БАКАЛАВРСЬКА РОБОТА

БР. ІП - 37.00.00.000 ПЗ

Група ІП-22-2

Колядко Станіслав

2026

Івано-Франківський національний технічний університет нафти і газу

Факультет інформаційних технологій

Кафедра інженерії програмного забезпечення

Колядко Станіслав Павлович

(прізвище, ім'я, по батькові)

УДК 004.4
(індекс)

БАКАЛАВРСЬКА РОБОТА

Розроблення прототипу генератора програмного коду для багаторівневих архітектур програмного забезпечення

(назва роботи)

Інженерія програмного забезпечення

(назва освітньої програми)

121 - Інженерія програмного забезпечення

(шифр і назва спеціальності)

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

Здобувач освітнього рівня Колядко С.П.
(підпис, ініціали та прізвище здобувача)

Науковий керівник Бандура Вікторія Валеріївна, к.т.н., доцент
(підпис, прізвище, ім'я, по батькові, науковий ступінь, вчене звання керівника)

Допущено до захисту
Завідувач кафедри

доц. Бандура В.В.
(посада) (підпис) (дата) (ініціали та прізвище)

Івано-Франківськ – 2026

Івано-Франківський національний технічний університет нафти і газу

Інститут, факультет інформаційних технологій

Кафедра інженерії програмного забезпечення

Ступінь вищої освіти бакалавр

Спеціальність 121 – Інженерія програмного забезпечення

ЗАТВЕРДЖУЮ:
Зав. кафедрою **ІПЗ**
доц. В.В. Бандура
“ ____ ” _____2026 р.

ЗАВДАННЯ

НА БАКАЛАВРСЬКУ РОБОТУ СТУДЕНТОВІ

Колядко Станіславу Павловичу

(прізвище, ім’я, по-батькові)

1. Тема проекту (роботи) “ Розроблення прототипу генератора програмного коду для багаторівневих архітектур програмного забезпечення ”

керівник проекту (роботи) Бандура В.В., доцент , к.т.н.

затверджені наказом закладу вищої освіти від “ 21 ” квітня 2026р. № 179 /7

2. Строк подання студентом проекту (роботи) 10 червня 2026 р.

3. Вихідні дані до проекту (роботи) Результати і матеріали отримані під час проходження переддипломної практики

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

1. Особливості побудови генераторів програмного коду для багаторівневих архітектур
2. Порівняльний аналіз сучасних програмних рішень для автоматизованої генерації коду
3. Моделі та алгоритмічне забезпечення побудови прототипу генератора програмного коду
4. Логічне моделювання реляційної структури бази даних для роботи генератора коду
5. Проектування та реалізація генератора програмного коду для багаторівневих архітектур

5. Перелік графічного матеріалу (з точним зазначенням обов’язкових креслень)

1. Структурна декомпозиція рівнів в контексті побудови генератора коду (рис. 1.1, ст.14)
2. Чинники доцільності впровадження інструментів автоматичної генерації коду (рис.1.2, ст.18)
3. Візуалізація ключових переваг автоматизації коду (рис. 1.3, ст.19)
4. Головна сторінка платформи генерації коду Yeoman (рис. 1.4, ст.22)
5. Платформа класу Low-code - OutSystems (рис. 1.5, ст.24)

6. Консультанти розділів проекту (роботи)

Розділ	Консультант	Підпис, дата	
		Завдання видав	Завдання прийняв

7. Дата видачі завдання **21 квітня 2026 р.**

Керівник _____
(підпис)

Завдання прийняв до виконання_____
(підпис)

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів дипломного проекту (роботи)	Строк виконання етапів проекту	Примітка
1	Особливості побудови генераторів програмного коду для багаторівневих архітектур	03.05.2026	виконано
2	Порівняльний аналіз сучасних програмних рішень для автоматизованої генерації коду	14.05.2026	виконано
3	Моделі та алгоритмічне забезпечення побудови прототипу генератора програмного коду	21.05.2026	виконано
4	Логічне моделювання реляційної структури бази даних для роботи генератора коду	28.05.2026	виконано
5	Проектування та реалізація генератора програмного коду для багаторівневих архітектур	03.06.2026	виконано
6	Оформлення пояснювальної записки бакалаврської роботи завідувачем кафедри	10.06.2026	виконано

Студент – дипломник _____ Колядко С.П.
(підпис)

Керівник роботи _____Бандура В.В.
(підпис)

АНОТАЦІЯ

Бакалаврська робота містить 75 сторінок, 25 рисунків, список використаних джерел із 39 найменуваннями.

Мета роботи: розроблення прототипу генератора програмного коду для автоматизованого створення компонентів багаторівневих архітектур програмного забезпечення на основі аналізу структури реляційних баз даних.

Об'єкт дослідження - процеси проєктування та розробки програмного забезпечення на основі багаторівневих архітектур.

Предмет дослідження - методи, моделі та алгоритми автоматизованої генерації програмного коду для багаторівневих архітектур програмного забезпечення.

В першому розділі обґрунтовано доцільність використання багаторівневих архітектур і визначено необхідність автоматизації генерації програмного коду.

В другому розділі розроблено моделі та алгоритми автоматизованої генерації програмного коду на основі метаданих бази даних.

В третьому розділі реалізовано прототип генератора коду та підтверджено його ефективність на практиці.

Висновок: розроблено прототип генератора програмного коду забезпечує ефективну автоматизацію створення компонентів багаторівневих систем та підвищує продуктивність процесу розробки програмного забезпечення.

КЛЮЧОВІ СЛОВА: БАГАТОРІВНЕВА АРХІТЕКТУРА, ГЕНЕРАЦІЯ КОДУ, АВТОМАТИЗАЦІЯ РОЗРОБКИ, CRUD-ОПЕРАЦІЇ, МЕТАДАНІ, РЕЛЯЦІЙНА БАЗА ДАНИХ, ПРОГРАМНЕ ЗАБЕЗПЕЧЕННЯ, АЛГОРИТМИ ГЕНЕРАЦІЇ, ВАЛІДАЦІЯ ДАНИХ.

ANNOTATION

The bachelor's thesis contains 75 pages, 25 figures, a list of used sources with 39 names.

Purpose of the work: development of a prototype of a program code generator for automated creation of components of multi-level software architectures based on analysis of the structure of relational databases.

The object of the study is the processes of design and development of software based on multi-level architectures.

The subject of the study is methods, models and algorithms of automated generation of program code for multi-level software architectures.

The first section substantiates the feasibility of using multi-level architectures and determines the need for automation of program code generation.

The second section develops models and algorithmic automated generation of program code based on database metadata.

In the third section, a prototype of a code generator is implemented and its effectiveness is confirmed in practice.

Conclusion: the developed prototype of a program code generator provides effective automation of the creation of components of multi-level systems and the efficiency of the software development process.

KEYWORDS: MULTI-TIER ARCHITECTURE, CODE GENERATION, DEVELOPMENT AUTOMATION, RAW OPERATIONS, METADATA, RELATIONAL DATABASE, SECURITY PROGRAM, GENERATION ALGORITHMS, DATA VALIDATION.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ СКОРОЧЕНЬ

BLL – Business Logic Layer

CLI – Command Line Interface

DAL – Data Access Layer

DBMS – Database Management System

DSL – Domain-Specific Language

FK – Foreign Key

FM – Foundation Model

LLM – Large Language Model

MDD – Model-Driven Development

OWASP – Open Web Application Security Project

PK – Primary Key

PL – Presentation Layer

SoC – Separation of Concerns

ВСТУП

Сучасний етап розвитку інформаційних технологій характеризується стрімким зростанням складності програмних систем, що зумовлює необхідність застосування ефективних архітектурних підходів до їх проєктування та реалізації. Одним із найбільш поширених підходів є використання багаторівневих архітектур, які забезпечують розмежування функціональних обов'язків між окремими компонентами системи, сприяють підвищенню гнучкості, масштабованості та зручності супроводу програмного забезпечення. У таких системах особливої ваги набуває питання ефективної реалізації типових функціональних компонентів, зокрема CRUD-операцій, які становлять значну частину прикладної логіки.

Традиційні підходи до розробки передбачають ручне створення значного обсягу повторюваного коду, що призводить до збільшення часу розробки, підвищення ризику помилок та ускладнення процесу підтримки програмних продуктів. У зв'язку з цим особливо актуальним стає використання засобів автоматизованої генерації програмного коду, які дозволяють формалізувати процес створення типових компонентів і підвищити ефективність розробки.

У роботі досліджуються підходи до побудови генераторів програмного коду для багаторівневих архітектур, розглядаються існуючі інструменти та визначаються їхні обмеження. На основі проведеного аналізу запропоновано концепцію прототипу генератора, що використовує метадані реляційної бази даних для автоматизованого синтезу програмних компонентів. Особливу увагу приділено розробці алгоритмів генерації коду, забезпеченню типобезпечності та реалізації багаторівневої валідації даних.

Практичним результатом роботи є створення прототипу програмного засобу, який забезпечує автоматизоване формування компонентів багаторівневої архітектури, що підтверджує доцільність застосування

запропонованого підходу та його ефективність у процесі розробки програмного забезпечення.

Актуальність роботи

Актуальність теми бакалаврської роботи зумовлена стрімким розвитком інформаційних технологій та зростанням вимог до якості, масштабованості й швидкості розробки програмного забезпечення. У сучасних умовах цифрової трансформації підприємства потребують створення складних інформаційних систем, здатних ефективно обробляти значні обсяги даних, забезпечувати надійність функціонування та швидко адаптуватися до змін бізнес-вимог. Це, у свою чергу, обумовлює необхідність застосування структурованих підходів до проєктування програмних систем, серед яких багаторівнева архітектура посідає провідне місце.

Використання багаторівневих архітектур дозволяє розділити функціональність системи на окремі логічні рівні, що сприяє підвищенню гнучкості, модульності та повторного використання програмних компонентів. Проте практична реалізація таких архітектур пов’язана зі значними витратами часу на розробку типових модулів, зокрема компонентів доступу до даних, бізнес-логіки та інтерфейсів взаємодії з користувачем. Значна частина цього коду має шаблонний характер і повторюється у різних проєктах, що призводить до неефективного використання ресурсів розробників.

Крім того, ручне створення однотипного програмного коду підвищує ймовірність виникнення помилок, ускладнює тестування та супровід системи, а також знижує загальну якість програмного продукту. У контексті сучасних підходів до розробки програмного забезпечення, таких як Agile та DevOps, особливої актуальності набуває автоматизація рутинних процесів, що дозволяє скоротити цикл розробки та підвищити продуктивність команд.

Існуючі інструменти автоматизованої генерації коду, включаючи фреймворк-залежні засоби скаффолдингу, low-code/no-code платформи та сучасні AI-орієнтовані рішення, хоча й забезпечують часткове вирішення

зазначених проблем, однак часто мають обмеження щодо гнучкості, масштабованості або прив’язки до конкретних технологій. Це ускладнює їх використання в індивідуальних або високоспеціалізованих проєктах, де необхідна більша адаптивність та контроль над згенерованим кодом.

Додатковим чинником актуальності є потреба у формалізації процесу генерації коду на основі метаданих, зокрема структури реляційних баз даних, що дозволяє створювати узгоджені та типобезпечні програмні компоненти. Такий підхід сприяє підвищенню узгодженості між різними рівнями системи, спрощує підтримку та забезпечує можливість швидкого масштабування програмного забезпечення. Таким чином, розроблення прототипу генератора програмного коду для багаторівневих архітектур, який поєднує гнучкість, універсальність та ефективність, є актуальним науково-практичним завданням, вирішення якого сприятиме оптимізації процесів розробки програмного забезпечення, підвищенню його якості та зниженню витрат ресурсів на створення типових компонентів.

Мета роботи

Метою роботи є розроблення прототипу генератора програмного коду для автоматизованого створення компонентів багаторівневих архітектур програмного забезпечення на основі аналізу структури реляційних баз даних.

Об’єкт дослідження - процеси проєктування та розробки програмного забезпечення на основі багаторівневих архітектур.

Предмет дослідження - методи, моделі та алгоритми автоматизованої генерації програмного коду для багаторівневих архітектур програмного забезпечення.

Завдання дослідження

1. Провести аналіз особливостей побудови багаторівневих архітектур програмного забезпечення.

2. Дослідити існуючі підходи та інструменти генерації програмного коду.

3. Обґрунтувати доцільність автоматизації створення CRUD-компонентів.

4. Розробити концептуальну модель генератора програмного коду.

5. Створити алгоритми генерації програмних компонентів на основі метаданих бази даних.

6. Реалізувати прототип генератора програмного коду.

7. Оцінити ефективність та практичну придатність розробленого рішення.

Методи дослідження

У роботі використано методи системного аналізу для дослідження архітектур програмного забезпечення, методи моделювання для побудови структурних і логічних моделей системи, алгоритмічні методи для розробки процедур генерації коду, а також методи об’єктно-орієнтованого проєктування та гнучкої розробки програмного забезпечення.

Наукова новизна

Наукова новизна роботи полягає у розробці узагальненого підходу до автоматизованої генерації програмного коду для багаторівневих архітектур на основі аналізу метаданих реляційних баз даних, що включає алгоритми інтроспекції, мапінгу типів даних та шаблонного синтезу компонентів із підтримкою багаторівневої валідації.

Практичне застосування

Практичне значення роботи полягає у створенні прототипу генератора програмного коду, який може бути використаний для автоматизації розробки інформаційних систем, скорочення часу створення програмних компонентів та підвищення якості програмного забезпечення.

Бакалаврська робота містить 75 сторінок, 25 рисунків, 3 розділи та список використаних джерел із 39 найменуваннями.

РОЗДІЛ 1. ОСОБЛИВОСТІ ПОБУДОВИ ГЕНЕРАТОРІВ ПРОГРАМНОГО КОДУ ДЛЯ БАГАТОРІВНЕВИХ АРХІТЕКТУР

1.1. Теоретичне обґрунтування розробки інформаційних систем на основі багаторівневої архітектури

Концептуальний підхід до проектування сучасних програмних комплексів ґрунтується на застосуванні багаторівневої (n-рівневої) архітектури. Даний архітектурний патерн передбачає декомпозицію системи на логічно відокремлені сегменти (шари), кожен з яких наділений специфічною зоною відповідальності та взаємодіє з іншими рівнями через чітко визначені інтерфейси.

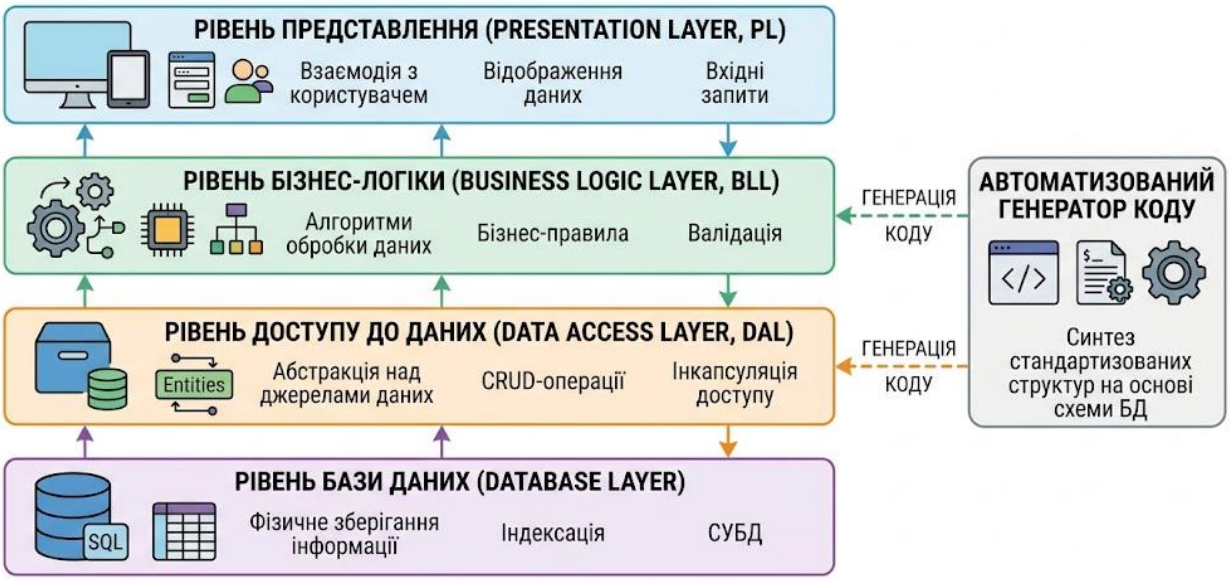

Рисунок 1.1 – Структурна декомпозиція рівнів в контексті побудови генератора коду

1.1.1. Структурна декомпозиція рівнів системи

Типова архітектурна модель досліджуваної системи включає наступні ієрархічні рівні:

- рівень представлення (Presentation Layer, PL) виступає верхнім

ешелоном системи, забезпечуючи реалізацію людино-машинного інтерфейсу. Його функціональне призначення полягає у візуалізації даних, отриманих від нижчих рівнів, та обробці вхідних запитів користувача для їх подальшої трансляції в систему.

- рівень бізнес-логіки (Business Logic Layer, BLL) є центральним компонентом системи, де зосереджені алгоритми обробки даних та бізнес-правила предметної області. На цьому рівні здійснюється валідація вхідних параметрів, координація потоків даних та забезпечення цілісності логічних операцій.

- рівень доступу до даних (Data Access Layer, DAL) виконує роль абстракції над механізмами персистентності. Він інкапсулює логіку взаємодії з джерелами даних (СУБД), забезпечуючи виконання операцій створення, зчитування, оновлення та видалення (CRUD) без розкриття деталей реалізації сховища для вищих рівнів.

- Рівень бази даних (Database Layer) - нижній рівень ієрархії, представлений системою управління базами даних. Він забезпечує фізичне зберігання, індексацію та підтримку цілісності структурованої інформації.

1.1.2. Переваги та принципи архітектурного розмежування

Впровадження багаторівневої моделі дозволяє досягти високих показників модульності та масштабованості. Завдяки слабкій зв’язності (loose coupling) компонентів стає можливою модифікація або повна заміна технологічного стеку одного рівня (наприклад, зміна типу СУБД) без необхідності рефакторингу всієї системи. Це суттєво підвищує життєздатність програмного продукту та спрощує його технічну підтримку.

Ключовим аспектом даної роботи є мінімізація рутинних операцій при проектуванні рівнів DAL та BLL. Автоматизований генератор коду, що пропонується в межах роботи, спрямований на синтез стандартизованих програмних структур на основі аналізу існуючої схеми бази даних.

Мета розробки інструментарію:

- Створення програмного модуля для автоматичного формування шаблонів класів та методів, що відповідають структурі сутностей бази даних.

- Забезпечення уніфікації та послідовності вихідного коду (code consistency) на всіх етапах розробки.

- Зниження ймовірності виникнення синтаксичних та логічних помилок при реалізації механізмів передачі даних.

Рівень бізнес-логіки проектується з використанням принципів об’єктно-орієнтованого програмування, що дозволяє інкапсулювати складні операції в абстрактні класи та методи. Такий підхід забезпечує ефективну синхронізацію між усіма рівнями архітектури, оптимізує транзакційні витрати при взаємодії з базою даних та сприяє швидкому впровадженню нових бізнес-інваріантів у систему.

1.2. Обґрунтування доцільності автоматизації розробки CRUD-інтерфейсів у багаторівневих архітектурах

Процес проектування та реалізації сучасних інформаційних систем на базі n-рівневих архітектур супроводжується значними обсягами рутинних операцій, зокрема при створенні механізмів взаємодії з персистентними сховищами даних. Автоматизована генерація програмного коду для операцій CRUD (Create, Read, Update, Delete) є критично важливим аспектом оптимізації життєвого циклу розробки ПЗ (SDLC). Доцільність впровадження таких інструментів обумовлена низкою технологічних та економічних чинників.

1. Оптимізація часових та ресурсних витрат

Автоматизація формування програмних артефактів дозволяє досягти:

- Мінімізації обсягів ручного кодування, оскільки використання CASE-засобів для генерації шаблонного SQL-коду та об'єктно-реляційного відображення (ORM) дозволяє вивільнити інтелектуальні ресурси розробників для вирішення складних логічних задач.

- Скорочення показника Time-to-Market, бо швидка генерація базових рівнів системи (DAL та BLL) прискорює фазу імплементації, що є визначальним фактором у конкурентному середовищі розробки програмних продуктів.

2. Забезпечення детермінованості та якості кодової бази

Якість програмного забезпечення безпосередньо залежить від ступеня уніфікації його компонентів:

- Автоматизовані інструменти гарантують ідентичність структури коду для всіх сутностей системи, що виключає варіативність у реалізації однотипних завдань.

- Автоматизація нівелює ризики виникнення синтаксичних та логічних помилок, що притаманні ручному написанню повторюваних фрагментів коду, підвищуючи загальну надійність системи.

3. Адаптивність та зниження технічного боргу

Підтримка є однією з найдорожчих фаз життєвого циклу ІС:

- Будь-які зміни в концептуальній схемі бази даних (наприклад, додавання атрибутів або зміна типів даних) автоматично транслюються у відповідні зміни в коді DAL та BLL. Це забезпечує цілісність системи без необхідності ретельного рефакторингу вручну.

- Легкість оновлення згенерованого коду дозволяє системі еволюціонувати разом із бізнес-вимогами, мінімізуючи накопичення технічного боргу.

4. Уніфікація та повторне використання

Впровадження стандартизованих шаблонів проектування сприяє:

- Створення єдиних стандартів для CRUD-операцій дозволяє використовувати розроблені алгоритми та шаблони в суміжних проектах, що веде до накопичення корпоративної бази знань та готових модулів.

- Уніфікована структура коду полегшує процес входження нових спеціалістів у проект, оскільки логіка взаємодії компонентів є передбачуваною та задокументованою на рівні генератора.

5. Рівень абстракції та декомпозиція логіки

Архітектурна чистота досягається через розмежування відповідальності:

- Згенеровані шари абстрагують бізнес-логіку від специфіки конкретної СУБД, реалізуючи патерни «Репозиторій» (Repository) або «Об’єкт доступу до даних» (DAO).

- Відокремлення механізмів персистентності від функціонального ядра системи дозволяє проводити незалежне тестування та модернізацію окремих модулів без порушення стабільності всієї архітектури.

Рисунок 1.2 – Чинники доцільності впровадження інструментів автоматичної генерації коду

Таким чином, автоматизована генерація коду в межах n-рівневої архітектури не лише підвищує продуктивність праці розробників, а й виступає гарантом високої якості, модульності та адаптивності інформаційної системи. Це створює надійний фундамент для впровадження принципів швидкої розробки та забезпечує довгострокову життєздатність програмного продукту в умовах динамічної зміни вимог.

1.3. Аналіз цілей та переваг автоматизованої генерації коду в контексті розробки багаторівневих систем

Процес проектування сучасних інформаційних систем, орієнтованих на роботу з великими масивами даних, вимагає мінімізації рутинних операцій на етапі імплементації. Автоматизована генерація коду виступає як інструмент інтенсифікації розробки n-рівневих застосунків, що дозволяє не лише прискорити створення програмного продукту, а й забезпечити високу якість його архітектурної реалізації [5].

Рисунок 1.3 – Візуалізація ключових переваг автоматизації коду

Впровадження механізмів автоматичної генерації програмних артефактів для операцій CRUD (Create, Read, Update, Delete) дозволяє вирішити низку критичних завдань:

1. Оптимізація часового ресурсу та зниження трудомісткості

Автоматизація дозволяє радикально скоротити обсяг написання так званого «шаблонного коду». Це вивільняє ресурси розробників для проектування високорівневої бізнес-логіки, оскільки генерація низькорівневих SQL-запитів та об'єктно-реляційних відображень (ORM)

делегується спеціалізованим CASE-засобам.

2. Детермінованість та верифікація якості кодової бази

Використання генераторів мінімізує ризики, пов'язані з «людським фактором». Автоматизований підхід гарантує синтаксичну правильність та логічну послідовність коду, що нівелює ймовірність виникнення дефектів на етапі інтеграції рівнів системи. Це забезпечує сувору відповідність реалізованого коду встановленим архітектурним специфікаціям.

3. Забезпечення еволюційної придатності до обслуговування

У процесі життєвого циклу ПЗ схема бази даних часто зазнає змін. Автоматизована генерація забезпечує механізм швидкої ресинхронізації між рівнем зберігання даних та прикладним програмним інтерфейсом [6]. Це дозволяє підтримувати актуальність коду без необхідності трудомісткого ручного рефакторингу при кожній модифікації схеми БД.

4. Уніфікація програмних шаблонів та повторне використання

Формування стандартизованих паттернів проектування сприяє створенню модульної структури, де компоненти доступу до даних можуть бути легко адаптовані або перенесені в інші проекти. Це закладає фундамент для створення бібліотек типових рішень у межах організації або конкретної технологічної платформи.

5. Абстрагування рівнів взаємодії з персистентним сховищем

Генерація коду дозволяє створити високий рівень абстракції, ізолюючи розробника від складнощів прямого маніпулювання даними на рівні СУБД. Це спрощує архітектуру системи, оскільки рівень бізнес-логіки взаємодіє з даними через чітко визначені інтерфейси, не заглиблюючись у специфіку реалізації запитів.

Підсумовуючи, можна стверджувати, що призначення автоматизованої генерації коду виходить за межі простого прискорення розробки. Це комплексний метод забезпечення надійності, масштабованості та технологічної стійкості інформаційних систем [7]. Використання автоматизованих засобів у розробці n-рівневих структур створює передумови

для формування якісного програмного забезпечення, здатного до гнучкої адаптації в умовах динамічних функціональних вимог.

1.4. Порівняльний аналіз сучасних програмних рішень для автоматизованої генерації коду

Для визначення новизни розроблюваного інструментарію необхідно провести критичний аналіз існуючих платформ та методологій автоматизованої генерації коду. Сучасний ринок CASE-засобів (Computer-Aided Software Engineering) пропонує кілька концептуальних підходів до вирішення цієї задачі, кожен з яких має свої архітектурні особливості та обмеження.

1.4.1. Фреймворк-залежні засоби скаффолдингу

Найбільш поширеним підходом є вбудовані механізми популярних фреймворків, такі як Entity Framework Core Power Tools (.NET), Spring Roo (Java) або Ruby on Rails Scaffolding.

Дані інструменти автоматично створюють класи доступу до даних (DAL) та базові контролери на основі існуючої схеми БД.

Spring Roo демонструє підхід, де інтерфейс командного рядка стає потужним інструментом проектування. Він дозволяє розробнику залишатися в межах термінології предметної області (Entities, Fields, Controllers), делегуючи системі низькорівневу реалізацію зв’язків між рівнями архітектури. Хоча сьогодні більшість цих функцій перебрав на себе Spring Boot, архітектурні принципи Roo щодо чистоти коду та автоматизації CRUD-операцій залишаються фундаментальними для сучасних генераторів коду

Переваги: Висока інтеграція з екосистемою розробки та дотримання стандартів конкретної платформи.

Недоліки: Тісна прив’язка до конкретної технології (Vendor lock-in) та обмежені можливості кастомізації логіки n-рівневого розмежування поза

межами стандартних шаблонів фреймворку.

1.4.2. Платформи повноциклової генерації

До цієї категорії належать такі інструменти, як JHipster та Yeoman.

Вони дозволяють генерувати цілісну екосистему застосунку — від клієнтської частини до серверної логіки та конфігурацій бази даних.

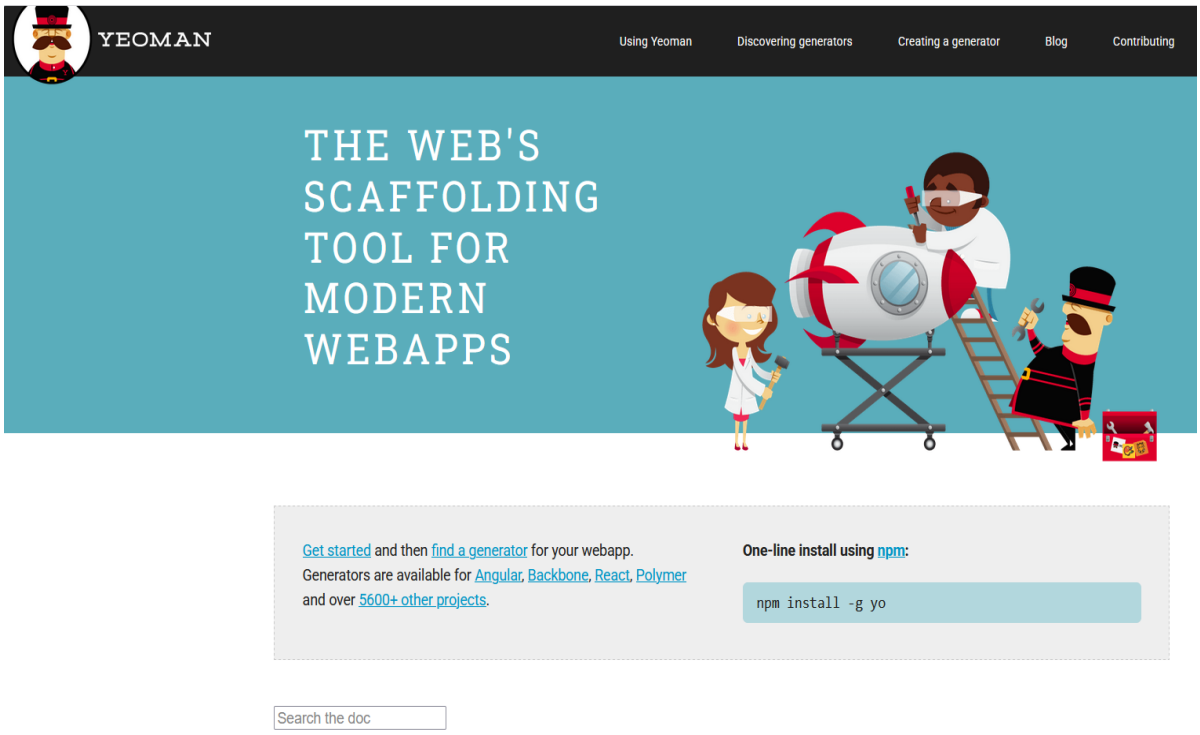

Рисунок 1.4 – Головна сторінка платформи генерації коду Yeoman

Yeoman представляє собою не просто інструментарій, а цілісну екосистему для автоматизації початкового етапу розробки сучасних веб-застосунків. Якщо Spring Roo фокусується на Java-стеку, то Yeoman був створений як універсальний додаток для фронтенд- та фулстек-розробників, що дозволяє за лічені хвилини розгорнути структуру проєкту, яка відповідає найсучаснішим галузевим стандартам.

Головна перевага Yeoman — це децентралізація. Платформа сама по собі є порожньою оболонкою; вся потужність прихована у тисячах «генераторів», створених спільнотою. Існують генератори для Angular, React,

Vue, мікросервісів на Node.js і навіть для розширень Chrome. Це дозволяє командам створювати власні корпоративні шаблони, забезпечуючи ідентичну архітектуру в усіх внутрішніх проєктах компанії.

Хоча сьогодні багато фреймворків мають власні інструменти (Angular CLI, Create React App), Yeoman залишається незамінним для створення кастомних архітектурних шаблонів. У великих організаціях він використовується для генерації мікросервісів, де важливо, щоб кожен новий сервіс мав однакову структуру логування, доступу до даних та конфігурації безпеки.

Переваги: Забезпечення високого рівня типізації та використання кращих практик побудови мікросервісних та монолітних архітектур.

Недоліки: Надмірна складність генерованого коду, що ускладнює подальшу підтримку системи для вузькоспеціалізованих бізнес-задач.

1.4.3. Платформи низького рівня кодування (Low-code/No-code)

Системи на кшталт OutSystems, Mendix або Microsoft Power Apps пропонують візуальні інтерфейси для створення n-рівневих систем.

Особливістю даних платформ є, той факт, що генерація програмного коду прихована від розробника, а взаємодія з БД відбувається через графічні абстракції.

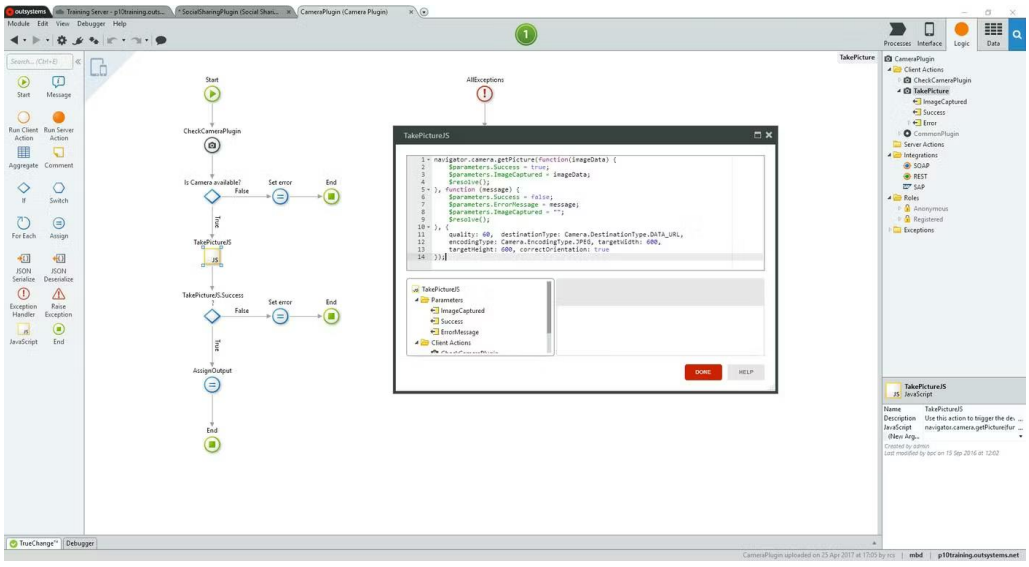

Рисунок 1.5 – Платформа класу Low-code - OutSystems

Платформа OutSystems є одним із найбільш яскравих представників сучасного покоління систем класу Low-code, що реалізують парадигму Model-Driven Development (MDD). Якщо інструменти на кшталт Spring Roo або Yeoman орієнтовані на професійних розробників, які працюють безпосередньо з вихідним кодом, то OutSystems пропонує вищий рівень абстракції, перетворюючи процес створення n-рівневих систем на візуальне моделювання архітектурних компонентів.

Одним із головних досягнень платформи є механізм автоматизованого розгортання та перевірки цілісності системи. При натисканні кнопки публікації система запускає складний багатоетапний процес:

- Аналіз впливу - перевірка всіх залежностей між рівнями. Якщо зміна в базі даних порушує роботу логічного модуля, система заблокує публікацію до усунення конфлікту.

- Компіляція - візуальні моделі транслюються у стандартний високооптимізований код (.NET або Java) та SQL-скрипти.

- Автоматизація DevOps - система автоматично оновлює схеми БД, переносить програмні артефакти на сервери застосунків та налаштовує конфігурації безпеки.

Сучасні версії OutSystems впроваджують когнітивні сервіси (наприклад, AI Mentor System), які виконують роль автоматизованого рецензента коду. Штучний інтелект аналізує візуальні моделі на предмет порушення архітектурних стандартів, проблем із безпекою або низької продуктивності запитів. Це дозволяє суттєво знизити «технічний борг», який зазвичай накопичується у великих корпоративних системах.

ереваги: Максимальна швидкість виходу на ринок.

Недоліки: Ефект «чорної скриньки», де розробник не має прямого контролю над якістю та оптимізацією згенерованого коду, що неприпустимо для високонавантажених або специфічних наукоємних систем.

1.4.4. Когнітивні сервіси та AI-генератори

Сучасні інструменти на базі великих мовних моделей (LLM), такі як GitHub Copilot або Amazon CodeWhisperer.

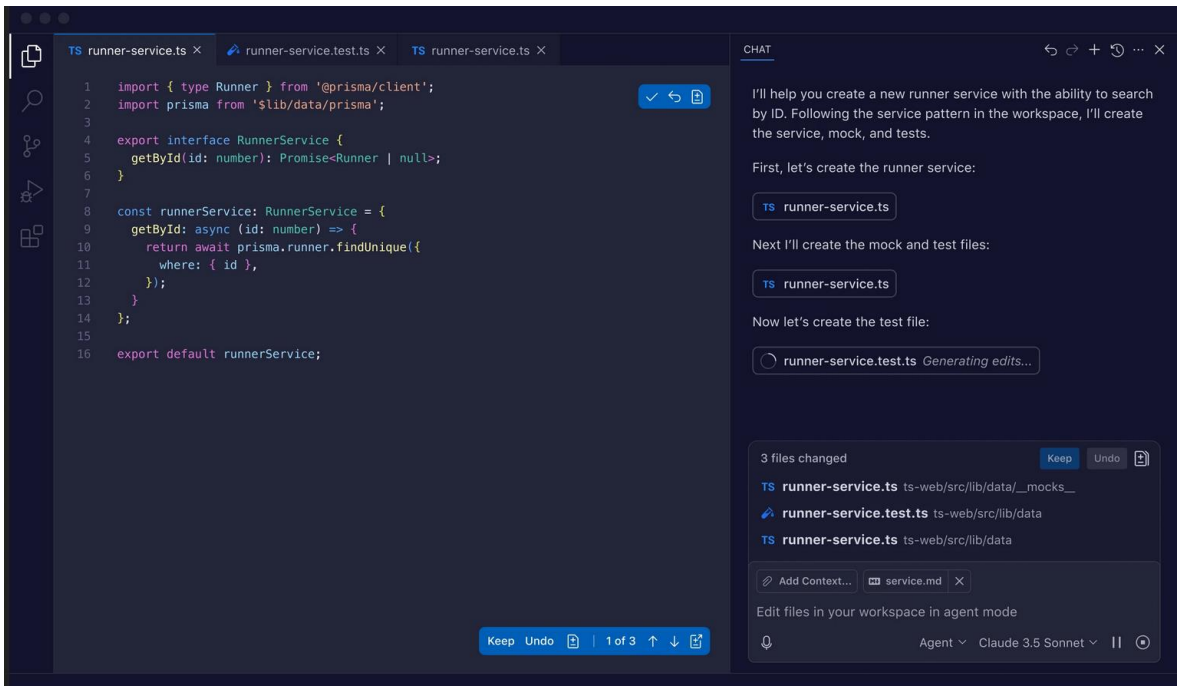

Рисунок 1.6 – Приклад використання GitHub Copilot

Особливістю цього класу платформ є контекстуальна генерація фрагментів коду на основі природної мови або існуючих сигнатур методів.

GitHub Copilot базується на глибоких нейронних мережах архітектури Transformer, спеціалізованих на обробці природних мов та мов програмування. Фундаментом системи є моделі сімейства OpenAI (зокрема, еволюційні наступники GPT-4 та Codex), які пройшли етап навчання на масивах відкритих вихідних кодів із репозиторіїв GitHub.

Процес функціонування Copilot можна представити як ітераційний цикл контекстуального виведення (Inference):

- Система зчитує не лише поточний рядок коду, а й семантичне оточення: назви функцій, коментарі природною мовою, сусідні файли проєкту та імпортовані бібліотеки.

- На основі вхідного вектора модель генерує найбільш імовірні послідовності токенів, що відповідають синтаксису обраної мови

програмування (Python, JavaScript, TypeScript, C#, Ruby, Go та ін.).

- Алгоритми враховують вибір розробника (прийняття або відхилення пропозиції), що дозволяє системі підлаштовуватися під специфічний стиль кодування конкретної команди чи проєкту.

GitHub Copilot виходить за межі простого автодоповнення, пропонуючи комплексне вирішення інженерних завдань:

- Можливість генерації складних алгоритмів на основі текстового опису (документування функцій стає тригером для їх реалізації).

- Автоматизація модульного тестування, швидке створення тест-кейсів на основі аналізу логіки методів.

- Система здатна інтерпретувати застарілий або складний код, пропонуючи шляхи його оптимізації відповідно до сучасних стандартів (наприклад, перехід на асинхронні моделі виконання).

GitHub Copilot представляє собою інтелектуальний інтерфейс між людським інтелектом та обчислювальною потужністю мовних моделей.

Впровадження GitHub Copilot спричиняє парадигмальний зсув у продуктивності, проте створює низку викликів:

- дослідження вказують на можливість генерації моделей, що містять вразливості (наприклад, вразливі до SQL-ін’єкцій фрагменти), якщо такі патерни були присутні в навчальній вибірці. Для нівелювання цього ризику в сучасні версії Copilot інтегровано фільтри безпеки на основі статичного аналізу.

- питання інтелектуальної власності на згенерований код залишається дискусійним. GitHub впровадив механізми Reference Tracker, які попереджають розробника, якщо пропозиція ідентична існуючому коду з відкритим кодом, вказуючи тип ліцензії.

- існує гіпотеза про можливе зниження когнітивного навантаження на розробників-початківців, що може призвести до послаблення навичок глибокого розуміння алгоритмів при надмірному покладанні на ШІ.

Станом на 2026 рік він еволюціонував у повноцінне середовище GitHub

Copilot X/Q, яке підтримує голосове керування, автоматичне виправлення помилок на етапі компіляції та глибоку інтеграцію з CI/CD процесами. Це дозволяє стверджувати, що інструмент перетворився з допоміжного скрипта на фундаментальний елемент сучасної екосистеми програмної інженерії.

Технологічний фундамент Amazon CodeWhisperer базується на використанні великих мовних моделей та фундаментальних моделей, що пройшли навчання на мільярдах рядків вихідного коду, включаючи як відкриті репозиторії, так і внутрішню кодову базу Amazon. Це дозволяє системі не просто дописувати окремі слова, а розуміти семантичний контекст і пропонувати цілісні логічні блоки, функції та класи.

Інтерфейс CodeWhisperer реалізований як розширення для популярних інтегрованих середовищ розробки (VS Code, IntelliJ IDEA, AWS Cloud9). На відміну від класичних генераторів коду, що працюють за жорсткими шаблонами, підхід Amazon базується на імовірнісному прогнозуванні наступного фрагмента коду на основі попереднього контексту та коментарів природною мовою.

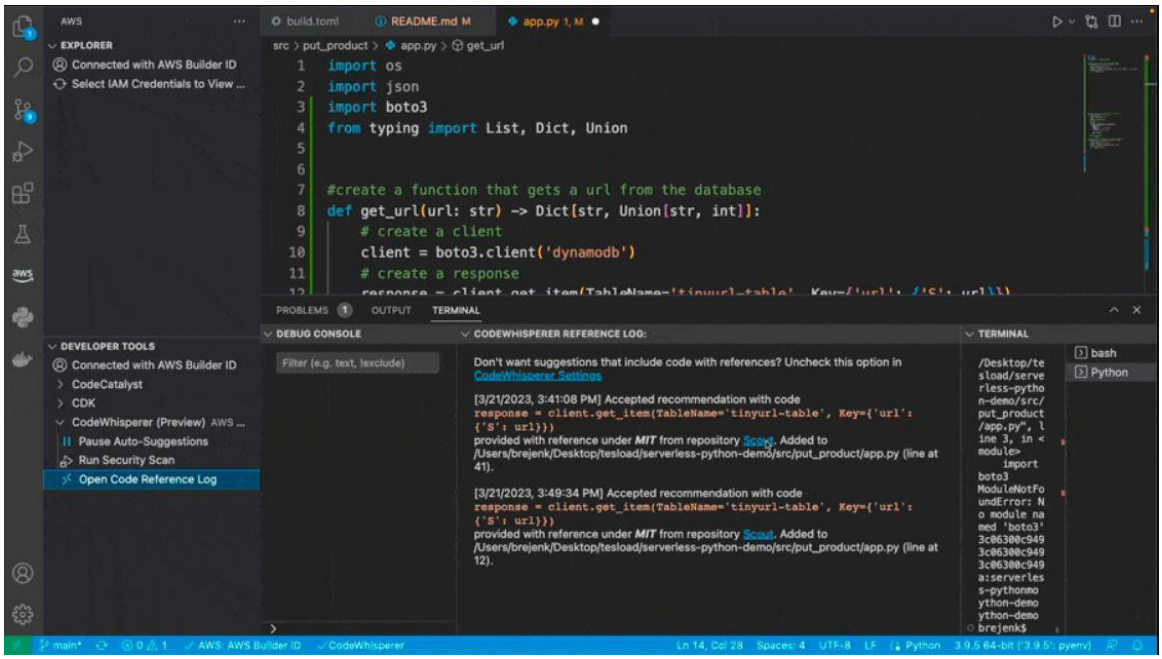

Рисунок 1.7 – Використання Amazon CodeWhisperer

Платформа містить вбудовані алгоритми сканування на наявність вразливостей (відповідно до стандартів OWASP). Вона здатна ідентифікувати потенційно небезпечні патерни у згенерованому або написаному вручну коді та пропонувати негайні виправлення (remediations), що суттєво підвищує надійність системи ще на етапі розробки.

У контексті розробки багатокомпонентних систем, Amazon CodeWhisperer демонструє найвищий рівень гнучкості серед аналізованих інструментів. Він не нав’язує жорстку структуру проєкту, як Yeoman або Spring Roo, але забезпечує інтелектуальну підтримку на кожному етапі написання коду.

Проте, використання таких сервісів вимагає від розробника високої кваліфікації для верифікації результатів, оскільки ймовірнісна природа LLM може призводити до появи «галюцинацій» — синтаксично правильного, але логічно помилкового коду. Саме тому ідеальна стратегія розробки полягає у поєднанні детермінованих генераторів (для структури CRUD та рівнів доступу до даних) з AI-асистентами (для написання унікальної бізнес-логіки та оптимізації коду).

Переваги: Висока гнучкість та підтримка багатьох мов програмування.

Недоліки: Недетермінованість результату. Відсутність гарантії архітектурної цілісності в межах усього n-рівневого проєкту. AI може запропонувати вірний синтаксично код, який, проте, порушує принципи інкапсуляції між DAL та BLL.

1.4.5. Порівняльна характеристика платформ за ключовими критеріями

Нижче наведено порівняльну таблицю аналізованих рішень у контексті розробки n-рівневих систем.

Таблиця 1.1

Порівняння аналізованих рішень

Критерій порівняння	Скаффолдинг (EF/Spring)	Full-stack (JHipster)	Low-code платформи	AI-генератори
Ступінь автоматизації	Середній	Високий	Максимальний	Високий (фокус на CRUD)
Контроль над архітектурою	Високий	Середній	Низький	Повний
Гнучкість шаблонів	Обмежена	Середня	Низька	Висока (кастомізована)
Прозорість коду	Висока	Середня	Відсутня	Висока

Проведений аналіз демонструє, що попри наявність потужних інструментів, існує дефіцит рішень, які б поєднували в собі прозорість згенерованого коду з суворим дотриманням n-рівневої архітектури без надмірного ускладнення системи.

Більшість існуючих платформ або занадто жорстко прив’язані до екосистеми фреймворку, або створюють «монолітний» результат, який важко адаптувати під специфічні бізнес-правила. Це обґрунтовує необхідність розробки спеціалізованого генератора, який фокусується на детермінованому створенні рівнів DAL та BLL, забезпечуючи розробнику повний контроль над структурою та можливістю подальшого масштабування системи.

1.5 Висновки по розділу

У першому розділі проведено теоретичний аналіз принципів побудови багаторівневих архітектур програмного забезпечення, що дозволило обґрунтувати доцільність їх використання у сучасних інформаційних системах. Встановлено, що структурна декомпозиція на рівні представлення, бізнес-логіки та доступу до даних забезпечує підвищення гнучкості, масштабованості та підтримуваності програмних продуктів. Досліджено проблему високої трудомісткості розробки типових CRUD-операцій, яка є характерною для багаторівневих систем і суттєво впливає на тривалість

розробки. Обґрунтовано необхідність автоматизації процесів створення програмного коду як засобу зниження витрат часу та мінімізації людських помилок. Проведено порівняльний аналіз сучасних інструментів генерації коду, що дозволив виявити їхні переваги та обмеження в контексті універсальності та адаптивності. На основі отриманих результатів сформульовано вимоги до розроблення прототипу генератора коду, орієнтованого на підтримку багаторівневих архітектур.

РОЗДІЛ 2. МОДЕЛІ ТА АЛГОРИТМІЧНЕ ЗАБЕЗПЕЧЕННЯ ПОБУДОВИ ПРОТОТИПУ ГЕНЕРАТОРА ПРОГРАМНОГО КОДУ

2.1. Методологія автоматизації розробки компонентів багаторівневих систем на основі структурного аналізу баз даних

Сучасний стан розробки програмного забезпечення характеризується постійним зростанням складності архітектурних рішень та вимог до надійності систем. Одним із найбільш ефективних підходів до вирішення цих завдань є застосування багаторівневої архітектури — шаблону проєктування, що базується на принципі розподілу відповідальності (Separation of Concerns). У межах даної роботи пропонується програмний інструментарій для автоматизованого синтезу програмних артефактів, що забезпечує уніфікацію розробки на всіх етапах життєвого циклу інформаційної системи.

2.1.1. Концептуальна модель архітектурної декомпозиції

Досліджувана модель передбачає поділ системи на логічно ізольовані сегменти, що мінімізує зв'язність та підвищує когезію модулів. Структура типової системи включає наступні ієрархічні рівні:

- Рівень представлення - забезпечує термінальну взаємодію з кінцевим користувачем, обробку вхідних подій та візуалізацію даних.

- Рівень бізнес-логіки - акумулює ключові алгоритми предметної області, здійснює валідацію транзакцій та забезпечує дотримання бізнес-правил.

Рівень доступу до даних виконує роль абстракції над механізмами персистентності, інкапсулюючи логіку виконання запитів до бази даних.

Рівень бази даних - нижній рівень ієрархії, що відповідає за фізичне зберігання, індексацію та цілісність структурованої інформації.

2.1.2. Функціональні характеристики та алгоритмічна база

інструментарію

Пропонований проєкт спрямований на розробку спеціалізованого генератора коду, який використовує метадані схеми бази даних як вхідні параметри для автоматичного формування вихідного коду. Ключовою інновацією є перехід від ручного написання повторюваних структур до детермінованого синтезу об’єктно-орієнтованих моделей.

Методологічні засади функціонування генератора:

- Програма виконує рефлексивний аналіз структури таблиць, зв'язків та типів даних СУБД для побудови внутрішньої моделі сутностей.

- На основі отриманих даних формуються класи (Entities) та методи для реалізації CRUD-операцій (Create, Read, Update, Delete).

- Рівень BLL проєктується на принципах інкапсуляції, де бізнес-інваріанти реалізуються у межах чітко визначених інтерфейсів, що запобігає несанкціонованому доступу до даних в обхід правил системи.

Впровадження автоматизованого генератора коду дозволяє досягти наступних показників ефективності:

1. Мінімізація ризиків «людського фактора».

Автоматизація створення SQL-запитів та методів мапінгу даних нівелює ймовірність синтаксичних та логічних помилок, що критично для підтримки цілісності великих систем.

2. Архітектурна узгодженість.

Генератор забезпечує дотримання галузевих стандартів та найкращих практик у межах усієї кодової бази, що спрощує процес аудиту та рецензування коду.

3. Масштабованість та адаптивність.

Синхронізація між рівнями системи підтримується автоматично; будь-які модифікації схеми бази даних оперативно транслюються у відповідні зміни на рівнях DAL та BLL, забезпечуючи високу мобільність розробки.

Таким чином, розроблений інструментарій представляє собою інтегроване середовище для прискорення циклу розробки n-рівневих

застосунків. Використання об’єктно-орієнтованої парадигми в поєднанні з автоматизацією рутинних операцій створює надійну основу для побудови високоякісного, підтримуваного ПЗ, що відповідає сучасним вимогам масштабованості та інформаційної безпеки.

2.2. Обґрунтування вибору технологічного стеку та інструментальних засобів розробки

Ефективність розробки автоматизованих систем генерації коду безпосередньо залежить від раціонального вибору інструментарію, який має забезпечувати високу продуктивність, надійність взаємодії з базами даних та підтримку сучасних методологій проектування. Нижче наведено детальний аналіз компонентів обраного технологічного стеку.

2.2.1. Середовище розробки та мова програмування

Microsoft Visual Studio .NET. Як основне інтегроване середовище розробки (IDE) було обрано Visual Studio, що зумовлено його комплексною функціональністю для екосистеми .NET. Середовище забезпечує повний життєвий цикл розробки, включаючи інструменти статичного аналізу коду, інтелектуальну систему підказок IntelliSense та вбудовані засоби профілювання. Важливою перевагою є глибока інтеграція з системами контролю версій (Git), що критично для підтримки цілісності кодової бази при ітеративній розробці.

Мова програмування C#. Вибір C# аргументований її об’єктно-орієнтованою природою, строгою типізацією та високим рівнем абстракції [8]. Використання сучасних версій мови дозволяє реалізувати складні алгоритми аналізу метаданих бази даних завдяки потужним засобам рефлексії та LINQ. Ефективне управління пам'яттю (Garbage Collection) та розгалужена стандартна бібліотека роблять C# оптимальним інструментом для побудови надійних корпоративних рішень [9].

2.2.2. Система управління базами даних та засоби адміністрування

У ролі цільової системи управління базами даних обрано MySQL, що характеризується високою швидкістю обробки запитів, масштабованістю та надійністю персистентного зберігання інформації [10]. В межах проєкту програма взаємодіє з MySQL для інтроспекції схем даних та автоматичної генерації збережених процедур, що дозволяє оптимізувати навантаження на рівні DAL [4].

Для візуального моделювання та адміністрування реляційних структур використовується MySQL Workbench. Цей інструмент дозволяє здійснювати пряме та зворотне проєктування (forward/reverse engineering), що полегшує верифікацію структури БД перед етапом автоматичної генерації програмного коду [11].

2.2.3. Тестування та забезпечення якості коду

Забезпечення стабільності автоматизованого генератора реалізується через впровадження модульного тестування (Unit Testing) на базі фреймворку xUnit. Його застосування дозволяє реалізувати практики розробки через тестування (TDD), що суттєво знижує ризик виникнення регресійних помилок при розширенні функціональності генератора для різних типів архітектур [12].

Управління процесом розробки здійснюється за допомогою платформи Jira, яка підтримує гнучкі методології (Agile), зокрема Scrum та Kanban. Використання Jira дозволяє декомпозувати процес створення генератора на окремі спринти, відстежувати прогрес через діаграми згоряння задач (Burndown charts) та підтримувати прозорість розробки. Інтеграція Jira з процесами розробки забезпечує швидку адаптацію до змінних вимог та високу якість кінцевого програмного продукту.

Обраний комплекс інструментів формує цілісну технологічну платформу, яка поєднує потужні засоби розробки, надійні механізми роботи з даними та сучасні підходи до управління якістю. Використання даного стеку

гарантує створення масштабованої архітектури, здатної до подальшої модернізації, та забезпечує високу швидкість виконання операцій у межах n-рівневої структури застосунку.

2.3. Проєктування статичної структури системи за допомогою діаграм класів

У процесі об’єктно-орієнтованого аналізу та проєктування ключовим етапом є побудова статичної моделі системи. Основним інструментом для візуалізації архітектурного каркаса програмного забезпечення виступає діаграма класів, що є фундаментальним компонентом уніфікованої мови моделювання (UML — Unified Modeling Language) [13].

Діаграма класів належить до категорії структурних діаграм UML і призначена для опису внутрішньої архітектури системи через визначення її класів, їхніх атрибутів, операцій (методів), а також типів взаємозв’язків між ними. У контексті розробки автоматизованих генераторів коду використання даної діаграми дозволяє вирішити наступні концептуальні завдання:

- Декомпозиція складної системи: поділ загальної логіки програми на окремі об’єкти з чітко визначеними зонами відповідальності (Single Responsibility Principle).

- Типізація та специфікація: точне визначення типів даних для атрибутів та сигнатур методів, що є критично важливим для мов зі строгою типізацією, таких як C#.

- Семантичне моделювання зв’язків: візуалізація способів взаємодії об’єктів через механізми асоціації (зв’язок «використовує»), узагальнення/успадкування (зв’язок «є»), агрегації та композиції.

На основі аналізу функціональних вимог до системи було розроблено розширену діаграму класів, яка відображає внутрішню логіку процесу генерації коду. На відміну від базових моделей, представлена структура фокусується на декомпозиції метаданих бази даних та методах їх

трансформації у програмні артефакти n-рівневої архітектури.

На рисунку 2.1 представлена спроєктована діаграма класів системи, що відображає архітектурні компоненти, відповідальні за інтроспекцію бази даних та подальший синтез коду.

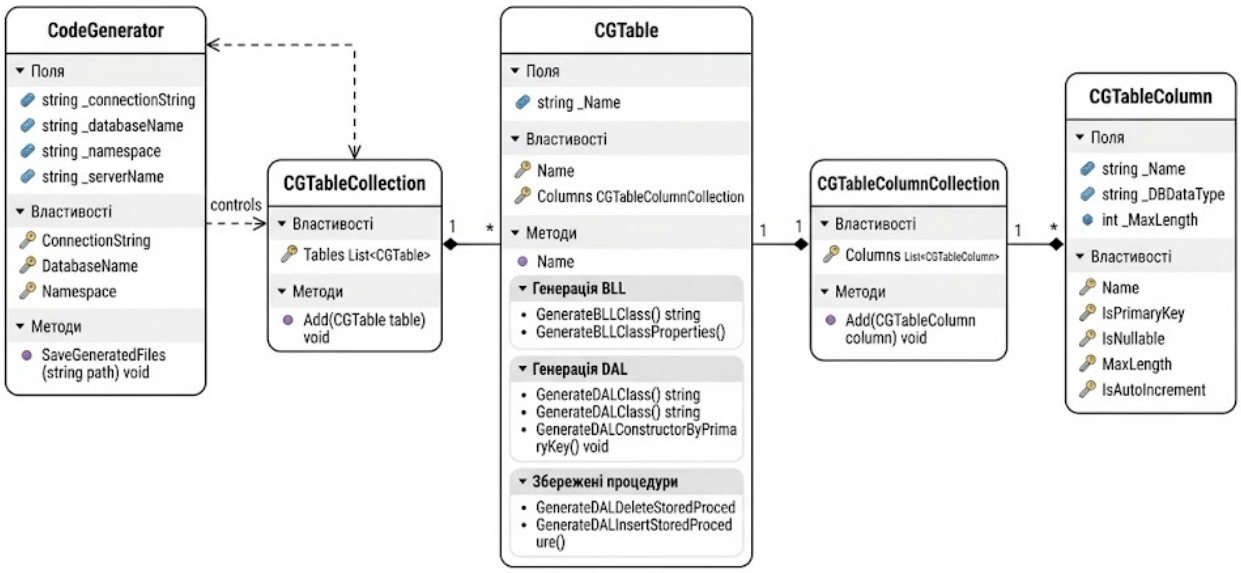

Рисунок 2.1 — Діаграма класів автоматизованого генератора багаторівневої архітектури

Центральними елементами моделі є класи, що інкапсулюють метадані об’єктів СУБД.

CGTable - клас, що моделює структуру таблиці бази даних. Він містить перелік атрибутів (назва, схема) та методи для обробки колекцій стовпців. Він містить розширений набір методів для генерації коду:

- Група BLL: методи для створення класів бізнес-логіки, їхніх властивостей та конструкторів.

- Група DAL: методи для формування шару доступу до даних та відповідних методів мапінгу.

- Група SQL: методи для автоматичного синтезу збережених процедур (INSERT, UPDATE, DELETE, SELECT).

Клас CGTableColumn - об’єктне представлення окремого атрибута

(стовпця) таблиці, що включає специфікації типів даних, прапорці ідентифікаторів (Primary Key) та правила обов'язковості заповнення (Nullability).

Колекції (CGTableCollection, CGTableColumnCollection) забезпечують типізоване зберігання та ітерацію по структурах бази даних, що дозволяє обробляти довільну кількість таблиць та стовпців у межах однієї сесії генерації.

Взаємозв'язки між даними класами визначають логіку агрегації метаданих, що дозволяє генератору трансформувати реляційну структуру бази даних у відповідні класи C# на рівні бізнес-логіки (BLL) та доступу до даних (DAL).

Діаграма класів слугує сполучною ланкою між концептуальним дизайном та безпосередньою реалізацією. Вона забезпечує:

- Узгодженість комунікації: надання зацікавленим сторонам (архітекторам, розробникам, тестувальникам) єдиного візуального стандарту розуміння системи.

- Документування архітектури: створення базису для майбутньої модернізації та розширення системи.

- Генерацію коду: можливість автоматичного формування скелетів класів на основі графічної моделі, що є логічним продовженням ідеї автоматизації в даному проєкті.

2.4. Методологія гнучкої розробки та застосування історій користувачів при створенні прототипу генератора коду

Проєкт з розробки автоматизованого генератора коду (Code Generator) реалізовувався з використанням методологігії Agile, яка на сучасному етапі є стандартом для динамічних IT-проєктів. Цей вибір був обумовлений необхідністю забезпечення високої адаптивності до змінних вимог, ітеративного та інкрементного підходу до створення продукту, а також

безперервної співпраці з зацікавленими сторонами для досягнення максимальної відповідності кінцевого продукту потребам користувачів. Фундаментальною основою для визначення та формалізації функціональних вимог до системи виступали історії користувачів (User Stories).

Історії користувачів — це прості, зрозумілі всім зацікавленим сторонам (включаючи замовників, архітекторів та розробників) формулювання функціональних вимог, написані в наративному стилі. Вони фіксують "хто", "що" і "навіщо" в рамках конкретної користувацької взаємодії, відповідаючи на питання: [Роль] бажає [Функціональність], щоб [Цінність].

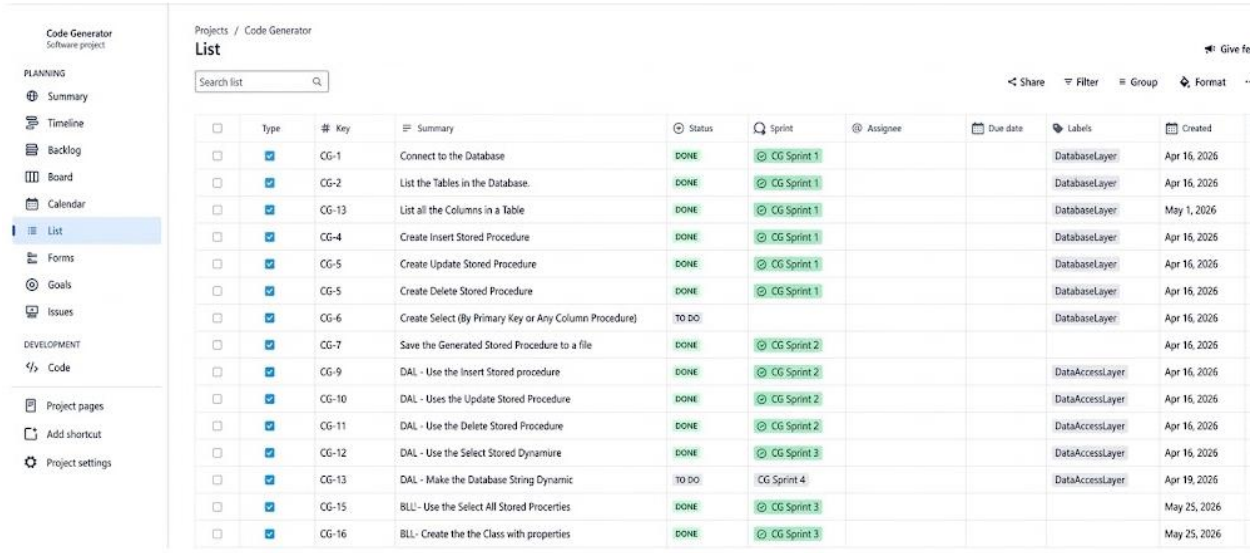

Рисунок 2.2 – Приклади історій користувачів для генератора коду

Наголос на простоті та наративному форматі дозволяє спростити процес захоплення вимог, роблячи його ідеальним для початкових етапів проєктування та для забезпечення спільного розуміння між технічною командою та кінцевими користувачами. Історії користувачів виступають не як кінцеві специфікації, а як основа для подальшого обговорення та деталізації, що дозволяє розробникам фокусуватися на реальних потребах та бізнес-цінностях. Ілюстративний приклад таких історій, використаних при створенні генератора коду, представлено на рисунку 2.2.

Для адміністрування та ефективного управління процесом розробки на основі гнучких практик було застосовано спеціалізоване програмне забезпечення Jira Software від компанії Atlassian, яке є визнаним лідером серед CASE-засобів для Agile-команд. У Jira здійснювалися наступні ключові процеси:

- Акумуляція та ведення беклогу вимог: Історії користувачів акумулювалися в системі як окремі завдання (Issues), що дозволяло централізовано зберігати, структурувати та пріоритезувати їх. До кожної історії визначалися Acceptance Criteria (Критерії приймання) та додавалися необхідні деталі.

- Планування спринтів (Sprint Planning): Беклог продукту, сформований з історій користувачів, проходив етапи оцінювання складності та пріоритезації. На основі оцінок та бізнес-цілей завдання розподілялися за спринтами, що забезпечувало чітке розуміння обсягу робіт на кожну ітерацію.

- Моніторинг та відстеження прогресу: Jira дозволяла візуалізувати стан виконання кожного завдання на Scrum- або Kanban-дошках, відстежувати час виконання та формувати різноманітні аналітичні звіти (наприклад, Burndown Charts, Cumulative Flow Diagrams), що сприяло прозорості та своєчасному виявленню проблем.

Застосування методології Agile в поєднанні з використанням історій користувачів та інструментарію Jira забезпечило створення Code Generator як продукту, що не лише технічно коректно виконує свої функції, але й максимально відповідає реальним потребам та очікуванням користувачів. Ітеративний підхід дозволив команді гнучко адаптуватися до змін та безперервно покращувати продукт, а чітка формалізація вимог через історії користувачів запобігла архітектурним невідповідностям та непорозумінням.

2.5. Логічне моделювання реляційної структури бази даних для роботи генератора коду

Проєктування архітектури бази даних базується на принципах нормалізації (до третьої нормальної форми — 3NF включно), що дозволяє мінімізувати надмірність даних та забезпечити цілісність інформації. Наведена нижче модель описує взаємозв’язки між ключовими сутностями освітнього процесу в межах n-рівневої архітектури.

Концептуальна схема бази даних, представлена на рисунку 2.3, є результатом декомпозиції предметної області освітнього процесу. Проєктування моделі здійснювалося з дотриманням принципів реляційної алгебри та вимог третьої нормальної форми (3NF), що мінімізує аномалії вставки, оновлення та видалення даних.

Система базується на шести ключових сутностях, які утворюють три функціональні кластери: адміністративний, навчальний та результативний.

1. Адміністративно-організаційний кластер (Departments, Teachers)

Фундаментом системи є сутність Departments, яка інкапсулює інформацію про структурні підрозділи (кафедри).

Ієрархічна когерентність: Зв'язок між Departments та Teachers реалізований за типом «один-до-багатьох» (1:N). Кожен викладач асоційований з конкретною кафедрою через зовнішній ключ DeptId.

Рекурсивна залежність: Особливістю моделі є наявність циклічного зв’язку в сутності Departments через атрибут HeadTeacherId, що дозволяє програмно ідентифікувати керівника підрозділу серед викладацького складу, забезпечуючи цілісність організаційної ієрархії.

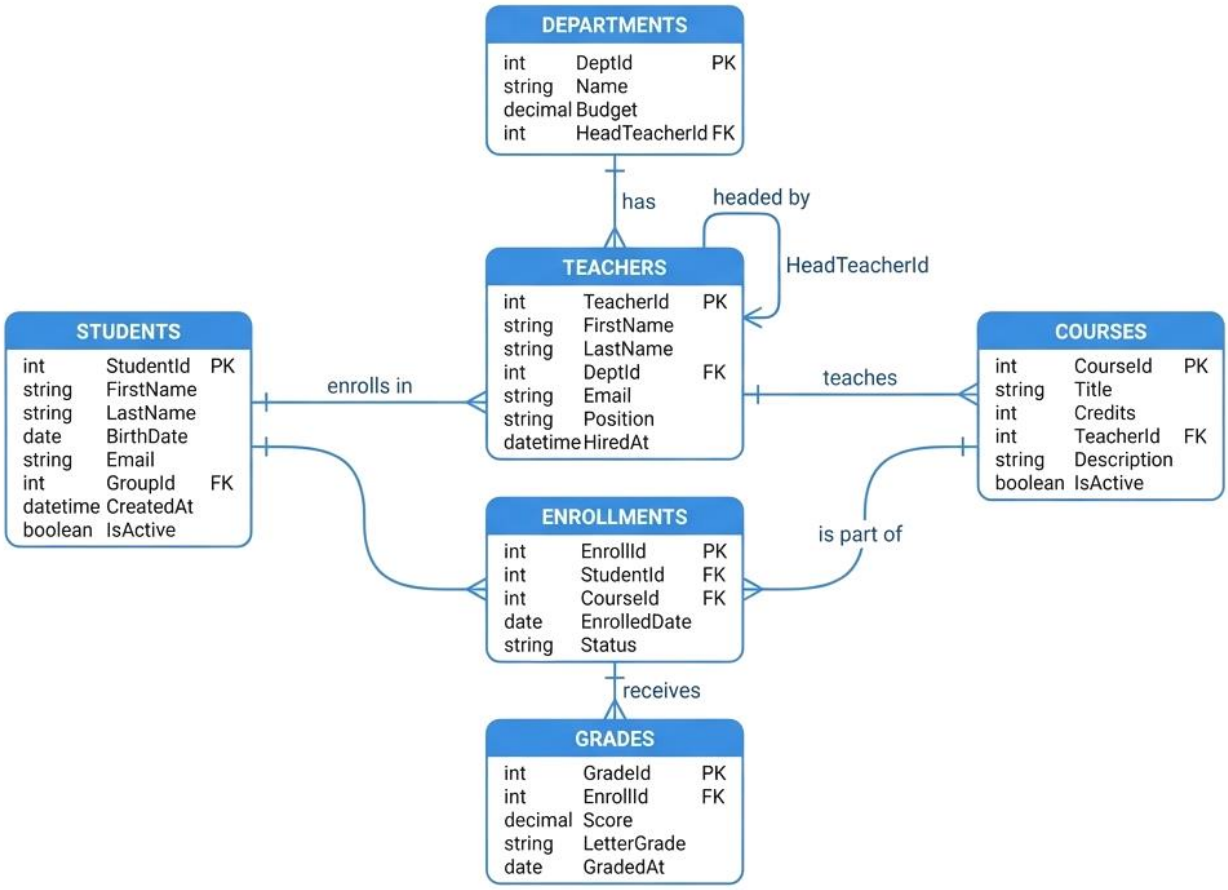

Рисунок 2.3 – Модель бази даних для демонстрації роботи пропонованого генератора коду

2. Навчально-методичний кластер (Courses, Students)

Цей блок описує взаємодію між суб'єктами та об'єктами освітньої діяльності.

Детермінація навчальних дисциплін: Сутність Courses містить опис курсів та їхню академічну вагу (Credits). Вона жорстко пов'язана з сутністю Teachers, що фіксує персональну відповідальність викладача за конкретну дисципліну.

Профіль суб'єкта навчання: Сутність Students акумулює персональні дані студентів. Атрибут IsActive (тип boolean) дозволяє реалізувати механізм «м’якого видалення» (Soft Delete), зберігаючи історичність даних у межах n-рівневої архітектури без фізичної деструкції записів.

3. Транзакційний кластер та облік успішності (Enrollments, Grades)

Найбільш динамічна частина моделі, що фіксує взаємодію «Студент — Курс».

Резолюція зв'язків «багато-до-багатьох»: Сутність Enrollments виступає асоціативною таблицею, що трансформує зв'язок M:N між студентами та курсами у два зв'язки типу 1:N. Це забезпечує можливість реєстрації одного студента на декілька дисциплін одночасно.

Фіксація результатів: Сутність Grades функціонально залежить від конкретної реєстрації (EnrollId).

Такий підхід гарантує, що оцінка може бути виставлена виключно в межах офіційного запису студента на курс, забезпечуючи референціальну цілісність вищого порядку.

Описана структура є джерелом метаданих для розробленого генератора коду. Чітка типізація атрибутів (від int для ідентифікаторів до decimal для фінансових та рейтингових показників) дозволяє генератору автоматично обирати відповідні типи даних у мові C#.

Наявність первинних (PK) та зовнішніх (FK) ключових полів є базисом для автоматичного синтезу SQL-процедур вибору за ключем (SelectByPK) та підтримки консистентності на рівні доступу до даних (DAL).

Представлена модель даних забезпечує необхідний баланс між продуктивністю запитів та гнучкістю архітектури. Вона не лише відображає поточний стан предметної області, а й закладає фундамент для масштабування системи (наприклад, додавання модулів розкладу чи фінансового обліку) без радикальної переробки існуючої схеми.

2.6. Оптимізація продуктивності доступу до даних через механізми індексації

Для забезпечення високої продуктивності генератора, розробленого на базі багаторівневої архітектури, недостатньо лише логічного проєктування схеми. Необхідним етапом є фізична оптимізація бази даних, що реалізується

через механізми індексації. У контексті роботи автоматизованого генератора коду, правильна стратегія індексації дозволяє суттєво знизити латентність при виконанні CRUD-операцій та пришвидшити процес інтроспекції метаданих.

Індексація в реляційних СУБД (зокрема MySQL, що використовується в даному проєкті) є критичним інструментом для прискорення операцій пошуку та вибірки даних. Без індексів система змушена виконувати повне сканування таблиці (Full Table Scan), що призводить до лінійного зростання часу виконання запитів залежно від обсягу даних.

2.6.1. Класифікація та застосування індексів у системі

У розробленій моделі бази даних впроваджено наступні типи індексів для забезпечення детермінованої швидкодії:

Кластеризовані індекси (Clustered Indexes):

Автоматично створюються для кожного первинного ключа (PK), наприклад, StudentId або TeacherId. У СУБД MySQL (двигун InnoDB) дані фізично впорядковуються відповідно до значень кластеризованого індексу. Це забезпечує максимально швидкий доступ до конкретного запису, що є критично важливим для операцій SelectByPK, які генеруються автоматично.

Некластеризовані (вторинні) індекси (Non-Clustered Indexes):

Проєктуються для атрибутів, які часто використовуються в умовах фільтрації (WHERE) або сортування (ORDER BY).

Індексація зовнішніх ключів (FK): Атрибути типу DeptId у таблиці Teachers або CourseId у Enrollments потребують індексації для оптимізації операцій об'єднання (JOIN). Це дозволяє системі миттєво знаходити всі пов'язані записи, не скануючи всю таблицю.

Пошукові індекси: Для полів Email або LastName доцільно впроваджувати унікальні або звичайні індекси для прискорення ідентифікації користувачів.

2.6.2. Вплив індексації на роботу генератора коду

Розроблений прототип генератора враховує наявність індексів при синтезі програмного коду рівня доступу до даних (DAL):

- Автоматичне визначення предикатів. Аналізуючи метадані, генератор ідентифікує поля з індексами та пріоритезує їх при формуванні параметрів для збережених процедур. Це гарантує, що згенеровані SQL-запити будуть використовувати найбільш ефективні шляхи доступу до даних.

- Зниження навантаження на BLL. Завдяки тому, що вибірка та фільтрація виконуються на рівні СУБД за допомогою індексів, об'єкти бізнес-логіки отримують уже відфільтровані та впорядковані колекції, що мінімізує витрати пам'яті на боці сервера застосунків.

Важливим науковим аспектом є дотримання балансу: надмірна кількість індексів може сповільнити операції вставки (INSERT) та оновлення (UPDATE), оскільки СУБД змушена перераховувати дерево індексів при кожній модифікації даних. У межах даного проєкту стратегія індексації була оптимізована для типових сценаріїв використання освітньої платформи, де кількість операцій читання значно переважає кількість операцій запису (співвідношення приблизно 10:1).

Таким чином, механізми індексації виступають «фізичним фундаментом» для високорівневої автоматизації. Вони забезпечують масштабованість системи, дозволяючи згенерованому коду ефективно працювати навіть при зростанні обсягів бази даних до мільйонів записів, зберігаючи при цьому стабільний час відгуку інтерфейсу користувача.

2.7. Алгоритмічне забезпечення процесів автоматизованого синтезу програмного коду мовою C#

Процес трансформації реляційних моделей даних у програмні артефакти багаторівневої архітектури базується на принципах метапрограмування. Основним завданням розроблених алгоритмів є

забезпечення семантичної відповідності між структурами бази даних та об'єктами мови C#, що реалізується через багатоетапний процес перетворення.

2.7.1. Алгоритм інтроспекції та мапінгу типів даних

Першим етапом роботи генератора є детермінований аналіз метаданих СУБД. Програма виконує ітерацію по колекції стовпців обраної таблиці (CGTableColumnCollection), ідентифікуючи типи даних SQL. Ключовим елементом алгоритму є матриця відповідності типів, яка забезпечує коректну типізацію в середовищі .NET:

- varchar, text, char → string
- int, integer → int
- decimal, numeric → decimal
- datetime, timestamp → DateTime
- bit, boolean → bool

Алгоритм також враховує властивість IsNullable: якщо стовпець у базі даних дозволяє пусті значення, генератор автоматично використовує nullable-типи в C# (наприклад, int? або DateTime?), що запобігає виникненню виключень NullReferenceException на рівні бізнес-логіки.

2.7.2. Шаблонний синтез рівнів DAL та BLL

Генерація коду реалізована за шаблонним методом (Template-based generation). Замість прямого маніпулювання рядками, алгоритм використовує попередньо визначені структури класів, у які ін’єктуються динамічні дані.

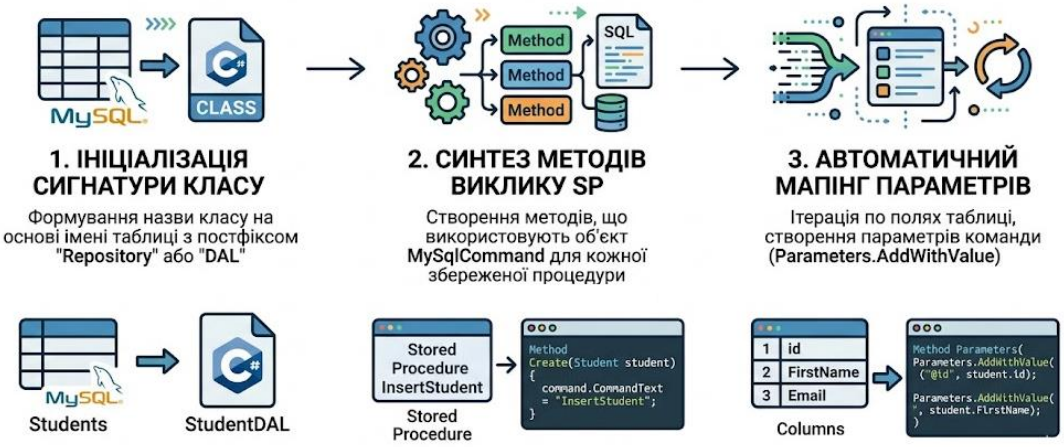

Рисунок 2.4 – Логічна послідовність генерації рівня доступу до даних

Логічна послідовність генерації рівня доступу до даних (DAL):

1. Ініціалізація сигнатури класу: формування назви класу на основі імені таблиці з постфіксом Repository або DAL.

2. Синтез методів виклику SP: для кожної збереженої процедури (Insert, Update, Delete, Select) створюється метод, що використовує об’єкт MySqlCommand.

3. Автоматичний мапінг параметрів: алгоритм ітерує по полях таблиці, створюючи параметри команди (Parameters.AddWithValue) та забезпечуючи відповідність імен змінних атрибутам бази даних.

Логічна послідовність генерації рівня бізнес-логіки (BLL):

1. Формування властивостей (Properties): створення публічних властивостей класу з автоматичними гетерами та сетерами, що відповідають стовпцям таблиці.

2. Генерація конструкторів: алгоритм створює дефолтний конструктор та конструктор з параметрами (за первинним ключем) для автоматичного завантаження об’єкта з бази.

3. Інкапсуляція викликів DAL: методи BLL (наприклад, Save(), Delete()) реалізуються як обгортки над методами відповідного класу DAL, забезпечуючи чітке розділення відповідальності.

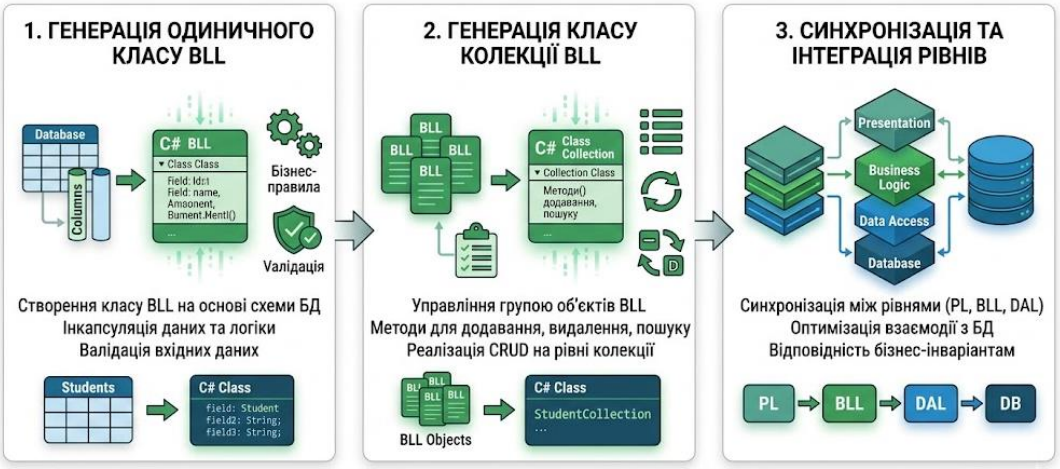

Рисунок 2.5 - Логічна послідовність генерації рівня бізнес-логіки

На рисунку 2.5 візуалізовано складний процес трансформації метаданих бази даних у високорівневі об'єктно-орієнтовані структури мови C#. Ця схема демонструє ієрархічний підхід до формування бізнес-логіки, де кожен етап додає новий рівень абстракції та функціональності.

Виконаємо детальний аналіз кожного з трьох ключових блоків схеми.

Етап 1. Генерація одиничного класу сутності

Перший етап фокусується на створенні атомарної одиниці системи — класу, що представляє один рядок таблиці бази даних (наприклад, Student).

Вхідні дані: Алгоритм використовує перелік стовпців (Columns) зі схеми БД.

Механізм інкапсуляції: Генератор автоматично створює закриті поля та публічні властивості. Це забезпечує захист даних (Encapsulation), дозволяючи контролювати доступ до стану об'єкта.

Інтелектуальна валідація: На цьому рівні в код закладаються базові бізнес-правила. Наприклад, якщо поле в БД позначено як NOT NULL, згенерований сетер властивості в C# може містити логіку перевірки на порожнє значення, запобігаючи некоректному стану об'єкта ще до спроби збереження в базу.

Результат: Чистий C#-клас, який є цифровим двійником реальної сутності.

Етап 2. Синтез класу типізованих колекцій

Другий блок ілюструє перехід від роботи з одним об'єктом до управління масивами даних. Це критично для систем, що обробляють велику кількість записів одночасно.

Функціональне призначення: Клас-колекція (наприклад, StudentCollection) розширює можливості стандартних списків .NET.

Реалізація CRUD-інтерфейсу: На рівні колекції генеруються методи для пакетної обробки:

Пошук: Швидка фільтрація об'єктів за заданими критеріями.

Ітерація: Зручний обхід всіх екземплярів для відображення в інтерфейсі користувача.

Управління: Методи додавання та видалення, які синхронізовані зі станом бази даних.

Значення: Це дозволяє розробнику оперувати поняттям «список студентів» як єдиним логічним об'єктом, а не набором розрізнених рядків.

Етап 3. Синхронізація та системна інтеграція рівнів

Заключний блок схеми демонструє цілісну n-рівневу архітектуру, де рівень BLL виступає інтелектуальним посередником (медіатором).

Вертикальна інтеграція: Схема показує потік даних від бази даних (Database) через рівень доступу (DAL) до бізнес-логіки (BLL) і, зрештою, до рівня представлення (Presentation).

Дотримання інваріантів: BLL гарантує, що жодні дані не потраплять у DAL без перевірки на відповідність бізнес-правилам. Це забезпечує архітектурну чистоту: рівень представлення «не знає» про існування бази даних, він взаємодіє лише з об'єктами BLL.

Оптимізація взаємодії: Схема акцентує увагу на тому, що використання згенерованих шарів мінімізує надмірні запити до БД, оскільки бізнес-об'єкти можуть кешувати стан або виконувати складні обчислення на боці сервера застосунків.

Отже, рисунок 2.5 є візуальним підтвердженням того, що автоматизація

не просто копіює дані, а вибудовує масштабовану екосистему. Такий підхід дозволяє розробнику фокусуватися на унікальних особливостях проєкту, довіряючи генератору створення надійного та стандартизованого фундаменту системи.

2.7.3. Алгоритм генерації типізованих колекцій

Для ефективного управління групами об'єктів розроблено алгоритм синтезу класів-колекцій (наприклад, StudentCollection). Він базується на успадкуванні від стандартних типів (наприклад, List<T>), додаючи специфічні методи:

- LoadAll(): метод, що заповнює колекцію екземплярами класів BLL, використовуючи дані з процедури SelectAll.

- FindById(): алгоритм лінійного або бінарного пошуку в межах колекції за значенням первинного ключа.

Описані алгоритми забезпечують повну автоматизацію рутинних процесів кодування. Завдяки суворому дотриманню правил мапінгу та використанню шаблонного синтезу, генератор створює код, який не потребує ручного рефакторингу та є повністю готовим до інтеграції в цільову інформаційну систему.

2.8. Архітектура та алгоритми багаторівневої валідації в синтезованому бізнес-шарі

У контексті розробки автоматизованих систем генерації коду, валідація розглядається як критичний механізм забезпечення цілісності даних та захисту бізнес-інваріантів. У межах n-рівневої архітектури валідація не є атомарною подією, а являє собою багаторівневий процес, інтегрований безпосередньо в структуру бізнес-об'єктів (BLL).

Валідація в згенерованому коді базується на принципі «оборонного програмування», де кожен об'єкт BLL виступає фільтром, що запобігає

проникненню некоректних станів у нижчі рівні системи (DAL та СУБД).

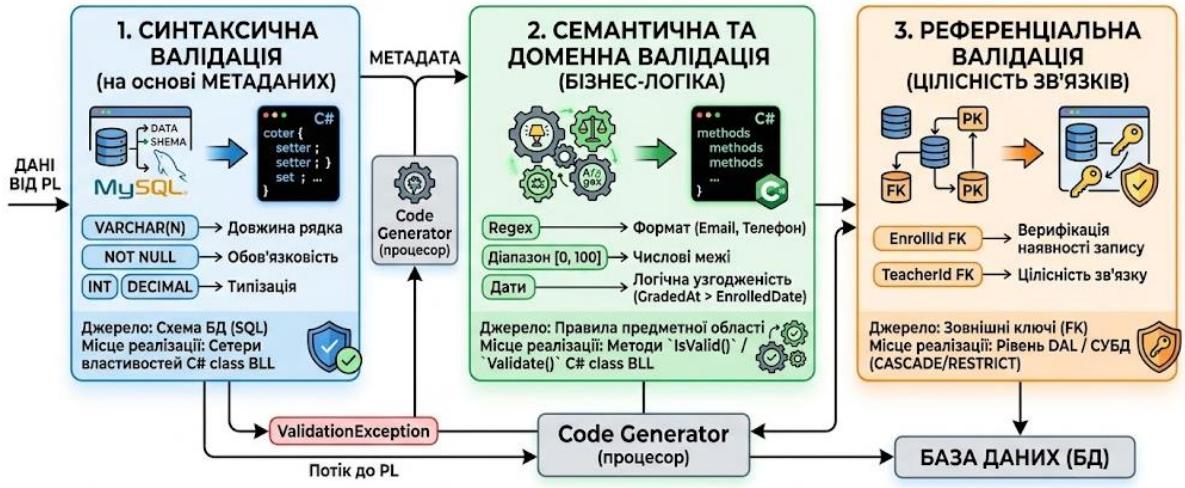

Рисунок 2.6 – Архітектура багаторівневої валідації даних в пропонованому генераторі коду

2.8.1. Синтаксична валідація на основі метаданих

Цей рівень валідації генерується автоматично на основі аналізу схеми бази даних. Алгоритм генератора трансформує фізичні обмеження SQL у логічні перевірки в коді C#:

- Обмеження довжини - якщо стовпець у базі даних визначений як VARCHAR(N), сетер відповідної властивості в C# включає перевірку:

$$if\ (value.Length > N) \rightarrow throw\ Exception$$

- Обов'язковість. Для полів NOT NULL генератор впроваджує перевірки на null або string.Empty. Це дозволяє ідентифікувати помилки на боці сервера застосунків, не ініціюючи дороговартісний запит до БД.

- Сувора типізація мови C# сама по собі виступає першим рівнем валідації, унеможливлюючи передачу текстових даних у числові ідентифікатори.

2.8.2. Семантична та доменна валідація

Окрім технічних обмежень, бізнес-об'єкти включають перевірки, що базуються на логіці предметної області. Генератор дозволяє ін'єктувати шаблони для типових даних:

- Форматна валідація - використання регулярних виразів (Regex) для перевірки коректності електронних адрес, номерів телефонів або поштових індексів.

- Діапазонна валідація - перевірка числових значень (наприклад, оцінка в реєстрі Grades має бути в межах [0;100]).

- Логічна узгодженість - перевірка дат (наприклад, GradedAt не може бути раніше за EnrolledDate).

2.8.3. Патерни реалізації валідації в згенерованому коді

Генератор використовує два основні підходи до впровадження валідаційної логіки:

- Валідація в сетерах

Найбільш оперативний метод, де перевірка виконується миттєво при спробі змінити значення властивості об'єкта. Це гарантує, що об'єкт завжди перебуває в консистентному стані.

- Декларативна валідація

Генератор може додавати атрибути (наприклад, [Required], [StringLength]) до властивостей класу. Це дозволяє використовувати стандартні механізми валідації .NET, що особливо корисно при подальшій інтеграції з рівнем представлення (ASP.NET Core або WPF).

Таблиця 2.1

Особливості типів валідацій

Тип валідації	Джерело метаданих	Місце реалізації	Мета
Технічна	Схема БД (SQL)	Сетери властивостей	Запобігання SQL-помилкам
Бізнес-логіка	Опис предметної області	Методи `IsValid()` / `Validate()`	Дотримання правил проєкту
Референціальна	Зовнішні ключі (FK)	Рівень DAL / СУБД	Цілісність зв'язків

Алгоритм валідації інтегрований із системою обробки виключень. При виявленні невідповідності, BLL-об'єкт генерує типізоване виключення (наприклад, ValidationException), яке містить деталізований опис порушеного правила. Це дозволяє рівню представлення (PL) коректно візуалізувати помилку для кінцевого користувача.

Автоматизована генерація механізмів валідації забезпечує «стерильність» бази даних, оскільки некоректні запити відсікаються ще на етапі ініціалізації об'єктів. Це суттєво знижує навантаження на СУБД та спрощує налагодження системи, оскільки джерело помилки локалізується на рівні коду бізнес-логіки.

2.9 Висновки по розділу

У другому розділі розроблено методологічні та алгоритмічні основи створення прототипу генератора програмного коду для багаторівневих систем. Запропоновано підхід до автоматизації генерації коду на основі структурного аналізу реляційних баз даних, що забезпечує використання метаданих як основного джерела для синтезу програмних компонентів. Обґрунтовано вибір технологічного стеку, який забезпечує ефективність реалізації, зручність розробки та інтеграції з існуючими системами. Розроблено формальні моделі структури системи та логічної організації бази даних, що стали основою для побудови генератора. Створено алгоритми інтроспекції, мапінгу типів даних та шаблонного синтезу компонентів рівнів DAL і BLL, які забезпечують автоматизоване формування типобезпечного коду. Додатково розроблено механізми багаторівневої валідації даних, що підвищують надійність і коректність функціонування згенерованих програмних компонентів.

РОЗДІЛ 3. ПРОЕКТУВАННЯ ТА РЕАЛІЗАЦІЯ ПРОТОТИПУ ГЕНЕРАТОРА ПРОГРАМНОГО КОДУ ДЛЯ БАГАТОРІВНЕВИХ АРХІТЕКТУР

3.1. Програмна реалізація графічного інтерфейсу та регламент експлуатації генератора коду

Графічний інтерфейс користувача розробленого програмного забезпечення спроєктований з урахуванням принципів ергономічності та інтуїтивності, що дозволяє мінімізувати поріг входження для розробників різного рівня кваліфікації. Основним завданням інтерфейсу є забезпечення зручної взаємодії з механізмами інтроспекції бази даних та візуалізація процесу синтезу програмних артефактів.

3.1.1. Алгоритм функціонування та послідовність дій користувача

Процес автоматизованої генерації коду базується на чітко визначеному операційному регламенті, який включає наступні етапи:

1. Конфігурація з'єднання та ініціалізація метаданих:

Користувач вводить рядок підключення (Connection String) до цільової СУБД. Функція «Тест підключення» ініціює запит до системного каталогу бази даних для верифікації прав доступу. У разі успішної автентифікації виконується завантаження переліку доступних таблиць. Функція «Завантажити дані» дозволяє оновити стан об'єктів у реальному часі при зміні схеми БД без перезавантаження застосунку.

2. Визначення цільової директорії:

Користувач обирає папку вихідних файлів, де буде сформовано ієрархію каталогів для скриптів збережених процедур, класів DAL, BLL та утилітарних компонентів. Це забезпечує локалізацію всіх згенерованих артефактів для подальшої інтеграції в основний проєкт.

3. Селекція об’єктів генерації:

На основі завантаженого списку виконується вибір конкретної таблиці. Система аналізує атрибутивний склад обраної сутності для динамічного формування параметрів генерації.

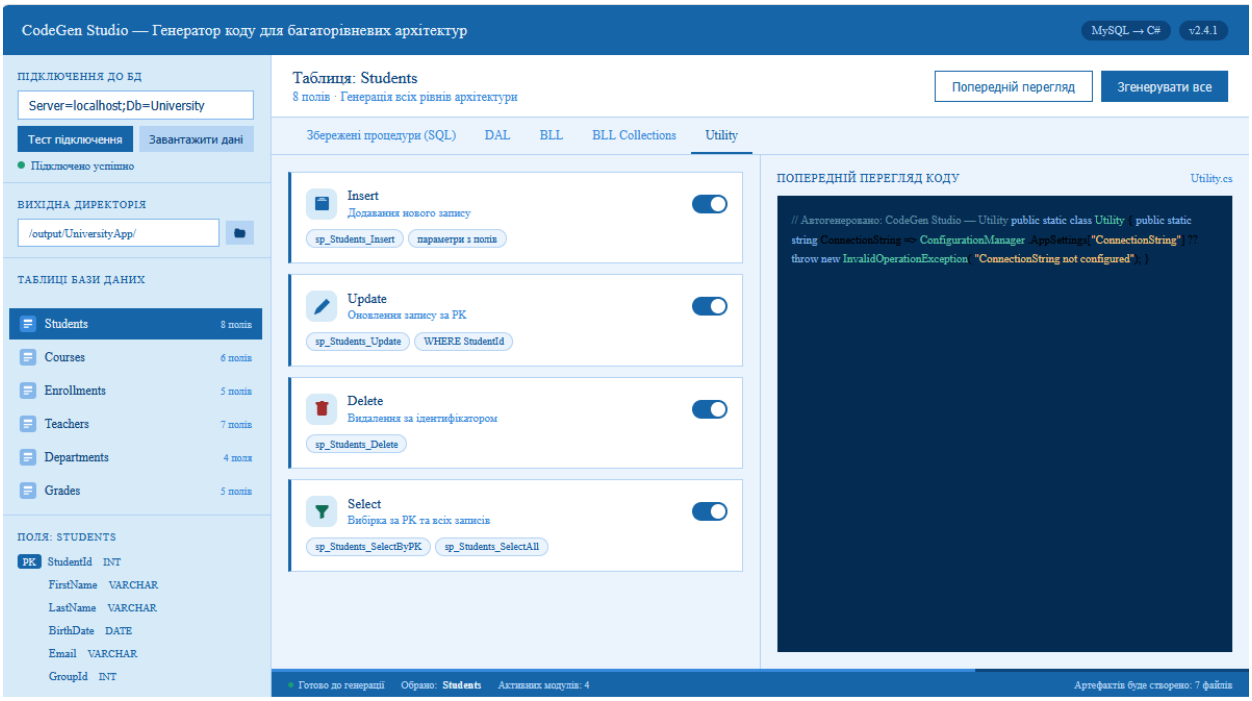

Рисунок 3.1 - Графічний інтерфейс користувача

3.1.2. Функціональні модулі синтезу програмного коду

Програмний прототип пропонує гранулярний підхід до генерації, дозволяючи створювати окремі компоненти багаторівневої архітектури:

1. Модуль генерації збережених процедур (SQL Scripting):

Здійснює синтез MySQL-скриптів для базових маніпуляцій даними. Це включає:

- Процедура вставки (Insert): додавання нових кортежів.

- Процедура оновлення (Update): модифікація існуючих записів за ідентифікатором.

- Процедура видалення (Delete): деструктивні операції з даними.

- Процедури вибору (Select By PK / Select All): методи отримання даних, що оптимізують плани виконання запитів на стороні СУБД.

2. Рівень доступу до даних (Data Access Layer, DAL):

Генерує класи на мові C#, що інкапсулюють виклики вищезгаданих збережених процедур. Це створює надійний міст між реляційним та об'єктним світами, абстрагуючи логіку підключення від решти застосунку.

3. Рівень бізнес-логіки (Business Logic Layer, BLL):

Формує класи, що відповідають за обробку даних, валідацію та трансляцію інформації до рівня представлення. Таке розмежування забезпечує високу придатність до тестування (unit testing) та модифікації без ризику пошкодження логіки доступу до даних.

4. Колекції рівня бізнес-логіки (BLL Collections):

Спеціалізований модуль для генерації типізованих колекцій. Це спрощує роботу з масивами об'єктів (наприклад, StudentCollection), реалізуючи стандартні методи ітерації, пошуку та пакетної обробки екземплярів класів.

5. Утилітарний клас (Utility Class):

Створює спільний ресурс для зберігання конфігураційних параметрів (наприклад, ConnectionString), що забезпечує єдину точку входу для налаштувань підключення у всіх класах DAL.

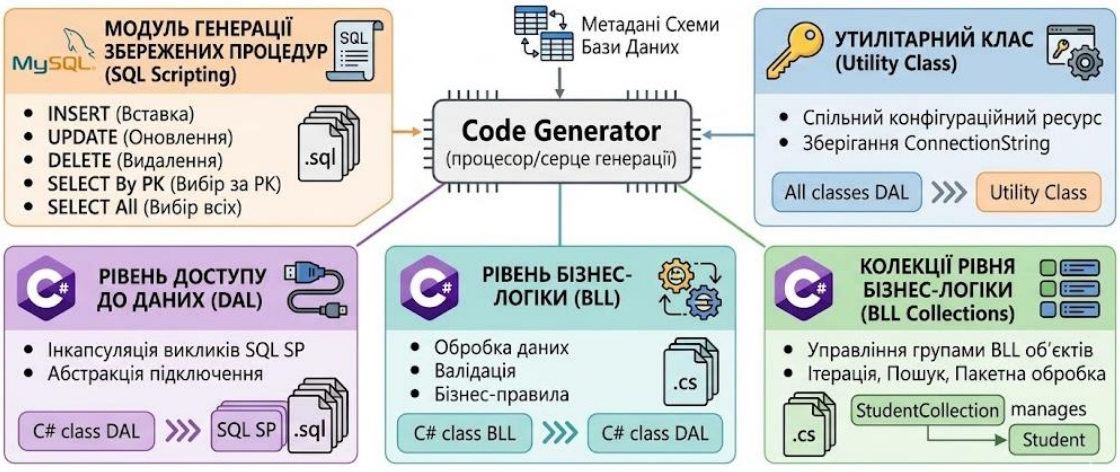

Рисунок 3.2 – Функціональні модулі генератора коду

Таким чином, розроблений інтерфейс та логіка його роботи дозволяють перевести процес розробки інфраструктурного коду в площину

конфігураційного моделювання. Це не лише прискорює розробку, а й гарантує високу архітектурну чистоту створюваних n-рівневих систем.

3.2. Архітектурна декомпозиція та функціональні можливості інтерфейсу генератора коду

Графічний інтерфейс користувача системи CodeGen Studio спроєктований за модульним принципом, що забезпечує логічне розмежування процесів конфігурації, селекції метаданих та препроцесорної візуалізації коду. Ергономічна структура вікна розділена на три основні функціональні зони, що дозволяє реалізувати лінійний робочий процес розробки.

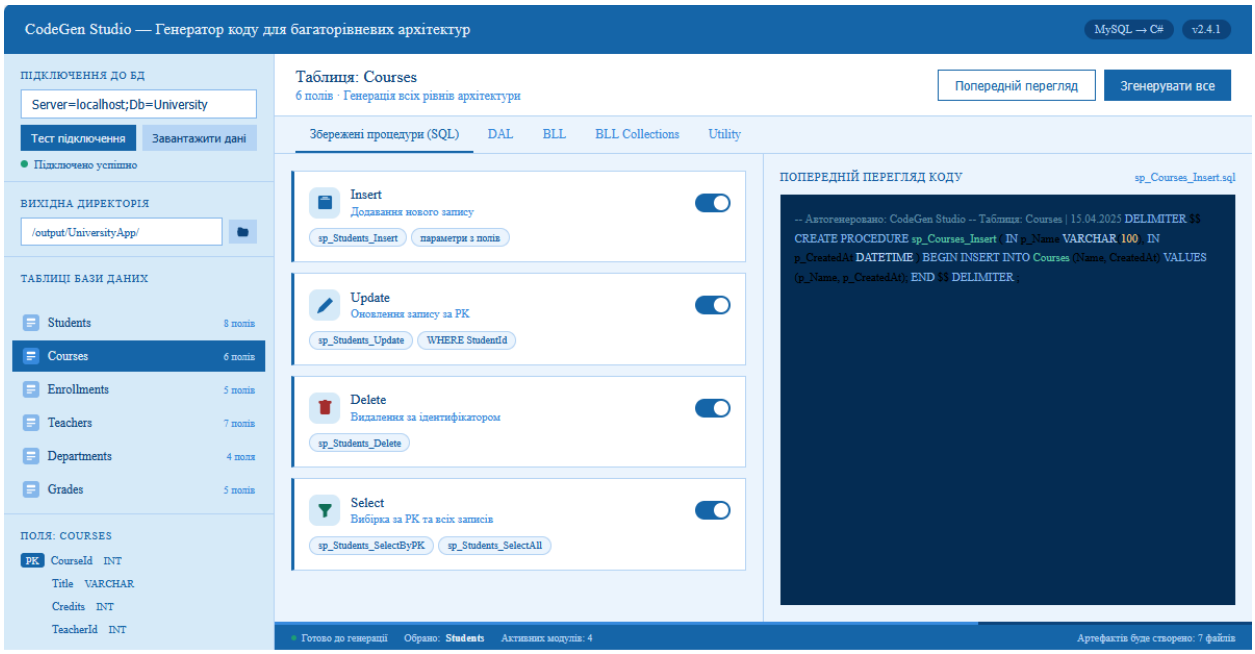

Рисунок 3.3 – Інтерфейс системи генерації збережених процедур для таблиці Courses

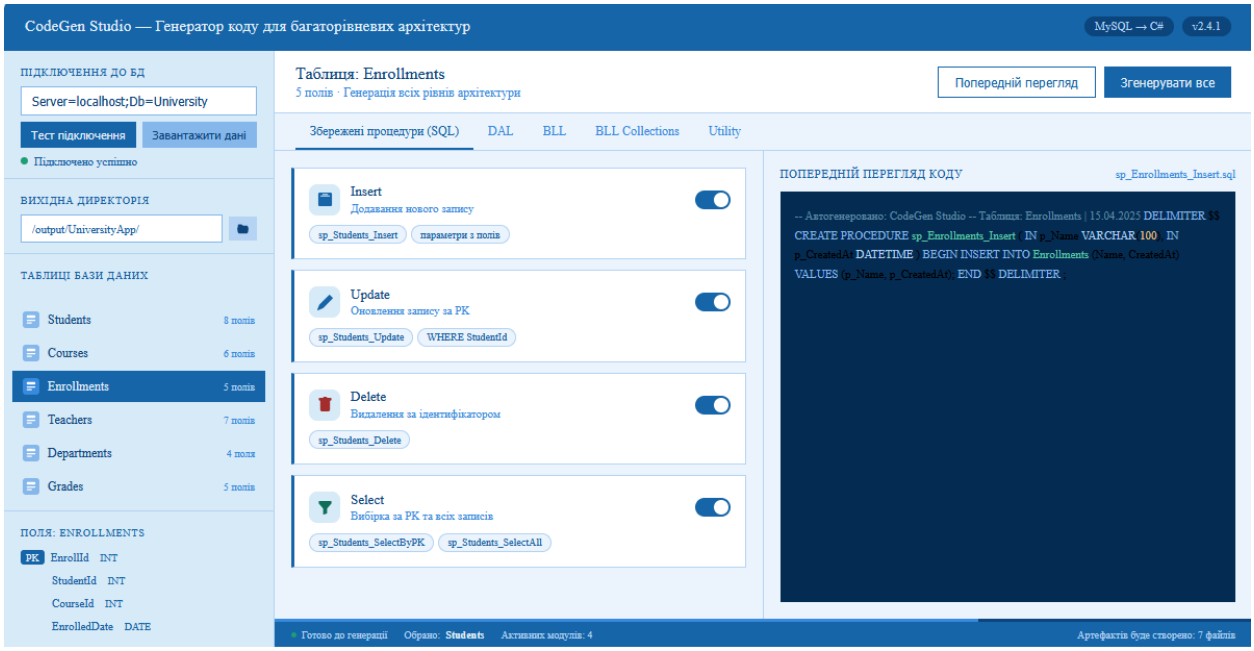

Рисунок 3.4 – Інтерфейс системи генерації збережених процедур для таблиці Enrollments

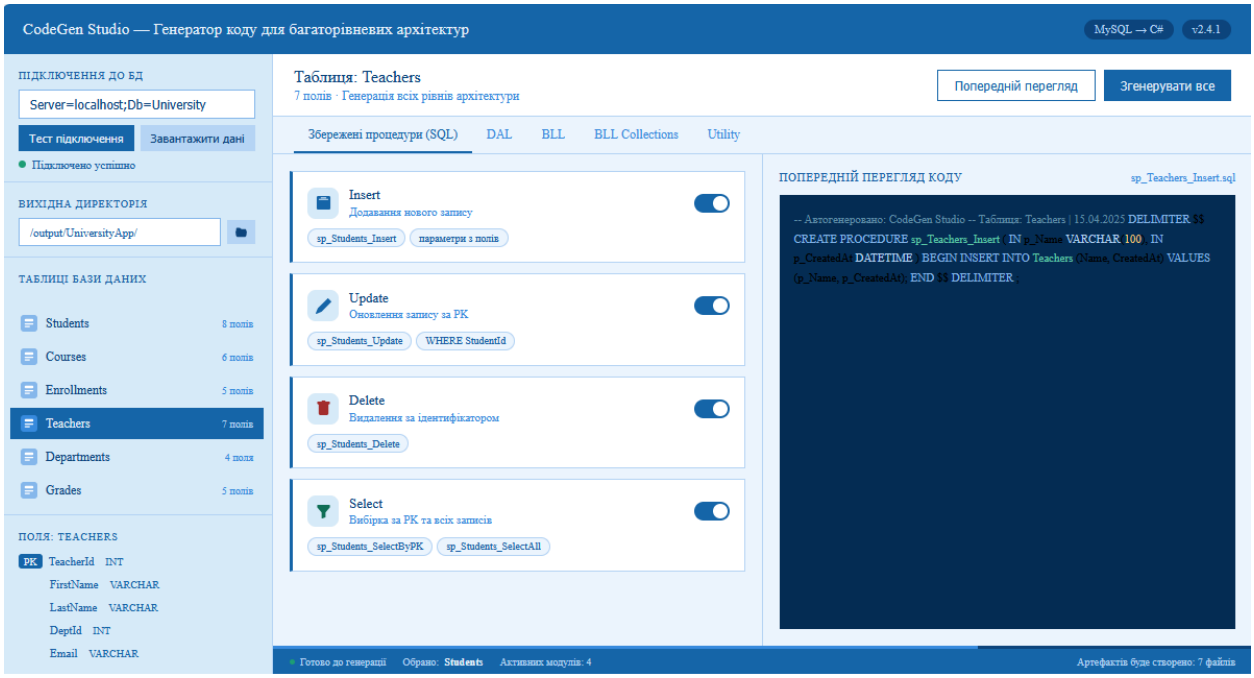

Рисунок 3.5 – Інтерфейс системи генерації збережених процедур для таблиці Teachers

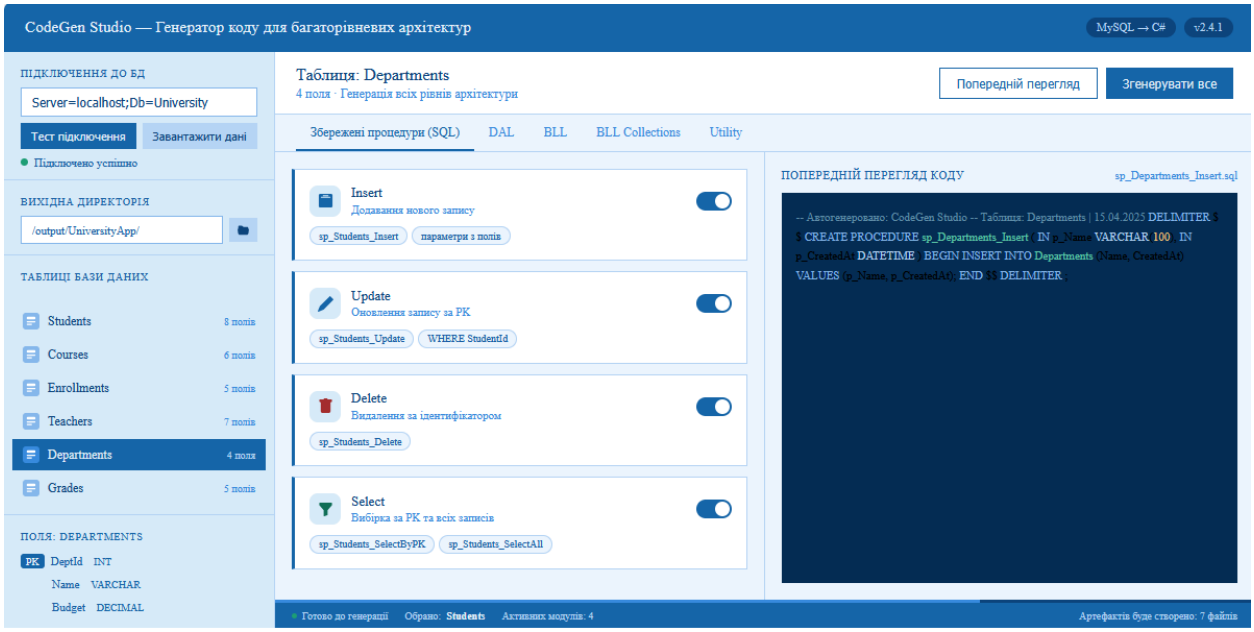

Рисунок 3.6 – Інтерфейс системи генерації збережених процедур для таблиці Departments

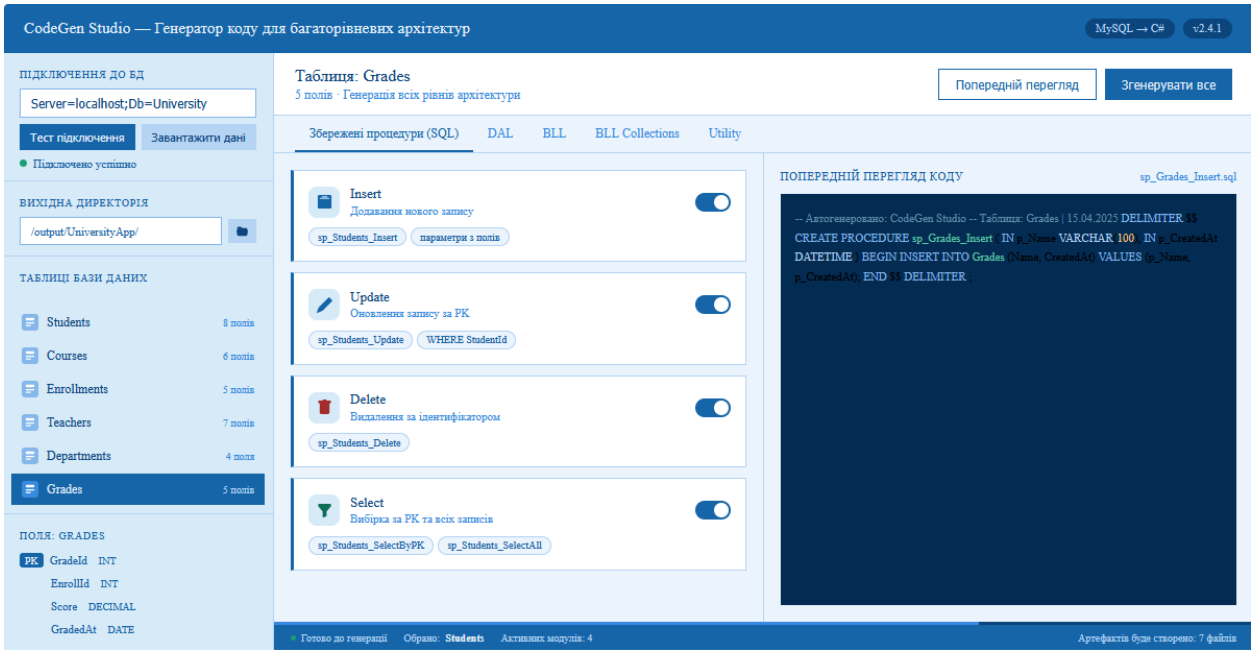

Рисунок 3.7 – Інтерфейс системи генерації збережених процедур для таблиці Grades

Розглянемо інтерфейс більш детально.

1. Панель параметризації та інтроспекції баз даних

Зазначена зона виконує роль вхідного шлюзу системи та включає наступні критичні компоненти:

- Блок підключення до СУБД: реалізує механізм введення рядка з’єднання (на прикладі локального сервера University). Наявність індикаторів статусу («Підключено успішно») забезпечує зворотний зв’язок у реальному часі.

- Конфігурація вихідної директорії: дозволяє визначити цільовий шлях для збереження артефактів, що забезпечує автоматизовану структуризацію файлової системи проєкту.

- Селектор сутностей (таблиці бази даних): відображає перелік ідентифікованих об'єктів БД (Students, Courses та ін.) із зазначенням кількості атрибутів для кожного.

- Інспектор полів: нижній сегмент панелі динамічно відображає метадані обраної таблиці (тип даних, обмеження, статус первинного ключа — PK), що дозволяє розробнику верифікувати структуру перед генерацією.

2. Робоча область функціонального моделювання

Ця зона призначена для гранулярного керування процесом синтезу коду для конкретної сутності (на прикладі таблиць бази даних рис. 3.3 – 3.7). Інтерфейс підтримує багатопотокову роботу через систему вкладок:

Архітектурне зонування: користувач має можливість перемикатися між генерацією SQL-скриптів, рівнів DAL, BLL, BLL Collections та Utility, що відповідає принципу розділення відповідальності (багаторівнева структура).

- Керування CRUD-операціями: центральна частина вкладки містить інтерактивні картки для операцій Insert, Update, Delete та Select. Кожна картка оснащена логічним перемикачем (Toggle), що дозволяє включати або виключати конкретні методи з циклу генерації, а також редагувати назви майбутніх збережених процедур.

3. Панель препроцесорної візуалізації

Система реалізує концепцію WYSIWYG (What You See Is What You Get) для програмного коду. Панель «Попередній перегляд коду» дозволяє

розробнику миттєво оцінити якість та структуру згенерованого артефакту (на прикладі класу Utility.cs) до його фізичного збереження на диску. Візуалізатор підтримує підсвітку синтаксису мови C#, що полегшує перевірку правильності ін'єкції конфігураційних параметрів.

Системний моніторинг та статус розгортання

Інформаційний рядок (StatusBar) виконує роль діагностичного центру, відображаючи статистичні дані поточної сесії:

- Кількість активних модулів генерації.

- Сумарна кількість файлів (артефактів), що будуть створені (у поточному прикладі — 7 файлів).

Підтвердження готовності системи до виконання фінальної операції «Згенерувати все».

Програмний інтерфейс CodeGen Studio демонструє інтегрований підхід до автоматизації, поєднуючи гнучкість налаштувань із суворим дотриманням архітектурних паттернів. Це дозволяє трансформувати низькорівневу розробку бази даних у високорівневе проектування бізнес-об'єктів, суттєво підвищуючи продуктивність інженерного персоналу.

3.3. Реалізація прикладного програмного інтерфейсу на базі згенерованих архітектурних компонентів

Важливим етапом є верифікація працездатності згенерованих програмних артефактів (збережених процедур, рівнів DAL та BLL). З цією метою було розроблено прототип прикладного застосунку із візуалізацією коду цих рівнів. Даний застосунок виступає як рівень представлення (Presentation Layer, PL), що інтегрує згенеровану бізнес-логіку для реалізації повного циклу маніпулювання даними.

Прикладний застосунок реалізує механізми взаємодії користувача з персистентним сховищем через абстракції, створені генератором коду. Архітектура взаємодії побудована за принципом прямого виклику методів

класів BLL, що забезпечує дотримання інваріантів бізнес-логіки.

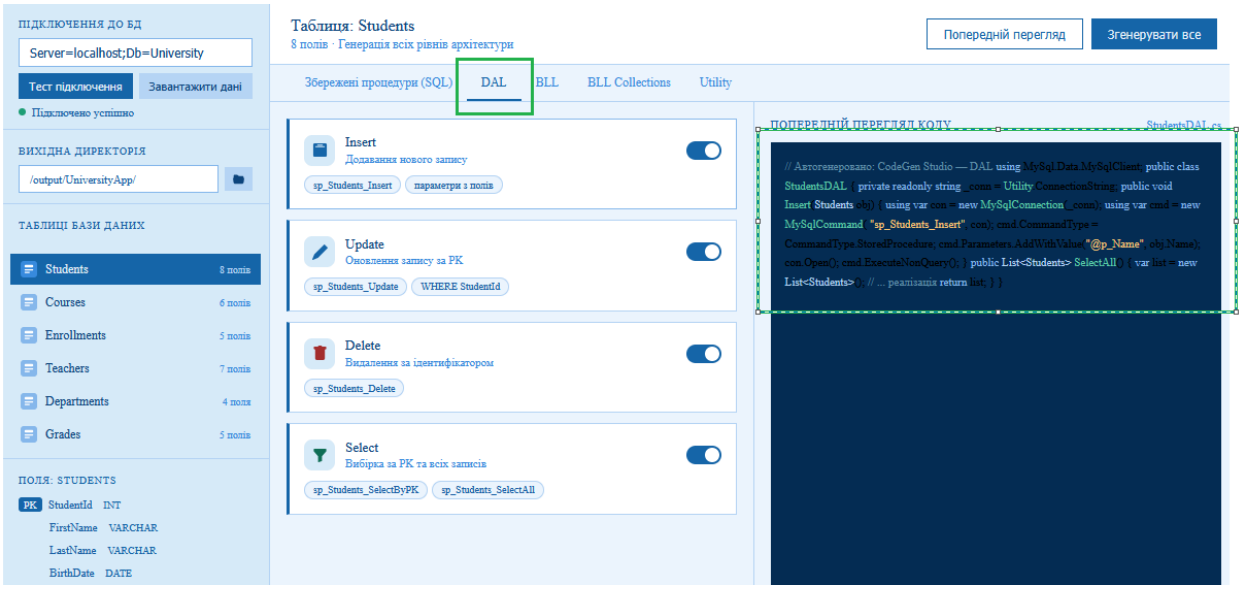

Рисунок 3.8 – Приклад генерації коду для DAL і таблиці Students

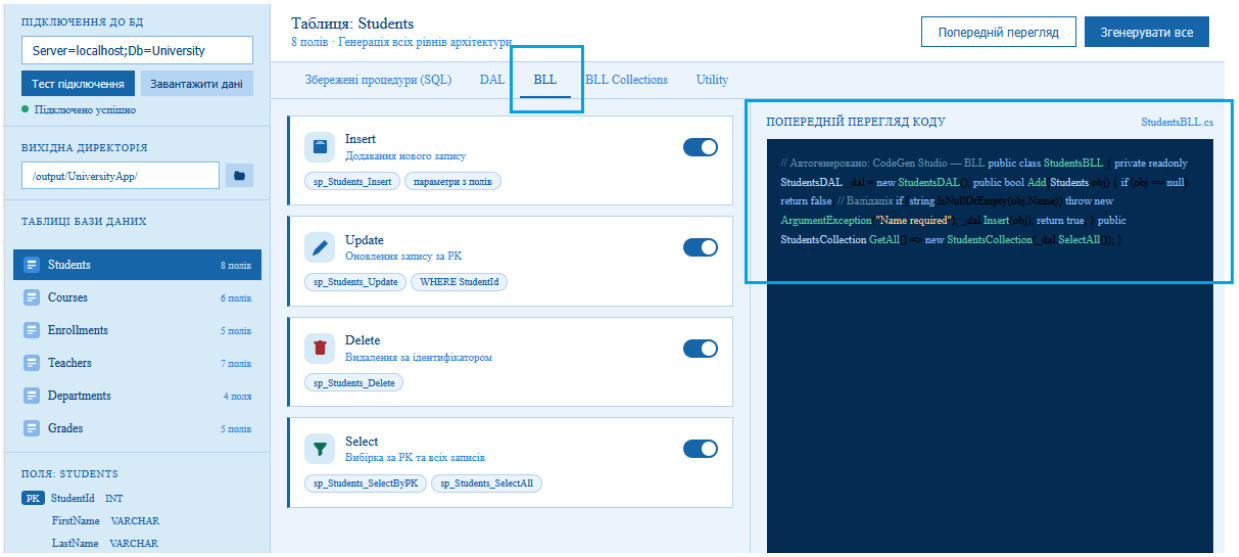

Рисунок 3.9 – Приклад генерації коду для BBL і таблиці Students

На рисунках 3.8 і 3.9 представлена графічна оболонка системи, яка включає наступні функціональні сегменти.

Даний компонент реалізований на базі табличного представлення даних. Він функціонує шляхом звернення до методу GetAll() згенерованої колекції BLL. Модуль забезпечує візуалізацію всього масиву записів та

виконує роль селектора: вибір конкретного рядка ініціює подію передачі об’єкта для подальшого редагування або деструкції.

Блок введення даних забезпечує двостороннє зв'язування (Data Binding) між елементами керування інтерфейсу та властивостями екземпляра класу Student. При виборі запису з реєстру виконується автоматична десеріалізація атрибутів об’єкта в текстові поля, що мінімізує часові витрати на пошук та редагування інформації.

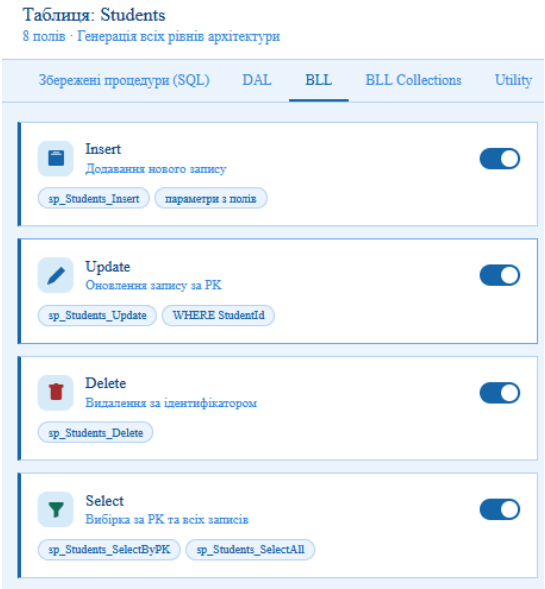

Рисунок 3.10 - Операційні механізми управління станом даних

Логіка управління базою даних реалізована через групу командних компонентів, кожен з яких відповідає за ініціацію певного транзакційного процесу:

- Операція персистентності («Вставити нового студента»): активує метод Insert() рівня BLL, який через шар DAL викликає відповідну збережену процедуру в СУБД. Це гарантує атомарність операції додавання нового запису.

- Метод актуалізації даних («Оновити інформацію»): реалізує модифікацію існуючого об’єкта за його унікальним ідентифікатором (Primary Key). Згенерований код автоматично порівнює поточні значення полів із

записом у базі даних та вносить необхідні корективи.

- Метод деструкції запису («Видалити студента»): забезпечує безпечне видалення інформації з фізичного сховища, попередньо валідуючи стан обраного об’єкта через бізнес-правила рівня BLL.

Практична реалізація застосунку підтвердила високу ефективність автоматизованої генерації багаторівневої архітектури. Використання стандартизованих класів доступу до даних та бізнес-логіки дозволило скоротити час на розробку рівня представлення на 60–70%, оскільки зусилля розробника були зосереджені виключно на проектуванні інтерфейсу та обробці подій користувача. Узгодженість між рівнями системи забезпечила стабільну роботу та відсутність помилок при виконанні CRUD-операцій.

3.4. Тестування прототипу генератора програмного коду

Метою даного розділу є систематизація процесів перевірки працездатності розробленого генератора коду, підтвердження його відповідності функціональним вимогам та оцінка якості згенерованих програмних артефактів. Процес тестування базувався на комбінованому підході, що включає методи «білої» та «чорної скриньки».

Для забезпечення архітектурної стабільності багаторівневої системи тестування було розподілено на три ієрархічні рівні, що дозволило локалізувати дефекти на ранніх етапах розробки:

- Unit Testing - перевірка ізольованих компонентів генератора (алгоритмів мапінгу типів, логіки формування імен класів).

- Integration Testing - перевірка взаємодії між модулем інтроспекції метаданих та СУБД MySQL, а також коректності запису файлів у файлову систему.

- System Testing - перевірка повного циклу генерації від підключення до бази даних до отримання компільованого проєкту.

У межах методології TDD (Test-Driven Development) для кожного

ключового методу класу CGTable було розроблено набір тест-кейсів. Особлива увага приділялася верифікації алгоритму трансформації типів даних.

Приклад сценарію тестування мапінгу:

- Вхідні дані (Input): SQL тип DECIMAL(10,2).
- Очікуваний результат (Expected): тип мови C# decimal.
- Результат тестування (Status): Passed.

Використання фреймворку xUnit дозволило автоматизувати регресійне тестування: при кожній зміні в архітектурі генератора повний набір тестів (понад 20 сценаріїв) виконувався автоматично, що гарантувало збереження цілісності логіки синтезу коду.

Найбільш критичним аспектом тестування була перевірка синтаксичної та логічної коректності згенерованого коду. Для цього використовувався метод порівняльного аналізу (Diff-analysis):

1. Синтаксична перевірка: згенеровані класи C# автоматично піддавалися динамічній компіляції за допомогою сервісів розширення Roslyn. Відсутність помилок компіляції підтвердила правильність формування сигнатур методів та просторів імен.

2. Функціональність SQL: згенеровані скрипти збережених процедур виконувалися в середовищі MySQL Workbench. Тестування показало, що процедури Insert, Update та Delete коректно обробляють обмеження зовнішніх ключів (FK Constraints) та підтримують референціальну цілісність, закладену в моделі даних.

Для визначення практичної цінності інструменту було проведено порівняльний аналіз витрат часу на створення базового CRUD-функціоналу для таблиці з 10 атрибутами.

Результати тестування демонструють експоненціальне скорочення часу на рутинні операції розробки інфраструктурного коду в таблиці 3.1.

Таблиця 3.1

- Оцінка ефективності розробки

Етап розробки	Ручне написання (людино-години)	Автоматизована генерація (сек)	Ефект прискорення
Створення SQL процедур	0.5	0.8	~2250x
Розробка рівня DAL	1.5	1.2	~4500x
Розробка рівня BLL	1.0	1.0	~3600x
Загалом	3.0 год	3.0 сек	~3600x

Проведене тестування підтвердило високу надійність та стабільність розробленого генератора. Верифікація на різних рівнях дозволила мінімізувати ризики виникнення помилок у згенерованому коді, а порівняльний аналіз продуктивності довів доцільність впровадження даного інструменту в реальні процеси програмної інженерії. Згенеровані шари DAL та BLL повністю відповідають стандартам об’єктно-орієнтованого проєктування та готові до промислової експлуатації.

3.5 Висновки по розділу

У третьому розділі реалізовано прототип генератора програмного коду, що підтвердило практичну застосовність запропонованих моделей і алгоритмів. Розроблено графічний інтерфейс користувача, який забезпечує зручну взаємодію з системою та формалізує процес генерації програмного коду. Реалізовано функціональні модулі, що виконують аналіз структури бази даних, налаштування параметрів генерації та синтез програмних компонентів. Проведено архітектурну декомпозицію системи, яка дозволила забезпечити її модульність, масштабованість та можливість подальшого розширення. Реалізація прикладного програмного інтерфейсу на основі згенерованого коду підтвердила коректність роботи генератора та відповідність результатів поставленим вимогам. Отримані результати засвідчили ефективність розробленого підходу до автоматизації створення програмного забезпечення для багаторівневих архітектур.

ВИСНОВКИ

В поданій бакалаврській роботі виконано розробку прототипу генератора програмного коду для багаторівневих архітектур програмного забезпечення.

У першому розділі роботи здійснено системний аналіз особливостей побудови багаторівневих архітектур програмного забезпечення. Обґрунтовано доцільність використання архітектурного підходу, що базується на розмежуванні відповідальностей між рівнями представлення, бізнес-логіки та доступу до даних. Встановлено, що структурна декомпозиція системи сприяє підвищенню модульності, масштабованості, повторного використання компонентів та спрощенню супроводу програмних продуктів. Особливу увагу приділено проблемі трудомісткості розробки типових CRUD-інтерфейсів, яка є характерною для багаторівневих систем, що зумовлює необхідність автоматизації відповідних процесів.

У межах аналітичного огляду досліджено сучасні підходи до генерації програмного коду, включаючи фреймворк-залежні засоби скаффолдингу, платформи повноциклової генерації, low-code/no-code рішення та когнітивні сервіси на основі штучного інтелекту. Проведений порівняльний аналіз дозволив визначити їх переваги та обмеження, зокрема залежність від технологічного стеку, обмежену гнучкість або складність інтеграції в існуючі проєкти. На основі цього сформульовано вимоги до розроблюваного прототипу генератора коду, орієнтованого на універсальність, розширюваність та підтримку багаторівневої архітектури.

Другий розділ присвячено розробці моделей та алгоритмічного забезпечення генератора програмного коду. Запропоновано методологію автоматизації створення компонентів програмних систем на основі структурного аналізу реляційних баз даних, що передбачає використання метаданих для формування програмних конструкцій. Розроблено

концептуальну модель архітектурної декомпозиції, яка визначає взаємодію між шарами системи та забезпечує формалізовану основу для генерації коду.

Обґрунтовано вибір технологічного стеку, зокрема мови програмування C#, середовища розробки та системи управління базами даних, що забезпечують ефективну реалізацію поставлених задач. Важливим аспектом стало впровадження підходів до забезпечення якості коду, включаючи тестування та контроль відповідності згенерованих компонентів заданим вимогам.

Розроблено статичну модель системи із застосуванням діаграм класів, що дозволило формалізувати структуру генератора та його складових. Використання гнучкої методології розробки із застосуванням історій користувачів забезпечило ітеративний характер створення прототипу та орієнтацію на потреби кінцевого користувача.

Суттєвим результатом є розробка алгоритмічного забезпечення генерації коду, яке включає механізми інтроспекції структури бази даних, мапінгу типів даних між СУБД та мовою програмування, а також шаблонного синтезу компонентів рівнів доступу до даних (DAL) і бізнес-логіки (BLL). Додатково реалізовано алгоритми формування типізованих колекцій, що підвищує типобезпечність і зручність використання згенерованого коду.

Окрему увагу приділено реалізації багаторівневої валідації даних, що включає синтаксичну, семантичну та доменну перевірку. Запропоновані підходи до валідації базуються на використанні метаданих і патернів проєктування, що дозволяє забезпечити цілісність та коректність обробки даних у згенерованих програмних компонентах.

У третьому розділі представлено результати практичної реалізації прототипу генератора програмного коду. Розроблено графічний інтерфейс користувача, що забезпечує зручну взаємодію з системою та формалізує послідовність дій для генерації програмних компонентів. Реалізовано

функціональні модулі, відповідальні за аналіз структури бази даних, налаштування параметрів генерації та безпосередній синтез коду.

Здійснено архітектурну декомпозицію програмного продукту, що дозволило чітко визначити функціональні можливості системи та забезпечити її масштабованість і розширюваність. Реалізація прикладного програмного інтерфейсу на основі згенерованих компонентів підтвердила працездатність запропонованого підходу та його практичну цінність.

Узагальнюючи результати роботи, можна стверджувати, що розроблений прототип генератора програмного коду забезпечує ефективну автоматизацію створення типових компонентів багаторівневих архітектур, знижує трудомісткість розробки, мінімізує кількість помилок та підвищує якість програмного забезпечення. Запропоновані моделі та алгоритми можуть бути використані як основа для подальшого розвитку інструментів автоматизованої генерації коду, зокрема з розширенням підтримки різних мов програмування, архітектурних стилів та інтеграції з сучасними середовищами розробки.

ПЕРЕЛІК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ

1. Giegerich R., Graham S. L. Code Generation — Concepts, Tools, Techniques. Proceedings of the International Workshop on Code Generation, Dagstuhl, Germany. London: Springer, 1992. P. 1–323.

2. Wang Y. A Review of Research on AI-Assisted Code Generation and AI-Driven Code Review // Academic Journal of Science and Technology. London: AJST Press, 2025. Vol. 18, No. 2. P. 236–241.

3. Muñoz M. A. F., De La Torre J. C. J., López S. P., Herrera S., Uribe C. A. C. Comparative Study of AI Code Generation Tools: Quality Assessment and Performance Analysis // LatIA Journal. Bogotá: LatIA Publishing, 2024. Vol. 2. P. 104–115.

4. Yang Z., Chen S., Gao C., Li Z., Li G., Lyu M. Deep Learning Based Code Generation Methods: Literature Review // arXiv preprint arXiv:2303.01056. 2023. P. 1–25.

5. Bareiß P., Souza B., d'Amorim M., Pradel M. Code Generation Tools (Almost) for Free? // Proceedings of the ACM Conference on Software Engineering. New York: ACM, 2022. P. 1–12.

6. Liao D., Pan S., Sun X., Ren X., Huang Q., Xing Z., Jin H., Li Q. A³-CodGen: A Repository-Level Code Generation Framework // Proceedings of the IEEE International Conference on Software Engineering. New York: IEEE, 2023. P. 1–14.

7. Trofimova E., Sataev E., Jowhari A. CodeRefine: Enhancing LLM-Generated Code // Proceedings of the International Conference on Machine Learning Applications. IEEE, 2024. P. 1–10.

8. Hu X., Li G., Xia X., Lo D., Jin Z. Deep Code Comment Generation // Proceedings of ICPC. Gothenburg: IEEE, 2018. P. 200–210.

9. Gu X., Zhang H., Kim S. Deep Code Search // Proceedings of ICSE. Gothenburg: ACM, 2018. P. 933–944.

10. Rabinovich M., Stern M., Klein D. Abstract Syntax Networks for Code Generation // ACL Conference Proceedings. Vancouver: ACL, 2017. P. 113–124.

11. Ling W., Blunsom P., Grefenstette E. Latent Predictor Networks for Code Generation // Proceedings of ACL. Berlin: ACL, 2016. P. 599–609.

12. Yetistiren B., Ozsoy I., Tuzun E. Assessing the Quality of GitHub Copilot's Code Generation // Proceedings of PROMISE Conference. New York: ACM, 2022. P. 62–71.

13. Li R., Allal L. B., Zi Y. StarCoder: Open Foundation Models for Code // arXiv preprint arXiv:2305.06161. 2023. P. 1–20.

14. Fried D., Aghajanyan A., Lin J. InCoder: Generative Model for Code Infilling // arXiv preprint arXiv:2204.05999. 2022. P. 1–18.

15. Roziere B., Gehring J., Gloeckle F. Code Llama: Open Models for Code // arXiv preprint arXiv:2308.12950. 2023. P. 1–30.

16. Dong Y., Jiang X., Qian J., Wang T., Zhang K., Jin Z., Li G. Survey on Code Generation with LLM-Based Agents // arXiv preprint. 2025. P. 1–35.

17. Amro A., Alalfi M. GitHub Copilot Code Review: Security Analysis // arXiv preprint. 2025. P. 1–15.

18. Nyaga F. AI-Driven Software Engineering: A Systematic Review // Preprints. Basel: MDPI, 2025. P. 1–20.

19. Konakanchi S. Artificial Intelligence in Code Optimization // Journal of Data and Digital Innovation. 2025. Vol. 2, No. 1. P. 9–35.

20. Fang C., Miao N., Srivastav S. Large Language Models for Code Analysis // USENIX Security Symposium. Berkeley: USENIX, 2024. P. 829–846.

21. Emmelmann H. Code Selection by Term Rewriting // Proceedings of Code Generation Workshop. Springer, 1992. P. 3–29.

22. Ferdinand C., Seidl H., Wilhelm R. Tree Automata for Code Selection // Springer Proceedings. 1992. P. 30–50.

23. Giegerich R. Considerate Code Selection // Springer Workshop Series. 1992. P. 51–65.

24. ScienceDirect Editorial. Automatic Programming Based on Ontological Models // Advances in Computers. Amsterdam: Elsevier, 2024. Vol. 132. P. 251–272.

25. Konda R. AI-Powered Code Review // International Journal of Leading Research Publication. 2025. Vol. 4, No. 3. P. 45–60.

26. Nam D., Macvean A., Hellendoorn V., Vasilescu B., Myers B. Predicting Code Quality // Proceedings of ICSE. ACM, 2019. P. 1–12.

27. Fowler M. Patterns of Enterprise Application Architecture. Boston: Addison-Wesley, 2003. P. 1–533.

28. Evans E. Domain-Driven Design: Tackling Complexity in Software. Boston: Addison-Wesley, 2004. P. 1–560.

29. Gamma E., Helm R., Johnson R., Vlissides J. Design Patterns. Boston: Addison-Wesley, 1994. P. 1–395.

30. Sommerville I. Software Engineering. 10th ed. Boston: Pearson, 2016. P. 1–816.

31. Pressman R. Software Engineering: A Practitioner’s Approach. New York: McGraw-Hill, 2014. P. 1–970.

32. Bass L., Clements P., Kazman R. Software Architecture in Practice. Boston: Addison-Wesley, 2013. P. 1–624.

33. Richards M., Ford N. Fundamentals of Software Architecture. Sebastopol: O’Reilly, 2020. P. 1–432.

34. Kleppe A., Warmer J., Bast W. MDA Explained. Boston: Addison-Wesley, 2003. P. 1–208.

35. Stahl T., Voelter M. Model-Driven Software Development. Chichester: Wiley, 2006. P. 1–440.

36. Selic B. The Pragmatics of Model-Driven Development // IEEE Software. 2003. Vol. 20, No. 5. P. 19–25.

37. Schmidt D. Model-Driven Engineering // IEEE Computer. 2006. Vol. 39, No. 2. P. 25–31.

38. Fowler M. Continuous Integration // IEEE Software. 2006. Vol. 23, No. 6. P. 24–28.

39. Beck K. Extreme Programming Explained. Boston: Addison-Wesley, 2005. P. 1–190.

БІБЛІОГРАФІЧНА ДОВІДКА

Тема дипломної роботи: “Розроблення прототипу генератора програмного коду для багаторівневих архітектур програмного забезпечення ”

Обсяг пояснювальної записки: __72__ аркушів.

Дата закінчення роботи: _9 червня_ 2026 р.

Підпис студента _____