан CHANGED та об’єкти, у яких стан DELETED та «ненульовий» ідентифікатор |
| Післяумови | Дані щодо видалених сутностей були видалені з джерела |
| Основний потік розвитку | 1. Усі об’єкти у листі помічаються на видалення шляхом виклику для кожного методу Task MarkDeleted()<br>2. Здійснюється виклик на збереження даних списку відповідному сховищу даних<br>3. Властивості кожного об’єкта та його дочірніх об’єктів ініціалізовані у значення за замовчуванням, стан цих об’єктів та списку – DB_NULL |
| Можливі помилки | Аналогічно до описаних у табл. 6.7 |

На цьому опис варіантів використання, пов’язаних з інтерфейсом роботи з об’єктами, завершено.

Далі, у таблицях 6.13 – 6.16 наведено опис діяльностей для прецедентів, що відносяться до роботи з SQL-джерелами. За побудованою діаграмою таких прецедентів у системі передбачено 4: один самостійний — «Автоматизована побудова select-запиту» — та два інші варіанти виконання команд, що можуть бути розширені прецедентом «Отримати дані» за умови, що ця команда є командою на вибір даних з джерела.

«Автоматизована побудова select-запиту» – найважливіша діяльність доступна у межах програмного засобу для роботи з SQL-джерелами. Вона описана у таблиці 6.13.

Таблиця 6.13 — Опис діяльності «Автоматизована побудова select-запиту»

| | |
|---|---|
| Назва | Автоматизована побудова select-запиту |
| ID | C_13 |
| Опис | Формується текстовий sql-запит на вибірку даних з таблиць сховища. До запита може бути включена не тільки основна таблиця, що відповідає логічній сутності у джерелі, для якої створено програмний об’єкт, а й інші пов’язані з нею таблиці, якщо вони включені до схеми об’єкта.<br>У запиті можна налаштувати набір даних, що повертається як результат, та умови, згідно з якими буде відбуватися відбір даних. Також є можливість задати порядок сортування, обмежити чисельність обраних записів (limit) та пропустити певну кількість (skip, offset). |
| Актори | Розробник |
| Тригер | Виклик методу string BuildSQL() у ISQLTextElement |

Продовження таблиці 6.13

| | |
|---|---|
| Післяумови | Отримано параметризований рядок, що представляє запит на вибірку даних, який відповідає стандарту мови SQL. Параметри у запиті використовуються для заміни реальних значень, перед назвою параметра у рядку ставиться спеціальний символ – @. |
| Основний потік розвитку | 1. Створюється об’єкт типу IQueryBuilder для побудови запиту<br>2. Викликається метод Select об’єкту IQueryBuilder для певної програмної сутності<br>3. За допомогою поверненого об’єкта типу IRootTableRef розробник налаштовує та розширює запит за потреби<br>4. Цей об’єкт використовується для завантаження об’єкта з джерела:<br>а) формується команда до стороннього SQL-джерела на отримання даних (див. табл. 6.14)<br>б) об’єкт ініціюється з результатів виконання команди |
| Альтернативний потік розвитку | 1. Аналогічно кроку 1 основного потоку<br>2. Аналогічно кроку 2 основного потоку<br>3. Аналогічно кроку 3 основного потоку<br>4. Цей об’єкт використовується для формування команди до стороннього SQL-джерела (див. табл. 6.14)<br>5. Дані, що запитувалися, доступні через відповідний програмний інтерфейс для поверненого SQL-сервером результату виконання команди (див. табл. 6.14) |
| Можливі помилки | Відсутні |

У таблиці 6.14 міститься опис діяльності «Виконати довільну команду» для однойменного варіанту використання. Цей прецедент представляє базову

функціональність, що необхідна при роботі зі стороннім SQL-джерелом. У рамках цієї функції розробнику надається можливість звернення до БД з будь-яким запитом. Якщо надіслана команда представляє собою select-запит, то даний варіант використання може бути розширений прецедентом «Отримати дані» для отримання даних, що повертаються БД внаслідок виконання.

Таблиця 6.14 — Опис діяльності «Виконати довільну команду»

| | |
|---|---|
| Назва | Виконати довільну команду |
| ID | C_14 |
| Опис | У рамках з'єднання з SQL-джерелом формується та надсилається SQL-команда |
| Актори | Розробник |
| Тригер | Виклик одного з перевантажених (overloaded) методів для виконання команди у об'єкта IDBGCommand.<br>При select-запитах використовується один з перевантажених методів IDataCursorSource ExecuteCursor, що дозволяє отримати усі дані, повернені джерелом як результат виконання. Для інших запитів варто викликати один з варіантів методу int Execute, що повертає кількість рядків, що зазнали вплив внаслідок виконання команди. |
| Післяумови | Отримано результат виконання SQL-команди |
| Основний потік розвитку | 1. За допомогою об'єкта IDBGConnection розробник викликає метод IDBGCommand CreateCommand() у IDBGate доступного через властивість Gate<br>2. Отримана команда налаштовується за допомогою ініціалізації наступних властивостей: IDBGConnection Connection — об'єкт зв'язку, string SqlCommand — текст команди |

Продовження таблиці 6.14

| | |
|---|---|
| Основний потік розвитку | 3. Використовуючи функцію Open() зв’язку, здійснюється підключення до БД (можливо використання вже існуючого з’єднання)<br>4. Розробник передає команду на виконання до джерела за допомогою методу-тригера Execute. При необхідності передає до методу параметри<br>5. Розробник отримує результат виконання команди<br>6. За допомогою функції Release() здійснюється відключення від БД (якщо на етапі 3 було взяте існуюче з’єднання, то відключення відбудеться, коли звільняться усі його використання) |
| Альтернативний потік розвитку | 1-3. Аналогічно відповідним крокам основного потоку<br>4. Розробник передає команду на виконання до джерела за допомогою методу-тригера ExecuteCursor. При необхідності передає до методу параметри<br>5. За допомогою поверненого функцією результату розробник отримує дані (див. табл. 6.16)<br>6. Аналогічно до кроку 6 основного потоку |
| Можливі помилки | Непередбачувані виключення на стороні джерела під час виконання команди |

Ще однією обов’язковою функціональністю для бібліотеки, що взаємодіє з реляційними БД, є можливість виконувати ряд команд в рамках однієї транзакції.

З цією метою у системі розроблена відповідна функціональність, що представлена на діаграмі прецедентів як варіант використання під назвою «Виконати команди у транзакції». Діяльність для цього прецеденту представлена у таблиці 6.15.

Таблиця 6.15 — Опис діяльності «Виконати команди у транзакції»

| | |
|---|---|
| Назва | Виконання команд у транзакції |
| ID | C_15 |
| Опис | В рамках однієї транзакції формується та надсилається множина SQL-команд до реляційної БД |
| Актори | Розробник |
| Тригер | Для внесення змін – виклик функції Commit() у об’єкта IDBGTransactionConnection, для скасування – Rollback() |
| Післяумови | Усі зміни, що відбулися внаслідок виконання команд у транзакції, були застосовані. Якщо під час виконання будь-якої команди у транзакції відбулася помилка, то зміни внаслідок попередніх команд нівелюються, подальші команди к транзакції виконані не будуть |
| Основний потік розвитку | 1. Розробник викликає метод IDBGTransactionConnection StartTransaction() об’єкта IDBGConnection<br>2-3. Формується команда аналогічно крокам 1-2 основного потоку з табл. 6.15 (оскільки отриманий об’єкт є спадкоємцем IDBGConnection)<br>4-5. Команда передається до джерела даних та отримується результат аналогічно крокам 4-5 основного чи альтернативного потоку розвитку<br>6. У разі необхідності виконання інших команд, кроки 2-5 повторюються<br>7. Здійснюється виклик функції-тригера Commit() для підтвердження внесення змін усіма командами |
| Альтернативний потік розвитку | 1-6. Аналогічно відповідним крокам основного потоку<br>7. Розробник викликає функцію-тригер Rollback() для відміни внесення змін усіма командами |
| Можливі помилки | Аналогічно до описаних у табл. 6.15 |

«Отримати дані» — прецедент, що розширює прецеденти «Виконання довільної команди» та «Виконання команд у транзакції». Як вже було згадано, точкою розширення для цього прецеденту є команда на вибірку даних (у термінології SQL вони носять назву select-запитів). Опис відповідної діяльності надано у таблиці 6.16.

Таблиця 6.16 — Опис діяльності «Отримати дані»

| | |
|---|---|
| Назва | Отримати дані |
| ID | C_16 |
| Опис | Внаслідок виконання команди на вибірку даних здійснюється обробка повернутих SQL-джерелом даних |
| Актори | Розробник |
| Тригер | Виклик будь-якого з перевантажених (overloaded) методів IDataCursorSource ExecuteCursor у об’єкта IDBGCommand |
| Післяумови | З результату виконання команди були отримані значення, які запитувалися у select-запиті |
| Основний потік розвитку | 1. Використовуючи повернутий функцією-тригером об’єкт IDataCursorSource (представляє контейнер для записів з даними), актор викликає функцію Open()<br>2. Для перегляду результатів викликається метод IDataCursor GetCursor(), що повертає курсор для пересування по рядкам даних<br>3. За допомогою методу CursorResult Current(out IBaseDataRecord dr) здійснюється перевірка наявності даних у поточному рядку, на який вказує курсор, та отримується рядок IBaseDataRecord |

Продовження таблиці 6.16

| | |
|---|---|
| Основний потік розвитку | 4. У випадку, коли метод повернув значення RECORD, з отриманого об’єкту IBaseDataRecord (далі – запис) по черзі отримуються дані зі списку вибірки за їх ім’ям у запиті, використовуючи метод object GetValue(string name)<br>5. Для переходу до наступного запису даних актор використовує такий метод IDataCursorSource, як CursorResult Next(out IBaseDataRecord dr)<br>6. Для обробки даних нових записів кроки 4-5 повторюються<br>7. Для курсору викликається функція Dispose, коли досягнуто кінця (по завершенню обробки усіх записів з курсора контейнером повертається значення EOF)<br>8. Розробник завершує роботу з IDataCursorSource викликаючи функцію Close() |
| Альтернативний потік розвитку | 1-3. Аналогічно відповідним крокам основного потоку<br>4. Метод повернув значення EOF, що означає відсутність у джерелі даних за надісланим запитом<br>5-6. Аналогічно крокам 7-8 основного потоку |
| Можливі помилки | Відсутні |

Зазначені нижче варіанти використання стосуються розробленого формату передачі даних. На діаграмі прецедентів зображено два основних варіанти використання: «Серіалізація об'єкта» та «Десеріалізація об'єкта».

Першим розглядається прецедент «Серіалізація об'єкта». Він використовується для формування представлення існуючого об’єкта у відповідності до формату. Відповідна для нього діяльність описана у таблиці 6.17.

Таблиця 6.17 — Опис діяльності «Серіалізація об'єкта»

| | |
|---|---|
| Назва | Серіалізація об'єкта |
| ID | C_17 |
| Опис | До переданого контейнера записуються пов’язані з об’єктом дані в спеціальному представленні. При серіалізації списку в контейнер по черзі записуються дані кожного об’єкту, шо входить до його складу. Перелік даних об’єкта для запису регулюється відповідним параметром функції-тригера |
| Актори | Розробник |
| Тригер | Виклик методу Task WriteInCursorSource (IMutableDataCursorSource cursorSource, DBECursorWriteBinding binding) у IDBEntity |
| Післяумови | До джерела записані затребувані дані об’єкта(ів) |
| Основний потік розвитку | 1. Розробник створює об’єкт типу StdDataSource, що представляє собою контейнер, де зберігаються дані<br>2. Актор формує об’єкт DBECursorWriteBinding, що містить так звані прив’язки – набір правил відповідно до яких дані об’єкта записуються у джерело<br>Так, розробник має заповнити структуру даних InnerBindings. В основі цього словника – шляхи до властивостей об’єкта-сутності, що містять дані, а для опису даних вкладених об’єктів сутностей словник передбачає збереження рекурсивних прив’язок у значенні. Також можна задати прив’язки за допомогою прапорів FullInner та FullRecursive<br>3. Для існуючого завантаженого об’єкта розробник ініціює серіалізацію в контейнер, передаючи сам контейнер та прив’язки |

Продовження таблиці 6.17

| | |
|---|---|
| Альтернативний потік розвитку | 1-2. Аналогічно відповідним крокам основного потоку<br>3. Для завантаженого списку об’єктів одного типу викликається функція-тригер для серіалізації, в якості аргументів їй аналогічно до основного потоку розвитку передається контейнер та прив’язки |
| Можливі помилки | Відсутні |

Далі наведено опис діяльності для варіанту використання «Десеріалізація об'єкта». Даний прецедент представляє протилежну за напрямком дію до серіалізації: об’єкт створюється відповідно до свого представлення у форматі. Відповідна діяльність наведена у таблиці 6.18.

Таблиця 6.18 — Опис діяльності «Десеріалізація об'єкта»

| | |
|---|---|
| Назва | Десеріалізація об'єкта |
| ID | C_18 |
| Опис | З даних, що містяться у переданому контейнері, ініціалізується об’єкт або список об’єктів. Правила, що регулюють дані для завантаження в об’єкт, передаються відповідним параметром функції |
| Актори | Розробник |
| Тригер | Виклик методу Task LoadFromCursor (IDataCursor cursor, DBECursorReadBinding bindng) у IDBEntity |
| Післяумови | В об’єкт заповнені усі затребувані властивості даними, взятими з контейнера |
| Основний потік розвитку | 1. Розробник отримує курсор до контейнера, що вказує на запис даних після серіалізації |

Продовження таблиці 6.18

| | |
|---|---|
| Основний потік розвитку | 2. Формується об’єкт прив’язки DBECursorReadBinding, в якому визначається: PropCategory – категорія даних для «заповнення» з джерела; флаг UseObjNames, що вказує чи дані у курсорі доступні за іменами властивостей, інакше — за внутрішніми іменами у джерелі даних<br>3. Актор викликає функцію-тригер для новоствореного об’єкта, що відповідає певній сутності, передаючи туди курсор та прив’язку. По завершенню виконання стан об’єкта – LOADED |
| Альтернативний потік розвитку | Аналогічний розвиток можливий під час десеріалізації списку об’єктів |
| Можливі помилки | Відсутні |

Останній варіант використання — Отримати внутрішньосистемне runtime представлення об'єктів — дозволяє отримати спеціальне розширене відображення структури об’єкта. Дана діяльність представлена у таблиці 6.19 і може використовуватися розробником для динамічного ефективного «розбору» об’єкта за схемою свого типу під час виконання застосунку.

Таблиця 6.19 — Опис діяльності «Отримати внутрішньосистемне runtime представлення об'єктів»

| | |
|---|---|
| Назва | Отримати внутрішньосистемне runtime представлення об'єктів |
| ID | C_19 |
| Опис | Використовуючи цю діяльність, актор може отримати метадані об’єкта, за допомогою яких визначається схема внутрішнього формування об’єкта з даних відповідної йому сутності у джерелі, включно з пов’язаними |

Продовження таблиці 6.19

| | |
|---|---|
| Актори | Розробник |
| Тригер | Виклик методу IDBPropInfo[] GetPropsInfo (PropCategory group) у IDBObjectSchemaDB |
| Післяумови | Отримано метадані, що описують властивості об’єкта, які мають відношення до даних з джерела |
| Основний потік розвитку | 1. Створюється об’єкт типу IDBObjectSchemaDB<br>2. Для формування схеми певної сутності викликається функція LoadSchema(Type type)<br>3. Розробник здійснює виклик методу-тригера для сформованого об’єкта схеми |
| Можливі помилки | Відсутні |

На основі описаних варіантів використання та сформованих у розділі 1, пункт 1.2 функціональних вимог було побудовано таблицю зв’язків між ними — таблиця 6.20. У даній таблиці FR — функціональна вимога, а UC — варіант використання.

Таблиця 6.20 — Requirements Traceability Matrix для варіантів використання

| UC<br>FR | C01 | C02 | C03 | C04 | C05 | C06 | C07 | C08 | C09 | C10 |
|---|---|---|---|---|---|---|---|---|---|---|
| Представлення даних | X | X | X | X | | | | | | |
| Внесення змін до даних | | | | | X | X | | | | |
| Додання нових даних | | | | | | | X | X | X | X |

Продовження таблиці 6.20

| UC<br>FR | C11 | C12 | C13 | C14 | C15 | C16 | C17 | C18 | C19 |
|---|---|---|---|---|---|---|---|---|---|
| Видалення даних | X | X | | | | | | | |
| Автоматизація select-запитів | | | X | | | | | | |
| Виконання довільної SQL-команди | | | | X | | | | | |
| Виконання SQL-команд у транзакції | | | | | X | | | | |
| Перегляд результатів виконання команди БД | | | | | | X | | | |
| Обмін даними між компонентами | | | | | | | X | X | |
| Метадані у runtime | | | | | | | | | X |

З таблиці 6.20 можна побачити, що кожна функціональна вимога покрита принаймні одним варіантом використання, що свідчить про функціональну завершеність розробленого програмного продукту. А відсутність варіантів використання, яким не відповідає жодна функціональна вимога, говорить про те, що розроблене рішення не містить незатребуваних у завданні можливостей.

## 6.2 Висновки

Отже, у даному розділі було наведено детальний опис діаграми прецедентів для бібліотеки, розробленої на основі запропонованої моделі. Так, на ній визначені 2 актори — розробник та користувач — та передбачені варіанти використання — прецеденти.

На діаграмі визначено 19 прецедентів, що логічно розподілені по функціональним блокам відповідно до призначення. Деякі з них є включеннями чи розширення інших. Для користувача системи, в межах якої використовується бібліотека, доступні лише 7 варіантів використання, для розробника – усі, при цьому він як актор на діаграмі є спеціальним випадком користувача і може взаємодіяти з тою самою множиною варіантів використання.

У межах даного розділу для кожного варіанту використання було надано опис відповідної діяльності у системі, а в кінці розділу побудовано узагальнюючу таблицю, що надає інформацію щодо покриття функціональних вимог до бібліотеки реалізованими у ній варіантами використання.

Виходячи з таблиці, можна зробити висновок, що розроблене рішення повністю задовольняє висунутим до нього функціональним вимогам.

# ВИСНОВКИ

У ході виконання магістерської дисертації було розроблено модель уніфікованої бібліотеки, що має забезпечувати функціональність для представлення та обробки складнозв'язаних даних в об'єктно-орієнтованій парадигмі.

На базі запропонованої у роботі моделі можуть бути розроблені бібліотеки для більшості платформ, технологій та мов програмування, що передбачають роботу в об'єктно-орієнтованій парадигмі. Практична цінність та можливість створення бібліотеки, побудованої за цією моделлю, була перевірена за допомогою тестової реалізації. Результати роботи були впроваджені у діяльність КМП «Інформпроект».

Бібліотека на основі моделі була розроблена та налагоджена на платформі .NET за допомогою мови програмування C#. Для тестування використовувався фреймворк NUnit та БД PostgreSQL в якості джерела даних.

На базі проведених випробувань було проаналізовано ефективність окремих розроблених у бібліотеці засобів. Згідно з отриманих результатів, використання розробленого формату даних дозволяє зменшити обсяг даних при передачі множини об'єктів в середньому на 45%. Також у розробленому програмному рішення реалізовано спосіб завантаження даних, що дозволяє зменшити загальний час завантаження складнозв'язаних об'єктів з реляційних баз даних.

Відповідне програмне рішення складається з наступних модулів:

— модуль для побудови середовища взаємодії між компонентами системи та надання абстракцій над інфраструктурою платформи виконання;
— модуль обробки та представлення даних в ОО парадигмі;
— модуль для взаємодії з реляційними базами даними;
— модуль, що містить реалізації конкретних об'єктів даних, виконаних з використанням засобів, наданих бібліотекою.

Результати досліджень було оприлюднено на міжнародній науково-практичній конференції «WORLD SCIENCE: PROBLEMS, PROSPECTS AND INNOVATIONS», а також у вітчизняних фахових журналах «Технічні науки та технології» та «Математичні машини і системи».

# ПЕРЕЛІК ПОСИЛАНЬ


1. Літвінова Н. О., Цитовцева А. С., Сопов О. О. Способи захисту баз даних від sql ін’єкцій для забезпечення якості програмного забезпечення // World science: problems, prospects and innovations. Abstracts of the 7th International scientific and practical conference. – Perfect Publishing. Toronto, Canada. 2021. — 458-464 с. Режим доступу: https://sci-conf.com.ua/vii-mezhdunarodnaya-nauchno-prakticheskaya-konferentsiya-world-science-problems-prospects-and-innovations-24-26-marta-2021-goda-toronto-kanada-arhiv.
2. Літвінова Н. О., Альперт М. І., Погульський А. М. Підвищення ефективності обміну даними сутностей у реляційному представленні та їх обробки // Технічні науки та технології, 2021, № 1. — 81-86 с.
3. Альперт С. І., Альперт М. І., Катін П. Ю., Літвінова Н. О. Програмно-апаратна інфраструктура наземної автономної платформи з елементами штучного інтелекту // Математичні машини і системи, 2021, № 1. — 24-31 с.
4. 2019 Database Trends – SQL vs. NoSQL, Top Databases, Single vs. Multiple Database Use [електронний ресурс]: https://scalegrid.io/blog/2019-database-trends-sql-vs-nosql-top-databases-single-vs-multiple-database-use.
5. Гурлі Д. HTTP: The Definitive Guide / Девід Гурлі, Брайан Тотті. – O'Reilly Media, 2002.
6. Вигерс К. Разработка требований к программному обеспечению: пер с англ. / Вигерс Карл, Битти Джой. — М.: Русская Редакция, 2004. — 314 с.
7. Вієгунаратне І., Фернандез Г. The Three-Tier Application Architecture. In: Distributed Applications Engineering. Practitioner Series. / Індраїт Вієгунаратне, Джордж Фернандез — Springer, London, 1998. — 41-78 с.
8. Кларк Дж. SQL Injection Attacks and Defense / Джастін Кларк. – Syngress Date, 2009.
9. Хальде Дж. Basics of SQL Injection Analysis, Detection and Prevention / Джагдін Хальде. – LAP LAMBERT Academic Publishing, 2014.

10. .NET | Free. Cross-platform. Open Source [електронний ресурс] — Режим доступу: https://dotnet.microsoft.com.
11. Richter J. CLR via C# / Jeffrey Richter. – Microsoft Press, 2015.
12. NUint [електронний ресурс] — Режим доступу: https://nunit.org.
13. PostgreSQL: The World's Most Advanced Open Source Relational Database [електронний ресурс]: https://www.postgresql.org.
14. DB-Engines Ranking – popularity ranking of database management systems [електронний ресурс] — Режим доступу: https://db-engines.com/en/ranking.
15. Npgsql – .NET Access to PostgreSQ [електронний ресурс] — Режим доступу: https://www.npgsql.org.
16. Json.NET – Newtonsoft [електронний ресурс] — Режим доступу: https://www.newtonsoft.com/json.
17. Західноукраїнський Національний Університет. Опорний конспект лекцій [електронний ресурс] / Західноукраїнський Національний Університет — Режим доступу: http://dspace.wunu.edu.ua/jspui/bitstream/316497/24194/1/опорний%20конспект%20лекцій.pdf
18. Microsoft. Monitor [електронний ресурс]: Microsoft Docs. – Режим доступа: https://docs.microsoft.com/en-us/dotnet/api/system.threading.monitor.
19. Кириллов В. В. Структуризованный язык запросов (SQL): учебн. пособ.: [електронний ресурс] / В. В. Кириллов, Г. Ю. Громов. – СПб: Санкт-Петерб. госуд. техн. универ., каф. выч. техники, 1998. – Режим доступу: http://www.citforum.ru/database/sql_kg.
20. Харрингтон Дж. Л. Проектирование реляционных баз данных: пер. с англ. / Джен Харрингтон Дж. Л. — М.: Лори, 2006. — 232 с.
21. Чкалов А. П. Базы данных: от проектирования до разработки приложений / А. П. Чкалов — СПб.: БХВ-Петербург, 2003. – 384 с.
22. Fine Code Coverage Tool [електронний ресурс] — Режим доступу: https://github.com/FortuneN/FineCodeCoverage.

23. Леоненков А. В. Самоучитель UML 2 / Александр Леоненков. — БХВ-Петербург, 2007. — 76-82 с.
24. Фаулер М. UML. Основы: пер. с англ./ Мартін Фаулер, Скотт Кендалл — СПб: Символ-Плюс, 2002. — 192 с.