Міністерство освіти і науки України
Харківський національний університет радіоелектроніки

Факультет комп'ютерних наук
(повна назва)

Кафедра програмної інженерії
(повна назва)

КВАЛІФІКАЦІЙНА РОБОТА
Пояснювальна записка

рівень вищої освіти другий (магістерський)

Дослідження ефективності та практичності інструментів для кодогенерації .NET застосунків з використанням графічних представлень
(тема)

Виконав:
студент 2 курсу, групи ІПЗм-22-6

Денисюк В.М
(прізвище, ініціали)

Спеціальність 121 – Інженерія програмного забезпечення
(код і повна назва спеціальності)
Тип програми освітньо-наукова

Керівник проф. Четвериков Г.Г.
(посада, прізвище, ініціали)

Допускається до захисту
Зав. кафедри _____ З.В.Дудар
(підпис) (прізвище, ініціали)

2024 р.

Харківський національний університет радіоелектроніки

Факультет Комп'ютерних наук

Кафедра Програмної інженерії

Рівень вищої освіти другий (магістерський)

Спеціальність 121 – Інженерія програмного забезпечення
(код і повна назва)

Тип програми освітньо-наукова програма

Освітня програма Інженерія програмного забезпечення
(повна назва)

ЗАВДАННЯ

НА КВАЛІФІКАЦІЙНУ РОБОТУ

студентові Денисюку Владиславу Михайловичу
(прізвище, ім'я, по батькові)

1. Тема роботи: «Дослідження ефективності та практичності інструментів для кодогенерації .NET застосунків з використанням графічних представлень.»

Затверджена наказом університета від 29.03.2024 р. № 250Ст

2. Термін подання студентом роботи до екзаменаційної комісії 19.06.2024р.

3. Вихідні дані до роботи: кодогенерація, процедура, графічне представлення, ефективність алгоритму, метод роботи, програмна реалізація, пояснювальна записка.

4. Перелік питань, що потрібно опрацювати в роботі: мета роботи, аналіз предметної галузі, постановка задачі, генетичні алгоритми,, дослідження способів генерації вихідного коду, моделювання даних, послідовні нейронні мережі.

КАЛЕНДАРНИЙ ПЛАН

№	Назва етапів роботи	Термін виконання етапів роботи	Примітка
1	Аналіз предметної галузі та постановка задачі	30.03 – 14.04.24	*виконано*
2	Аналіз та вибір API для дослідження	15.04 – 24.04.24	*виконано*
3	Аналіз та моделювання предметної області	25.04 – 28.04.24	*виконано*
4	Планування експериментів	29.04 – 08.05.24	*виконано*
5	Програмна реалізація кожного з обраних для дослідження API	09.05 – 19.05.24	*виконано*
6	Експериментальні дослідження	20.05 – 25.05.24	*виконано*
7	Аналіз результатів експериментальних досліджень та розробка рекомендацій	25.05 – 28.05.24	*виконано*
8	Написання та оформлення статті та тез доповіді	28.05 – 31.05.24	*виконано*
9	Підготовка пояснювальної записки	01.06 – 09.05.24	*виконано*
10	Підготовка презентації та доповіді	10.06 – 12.06.24	*виконано*
11	Нормоконтроль	13.06 – 14.06.24	*виконано*
12	Рецензування	14.06 – 15.06.24	*виконано*
13	Занесення диплома в електронний архів	16.06.2024	*виконано*
14	Попередній захист	17.06.2024	*виконано*
15	Допуск до захисту у зав. кафедри	18.06.2024	*виконано*

Дата видачі завдання *« 30 »* *березня* *2024 р.*

Студент _____ Денисюк В.М
(підпис)

Керівник роботи _____ проф. Четвериков Г.Г.
(підпис) (посада, прізвище, ініціали)

РЕФЕРАТ / ABSTRACT

Пояснювальна записка: 55 с., 12 рис., 4 додатків, 10 джерел.

ГРАФІЧНІ ПРЕДСТАВЛЕННЯ, ЕФЕКТИВНІСТЬ, КОДОГЕНЕРАЦІЯ, ПРАКТИЧНІСТЬ, .NET,

Об’єктом дослідження є ефективність та практичність інструментів для кодогенерації .NET застосунків з використанням графічних представлень.

Метою роботи є аналіз та порівняння різних засобів для автоматизованої генерації коду, спрямованих на полегшення розробки .NET додатків з використанням графічних елементів інтерфейсу.

У процесі дослідження використовуються методи оцінки продуктивності, аналізу зручності використання та порівняльного аналізу вихідного коду, створеного різними інструментами.

В результаті дослідження визначено оптимальні інструментальні рішення для ефективної та практичної кодогенерації .NET застосунків з використанням графічних представлень.

CODE GENERATION, .NET, GRAPHICS, EFFICIENCY, PRACTICALITY.

The object of the study is the effectiveness and practicality of tools for code generation of .NET applications using graphical representations.

The purpose of the work is to analyze and compare various tools for automated code generation aimed at facilitating the development of .NET applications using graphical interface elements.

The research process uses methods of performance evaluation, usability analysis and comparative analysis of source code generated by different tools.

As a result of the study, optimal instrumental solutions for effective and practical code generation of .NET applications using graphical representations were determine

Я, Денисюк Владислав Михайлович, студент гр.ІПЗм-22-6, здобувач вищої освіти на другому (магістерському) рівні кафедри «Програмна інженерія», заявляю: моя робота, що буде представлена для публічного захисту, виконана самостійно, в ній не містяться елементи плагіату і вона може бути опублікована в електронному архіві відкритого доступу ElAr KhNURE. Усі запозичення з друкованих та електронних джерел мають відповідні посилання.

Я ознайомлений з діючим положенням «Про протидію академічному плагіату в ХНУРЕ», згідно з яким виявлення плагіату є підставою для відмови до допуску кваліфікаційної роботи до захисту та застосування дисциплінарних заходів.

ЗМІСТ

ВСТУП

В сучасному світі інформаційних технологій розробка програмного забезпечення стає все складнішою і вимагає від розробників швидкості, ефективності та високої якості коду. Одним із ключових аспектів удосконалення цього процесу є використання інструментів для кодогенерації, які спрощують створення програмного забезпечення і забезпечують більшу продуктивність розробників.

Ця кваліфікаційна робота присвячена дослідженню ефективності та практичності інструментів для кодогенерації .NET застосунків, з фокусом на використанні графічних представлень для полегшення процесу розробки. З поглибленням технологій та зростанням складності вимог до програмного забезпечення, важливо розглядати нові підходи, які дозволяють розробникам швидше та ефективніше створювати високоякісні додатки.

В ході цього дослідження буде проведено аналіз різноманітних інструментів для кодогенерації .NET застосунків, враховуючи їхні переваги та недоліки. Особлива увага буде приділена інструментам, що використовують графічні представлення для візуалізації процесу розробки, з метою визначення їхньої придатності та ефективності у порівнянні з традиційними методами програмування.

Кваліфікаційна робота має на меті не лише розглядати існуючі інструменти, але й визначити перспективи вдосконалення та подальшого використання графічних засобів для кодогенерації .NET застосунків у сучасній розробці програмного забезпечення. В контексті стрімкого розвитку технологій та постійного поновлення вимог до програмного забезпечення, інструменти для кодогенерації відіграють важливу роль у спрощенні рутинних завдань, що дозволяє розробникам більше уваги приділяти творчим та стратегічним аспектам роботи. Структуроване порівняння різних інструментів, їхніх можливостей та обмежень допоможе визначити оптимальний вибір для конкретного випадку використання. Зокрема, дослідження орієнтується на .NET-екосистему, оскільки це один із

ключових фреймворків для створення масштабованих та ефективних програмних рішень.

Задачею цього дослідження також є визначення того, наскільки графічні представлення можуть полегшити розуміння коду та сприяти зниженню його складності. Аналіз інтерфейсів для візуалізації програмного коду відкриє можливості для покращення якості та швидкості розробки, а також допоможе визначити питання, пов'язані з прийняттям таких інструментів у сучасному програмуванні. Усе це спрямовано на створення комплексного уявлення про ефективність та практичність інструментів для кодогенерації .NET застосунків із застосуванням графічних представлень.

1 АНАЛІЗ ТА ОГЛЯД ВИХІДНИХ МАТЕРІАЛІВ

1.1 Аналіз предметної галузі

У сучасній розробці програмного забезпечення шаблони коду відіграють значущу роль у спрощенні рутинних завдань та підвищенні ефективності розробників. Підпункт цього аналізу буде присвячений дослідженню та порівнянню шаблонів коду на платформі .NET, таких як T4 (Text Template Transformation Toolkit) та Razor. Розглядатимуться їхні можливості, сфери застосування та вплив на продуктивність та якість згенерованого коду.

В контексті аналізу методів кодогенерації .NET-застосунків важливим аспектом є вивчення та порівняння ролі шаблонів коду, особливо T4 (Text Template Transformation Toolkit) та Razor. Ці інструменти надають розробникам можливість ефективно генерувати код на основі певних шаблонів, але вони мають відмінності у функціоналі та спрямовані на різні завдання.

T4 (Text Template Transformation Toolkit) є одним із основних інструментів для генерації коду в середовищі .NET. Використовуючи текстові шаблони, T4 дозволяє розробникам вбудовувати код C# або VB.NET у текстові файли. Завдяки інтеграції з Visual Studio, T4 надає можливість автоматичної генерації коду під час компіляції або редагування відповідного шаблону. Розробники можуть використовувати T4 для широкого спектру завдань, від автоматичного створення DTO (Data Transfer Objects) до генерації шаблонів коду для стандартів оформлення.

Razor, в першу чергу використовуваний для створення веб-сторінок в ASP.NET, також може бути використаний як інструмент для кодогенерації. Його синтаксис більше спрямований на веб-розробку та включає в себе можливості для вбудовання C#-коду прямо в HTML-файли. Це дає розробникам можливість генерувати HTML-код або інші текстові файли, використовуючи схожий синтаксис, що забезпечує чітку інтеграцію логіки та представлення.

При порівнянні T4 і Razor важливо враховувати їхні сфери застосування. T4 найчастіше використовується для генерації будь-якого типу коду, тоді як Razor більше спрямований на веб-розробку та генерацію вмісту для веб-сторінок. T4

може бути використаний для різноманітних задач у проекті, включаючи створення додаткових класів, конфігураційних файлів чи тестових скриптів.

Важливо також враховувати різницю в синтаксисі: T4 використовує текстові шаблони, в той час як Razor більше орієнтований на вбудовання коду в HTML. Вибір між цими інструментами повинен базуватися на конкретних потребах проекту та предметній області його використання.

Текстовий шаблонний механізм T4 надає значні можливості для генерації коду, але також має свої обмеження. На перший погляд, він може здатися простим, але при розвитку проекту можуть виникати складнощі в управлінні великими обсягами шаблонного коду. Однак завдяки активній спільноті та добре розробленій інтеграції з Visual Studio, розвиток та утримання проекту з використанням T4 залишаються реалізованими.

Зокрема, у контексті веб-розробки, Razor пропонує гнучкість та зручність в інтеграції логіки та представлення. Вбудований синтаксис дозволяє розробникам визначати умови, цикли та використовувати змінні безпосередньо в HTML. Однак, в порівнянні з T4, в якому шаблони можуть бути менш зв'язаними з контекстом використання, у випадку Razor більше слід враховувати обмеження на використання в рамках веб-технологій.

Обидва підходи можуть ефективно інтегруватися з іншими інструментами розробки. T4 має велику кількість плагінів та розширень для Visual Studio, що полегшує роботу з цим інструментом. Razor, з іншого боку, впроваджений у стандартні засоби розробки ASP.NET та може бути використаний як частина розширених рішень для створення веб-додатків. У випадку процедурного генерування коду надалі використовуватися буде Roslyn API[1].

1.2 Використання Roslyn API для кодогенерації

Roslyn, як відкритий компілятор від Microsoft, відкриває широкі можливості для аналізу та генерації коду на платформі .NET. Цей підпункт буде приділяти увагу аспектам використання Roslyn API у розробці програмного забезпечення, зокрема аналізу та трансформації синтаксичних дерев коду. Розглядатиметься, як

розробники можуть використовувати ці можливості для автоматизації завдань, які зазвичай вимагають значної кількості коду вручну. Компоненти Roslyn наведено на рисунку 1.1

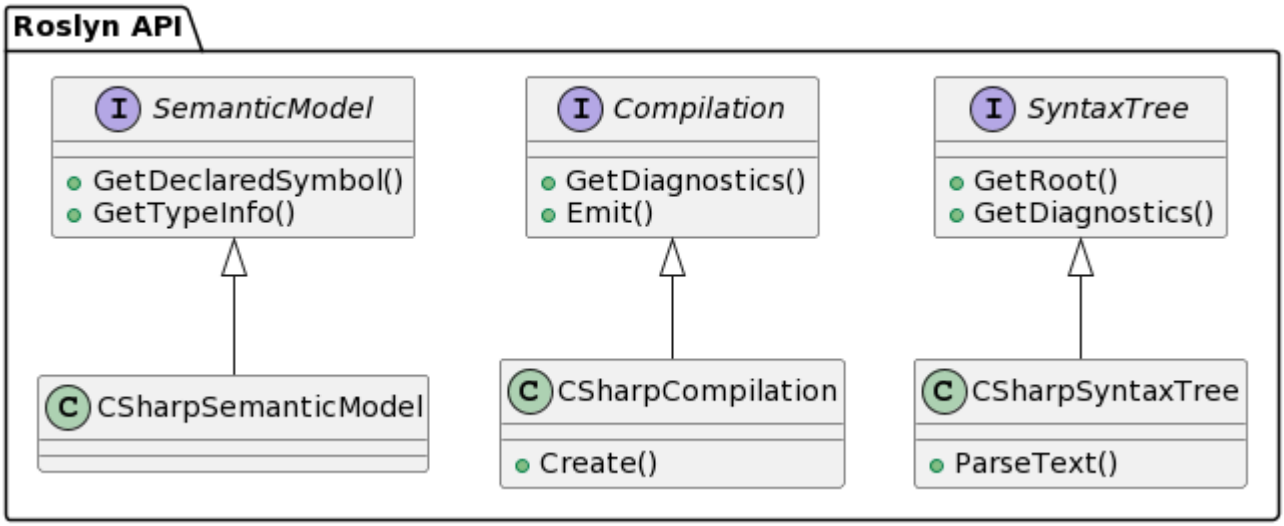

Рисунок 1.1 Компоненти Roslyn API

Використання Roslyn API для кодогенерації в .NET є однією з передових та потужних стратегій, оскільки це відкриває доступ до внутрішньої структури та аналізу синтаксичного дерева коду мов C# та VB.NET. Roslyn може бути використаний для створення власних інструментів, які здатні аналізувати, змінювати та генерувати код динамічно, в залежності від користувацького вводу, в нашому випадку, використовуючи діаграму UML. Синтаксичне дерево відповідає за створення залежностей та відображає актуальні дані про структуру коду, його характеристики. Там зберігаються усі семантичні деталі реалізації та усі назви методів та властивостей.

Можливості використання Roslyn для кодогенерації:

– аналіз та модифікація синтаксичного дерева – Roslyn надає API для аналізу та модифікації синтаксичного дерева коду. Розробники можуть переглядати всі складові програми – класи, методи, змінні, тощо – і вносити власні зміни. Це дозволяє генерувати код, додавати нові функції або автоматично впроваджувати зміни;

– генерація коду за допомогою Roslyn дає можливість генерувати новий код на основі абстракцій синтаксичного дерева. Це корисно при автоматизації створення певних шаблонів коду, патернів або навіть цілих класів. Розробники можуть динамічно генерувати код під час виконання програми або на етапі компіляції;
– плагіни та розширення – Roslyn розроблено з урахуванням створення плагінів та розширень для різних розробничих середовищ, таких як Visual Studio. Розробники можуть створювати власні інструменти для підтримки кодогенерації, що дозволяє вбудовувати їхні можливості безпосередньо в середовище розробки.

Розглянемо приклад використання Roslyn для динамічної генерації коду. Розробник може написати програму, яка аналізує існуючий код та автоматично додає методи або властивості до класів на основі певних умов чи конфігурацій. Це дозволяє здійснювати динамічні зміни в коді без його ручного втручання.

Обмеження використання Roslyn:

– складність використання – використання Roslyn вимагає глибокого розуміння структури мови програмування C# та VB.NET. Для новачків це може бути відносно складним завданням;
– потенційні проблеми з продуктивністю – несправна реалізація аналізаторів може призводити до проблем з продуктивністю, оскільки аналіз та генерація коду може вимагати значних обчислювальних ресурсів;
– потреба в обов'язковому тестуванні – зміни, внесені до коду за допомогою Roslyn, можуть мати значний вплив на проект, тому тестування та валідація генерованого коду є критичним етапом в процесі використання цього підходу.

Загалом, використання Roslyn API для кодогенерації відкриває широкі можливості для розробників, але його використання вимагає уважної обробки та розуміння його функціоналу.

1.3 Інтеграція з іншими інструментами

Обидва підходи можуть ефективно інтегруватися з іншими інструментами розробки. T4 має велику кількість плагінів та розширень для Visual Studio, що полегшує роботу з цим інструментом. Razor, з іншого боку, впроваджений у стандартні засоби розробки ASP.NET та може бути використаний як частина розширених рішень для створення веб-додатків[2].

Roslyn API не тільки надає потужний засіб для аналізу та генерації коду, але також легко інтегрується з різноманітними іншими інструментами розробки та середовищами програмування, надаючи розробникам різноманітні можливості для автоматизації та покращення їхніх робочих процесів. Давайте розглянемо детальніше цей пункт:

- інтеграція з Visual Studio – Roslyn API має тісну інтеграцію з Visual Studio, офіційною інтегрованою середовище розробки (IDE) для мови програмування C#. Завдяки цій інтеграції, розробники можуть використовувати можливості аналізу та генерації коду прямо в середовищі розробки. Також можна створювати власні плагіни для розширення функціоналу Visual Studio за допомогою Roslyn;
- підтримка сторонніх інструментів – Roslyn API дозволяє інтегруватися з різноманітними сторонніми інструментами розробки. Наприклад, популярні інструменти для статичного аналізу коду, такі як ReSharper чи SonarQube, можуть використовувати Roslyn для здійснення своїх аналітичних операцій та надання розширених порад щодо якості коду;
- інтеграція з системами контролю версій – розробники можуть використовувати Roslyn для аналізу та генерації коду в контексті систем контролю версій, таких як Git або SVN. Це дозволяє автоматизувати певні завдання, такі як визначення відмінностей у коді або автоматичне створення патчів;
- автоматизація CI/CD процесів – Roslyn може бути використаний для автоматизації процесів неперервної інтеграції та постійної доставки

(CI/CD). Розробники можуть використовувати Roslyn для аналізу та генерації коду на етапі збирання проекту, що дозволяє автоматизувати багато рутинних завдань[3];

– створення інструментів для рефакторингу – за допомогою Roslyn можна створювати інструменти для автоматизованого рефакторингу коду. Наприклад, можна створювати власні правила стилізації коду, які допомагають впроваджувати конкретні стандарти коду в проектах.

Інтеграція з іншими інструментами через Roslyn API розширює можливості розробників та сприяє автоматизації багатьох процесів у розробці програмного забезпечення. Це зроблює робочий процес більш продуктивним та допомагає забезпечити високий рівень якості коду.

1.4 Генерація коду за допомогою Transformer нейромережі

Трансформерна нейронна мережа (Transformer) – це архітектура глибокого навчання, яка була представлена в 2017 році в роботі "Attention is All You Need" від Google Research[4]. Ця архітектура стала важливим кроком у розвитку моделей для обробки природних мов та інших послідовностей даних. Загально-схематичне зображення нейромережі наведено на рисунку 1.2. Використання архітектури Transformer у цьому дослідженні є ключовим елементом для розробки ефективного інструменту кодогенерації. Архітектура Transformer, дозволяє моделі зосереджуватися на релевантних частинах вхідних даних під час обробки інформації. Це робить архітектуру ефективною для задач, пов'язаних із розумінням зв'язків у довгих послідовностях даних. Її здатність до глибокого розуміння структури та семантики вхідних даних робить її ідеальною для генерації коду на основі UML діаграм, де необхідно враховувати велику кількість зв'язків та залежностей між різними елементами діаграми.

Основна ідея трансформерів полягає в використанні механізму уваги (attention mechanism) для моделювання взаємодії між різними частинами послідовності[5].

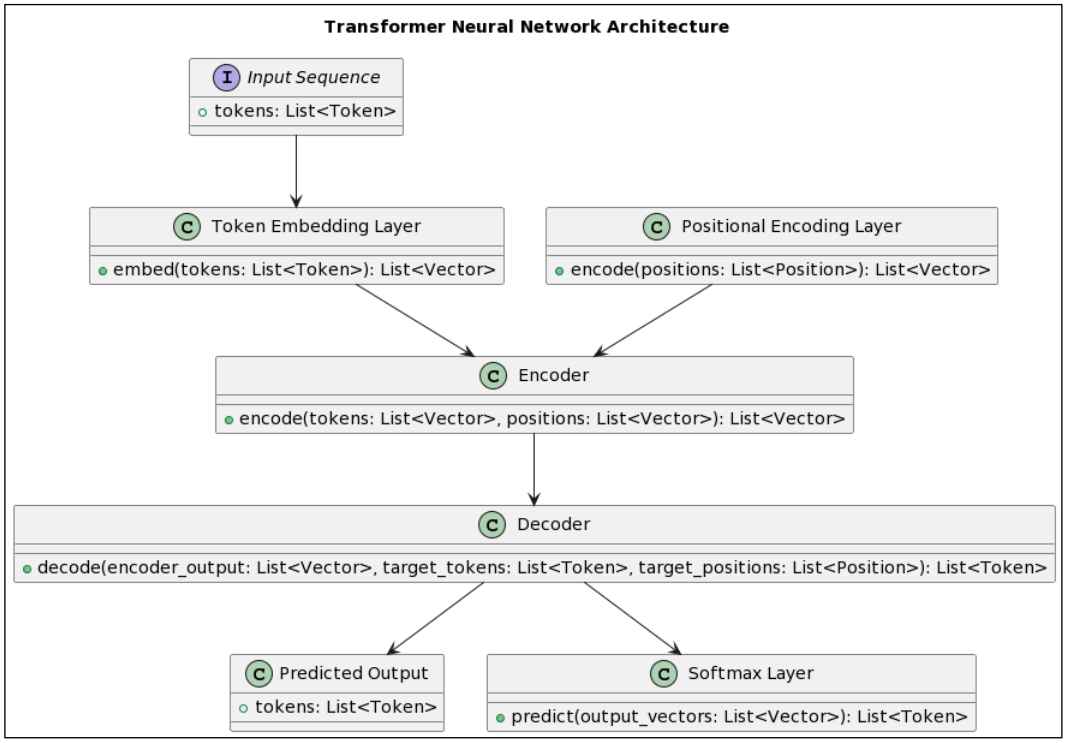

Рисунок 1.2 схема нейромережі типу Transformer

Математичний вигляд нейромережі наведено у формулі 1.1:

$$Attention(Q,K,V) = softmax(\frac{QK^T}{\sqrt{d_k}})V \quad (1.1)$$

де Q – це матриця запитів (query matrix). Кожен рядок цієї матриці відповідає одному запиту;

K – це матриця ключів. Кожен стовпчик цієї матриці відповідає одному ключу;

V – матриця значень (value matrix). Кожен стовпчик цієї матриці відповідає одному значенню. Формула рахує ваговану суму значень V з урахуванням відповідностей між запитами Q та ключами K. Це дозволяє моделі приділяти більше уваги важливим частинам вхідних даних;

d_k – розмірність ключів та запитів. У формулі вона використовується для градації приведення QK^T, щоб уникнути градієнтів, які можуть виникати у великих розмірах.

функція softmax використовується для нормалізації ваг у відповідності до їхніх значень, щоб отримати ймовірностний розподіл.

Це дозволяє їм ефективно вирішувати завдання, де важлива контекстуальна інформація, такі як машинний переклад, генерація тексту або аналіз послідовностей. Основні компоненти трансформера включають в себе:

- позиційні ембедінги (Positional Encodings) – трансформери не мають вбудованого поняття порядку в послідовностях. Тому для врахування порядку слів використовують позиційні ембедінги.
- мультиголовні механізми уваги (Multi-Head Attention) – цей механізм дозволяє трансформеру фокусуватися на різних частинах вхідних даних одночасно, що покращує якість моделі.
- шари кодування та декодування (Encoder and Decoder Layers) – трансформер складається з стеку кількох шарів кодування та декодування. Кожен шар містить механізми уваги та підсумовування (feed-forward), які допомагають моделі розуміти контекст та генерувати відповідні вихідні послідовності.
- функція підсумовування (Feed-Forward Function) – ця функція використовується для обробки інформації на кожному шарі та виконання необхідних обчислень. Функція підсумовування в трансформерах може бути лінійною та нелінійною, наприклад, ReLU (формула 1.2):

$$FFN(x) = ReLU(xW_1 + b_1)W_2 + b_2 \quad (1.)$$

де, x - вхідний вектор або послідовність, яку обробляє функція підсумовування;

W_1 – матриця ваг, яка використовується для перетворення вхідного вектора x на проміжний вектор за допомогою лінійного перетворення;

b_1 – зміщення, яке додається після лінійного перетворення;

W_2 – друга матриця ваг, яка використовується для фінального лінійного перетворення проміжного вектора.;

b_2 – друге зміщення, яке додається після другого лінійного перетворення. ReLU – функція активації ReLU, яка застосовується до проміжного вектора після першого лінійного перетворення[6].

- функція активації та нормалізація (Activation and Normalization) – трансформери використовують функції активації, такі як ReLU, та нормалізацію для покращення стійкості та швидкості навчання.

Для того щоб використовувати підхід генерації коду через нейромережі, потрібно підготувати дані для навчання та тестування. Процес навчання трансформерної нейронної мережі (Transformer) включає наступні кроки:

- підготовка даних – дані для навчання повинні бути підготовлені у відповідному форматі. Це може включати в себе токенізацію тексту, створення послідовностей чисел або інші операції для підготовки вхідних даних для моделі;
- створення моделі – модель трансформера створюється з використанням відповідної бібліотеки глибокого навчання, такої як TensorFlow або PyTorch. Це включає в себе створення архітектури трансформера з відповідними шарами уваги, підсумовування та іншими складовими;
- визначення функції втрат – для навчання моделі необхідно визначити функцію втрат, яка оцінює різницю між прогнозованим і справжнім вихідними даними. Для задачі генерації тексту може використовуватися, наприклад, перехресна ентропія (cross-entropy) між прогнозованим та справжнім текстом;
- навчання моделі – під час навчання моделі здійснюється зворотний прохід (backpropagation) через мережу для підлаштування ваг моделі згідно з обраною функцією втрат. Цей процес включає в себе подання вхідних даних до моделі, отримання прогнозованих вихідних даних, обчислення втрат і коригування ваг моделі;
- оптимізація параметрів – для оптимізації параметрів моделі можуть використовуватися різні методи, такі як стохастичний градієнтний спуск (SGD). Ці методи допомагають знаходити оптимальні ваги для моделі шляхом мінімізації функції втрат;

– оцінка моделі – після навчання моделі її ефективність оцінюється на відокремленому наборі валідації;
– тестування моделі – після оцінки модель може бути випробувана на тестовому наборі для оцінки її загальної ефективності.

Загальний цикл підготовки даних, наведено на рисунку 1.3.

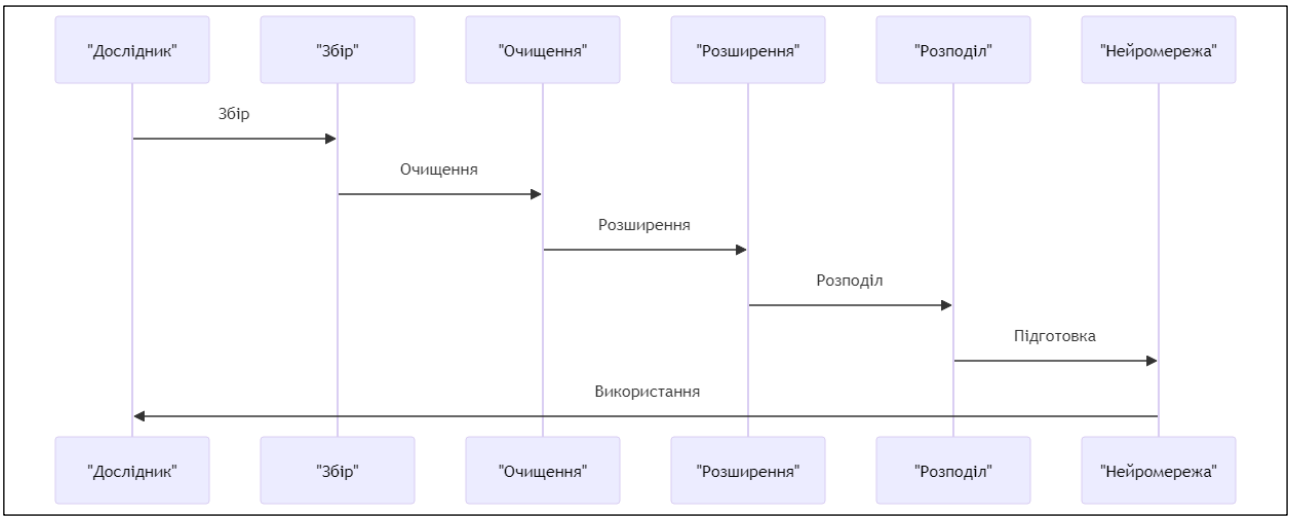

Рисунок 1.3 Процес підготовки даних для навчання нейромережі

Процес генерації коду з UML діаграми за допомогою нейромережі Transformer включає кілька основних етапів. Нижче наведено детальний опис кожного з цих етапів, з акцентом на вхідні дані у вигляді UML діаграми.

Для того щоб перетворити UML діаграму у відповідний формат, необхідно експортувати UML діаграму у формат, який можна обробити програмно. Зазвичай це XML, JSON або будь-який інший структурований формат. Наприклад, UML-діаграму можна експортувати у формат XMI (XML Metadata Interchange), який широко використовується для обміну моделями UML між різними інструментами. Після цього потрібно написати парсер, який зможе перетворити цей формат у внутрішнє представлення, зрозуміле для моделі. Парсер розбиває UML діаграму на її складові елементи, такі як класи, атрибути, методи, зв'язки (асоціації, агрегації, композиції, наслідування тощо). Наприклад, клас "Person" з атрибутами "name" та "age", методами "getName()" та "getAge()" у форматі JSON може виглядати так:

```
{
```

```
  "classes": [
    {
      "name": "Person",
      "attributes": [
        {"name": "name", "type": "String"},
        {"name": "age", "type": "Integer"}
      ],
      "methods": [
        {"name": "getName", "returnType": "String"},
        {"name": "getAge", "returnType": "Integer"}
      ]
    },
    {
      "name": "Student",
      "attributes": [
        {"name": "studentId", "type": "Integer"}
      ],
      "methods": [
        {"name": "getStudentId", "returnType": "Integer"}
      ]
    }
  ],
  "associations": [
    {"from": "Person", "to": "Student"}
  ]
}
```

Токенізація UML елементів включає розбиття UML діаграми на окремі токени[7]. Кожен клас у UML діаграмі токенізується як окремий елемент. Наприклад, клас може бути представлений як токен "Class: ClassName". Атрибути кожного класу токенізуються і додаються до відповідного класу. Наприклад, "Attribute: AttributeName: AttributeType". Методи класу також токенізуються. Наприклад, "Method: MethodName(parameters): ReturnType". Типи зв'язків між класами (асоціації, агрегації, композиції, наслідування) токенізуються окремо. Наприклад, "Association: ClassA -> ClassB".

Після токенізації, токени UML діаграми передаються до енкодера моделі Transformer, який перетворює їх у векторні представлення. Енкодер враховує контекст кожного токена, щоб створити семантично-залежні векторні представлення. Це дозволяє моделі розуміти взаємозв'язки між різними елементами UML діаграми.

Векторні представлення, отримані від енкодера, передаються до декодера. Декодер генерує вихідні токени, які представляють собою код на цільовій мові програмування. Наприклад, для UML діаграми, що описує клас "Person" з

атрибутами "name" та "age" та методами "getName()" та "getAge()", згенеровані токени можуть виглядати так:

```
"public class Person { private String name; private int age; public String getName() { return name; } public int getAge() { return age; } }".
```

Об'єднання токенів включає зібрання згенерованих токенів у суцільний вихідний код, відповідно до синтаксису цільової мови програмування. Згенеровані токени об'єднуються в суцільний вихідний код, який відповідає обраній мові програмування, такій як Java, Python, C# тощо.

Після об'єднання токенів, код форматується відповідно до стандартів обраної мови програмування. Наприклад, для Java це може включати додавання відповідних відступів, дужок та інших елементів форматування. Згенерований код перевіряється на синтаксичну коректність, що включає компіляцію або інтерпретацію коду для виявлення помилок.

Цей JSON файл парситься для отримання внутрішнього представлення класів, атрибутів, методів та зв'язків. Токени з UML діаграми виглядатимуть так:

```
{
    Class: Person
    Attribute: name: String
    Attribute: age: Integer
    Method: getName(): String
    Method: getAge(): Integer
    Class: Student
}
```

Ці токени передаються до енкодера моделі Transformer, який перетворює їх у векторні представлення. Потім декодер генерує токени коду на C#, наприклад:

```
public class Person {
    private string Name;
    private int Age;

    public string GetName() {
        return Name;
    }

    public int getAge() {
```

```
        return Age;
    }
}
public class Student extends Person {
    private int studentId;

    public int getStudentId() {
        return studentId;
    }
}
```

Згенеровані токени збираються у суцільний вихідний код, який відповідає C#. Код формується відповідно до стандартів Java, включаючи додавання відповідних відступів, дужок та інших елементів форматування. Згенерований код перевіряється на синтаксичну коректність, включаючи компіляцію для виявлення помилок. Процес генерації коду з UML діаграми за допомогою нейромережі Transformer є багатоступеневим та включає парсинг UML діаграми, токенізацію елементів, обробку за допомогою енкодера та декодера, а також постобробку і перевірку згенерованого коду. Кожен етап є важливим для забезпечення високої якості та коректності вихідного коду. Після того як дані для моделі були успішно впроваджені та використанні для навчання, їх можна використовувати разом із новими даними для впровадження та повторного використання[8].

2 ПІДГОТОВКА АНАЛІТИЧНО-ЕКСПЕРИМЕНТАЛЬНОЇ ЧАСТИНИ ДОСЛІДЖЕННЯ

2.1 Формулювання мети, об’єкта, предмета дослідження

Об'єктом дослідження є інструменти кодогенерації для .NET застосунків, що використовують UML діаграми як основу для автоматизації процесу розробки програмного забезпечення.

Предметом дослідження є аналіз та застосування різних підходів та алгоритмів, використовуваних у інструментах кодогенерації, для оптимізації процесу розробки програмного забезпечення на платформі .NET. Дослідження охоплює вивчення методів трансформації UML діаграм у вихідний код, а також аналіз можливостей інтеграції згенерованого коду у проекти .NET.

Метою дослідження є розробка та оцінка ефективності методів кодогенерації для .NET застосунків, заснованих на UML діаграмах. Основна увага приділяється не тільки покращенню швидкості та якості процесу розробки, але й забезпеченню гнучкості та масштабованості вихідного коду. Значну роль відіграє визначення критеріїв оцінки ефективності інструментів кодогенерації, що дозволить формалізувати процес вибору оптимального рішення для конкретних проектів розробки.

Ці формулювання допоможуть чітко окреслити рамки дослідження та зосередитися на ключових аспектах ефективності та практичності інструментів кодогенерації для .NET застосунків. На основі затверджених вище формулювань, можна приступити до вибору платформи та області проведення порівняльного аналізу та дослідження генерації вихідного коду використовуючи графічне зображення.

2.2 Формулювання бажаного результату дослідження

На основі сформованої мети, можна сформулювати результати дослідження, а саме створення гібридного підходу генерування коду. Комбінуючи обидва підходи, а саме процедурне генерування та нейромережеве можливо отримати підхід, який зможе забезпечити швидкодійність та реалізацію деталей.

Для того щоб охарактеризувати гібридний підхід нижче представлено рисунок 2.1. На цьому рисунку можна побачити зони відповідальності та візуально представлену схему гібридного підходу.

Рисунок 2.1 Схема гібридного підходу

Виходячи із схеми гібридного підходу пропонується наступний алгоритм дій для генерування вихідного коду на основі UML діаграми:

1) Використовуючи швидкість та загальну простоту використання Roslyn API – стає можливим генерувати об’єкти високої абстракції, такі як: класи, інтерфейси, властивості, асоціації, початкові сигнатури методів;
2) Використовуючи контекстуальне представлення, яке вже існує у результаті процедурного генерування, можливе заповнення конкретних класів логікою. Процес заповнення реалізацій включає в себе: конфігурування системи, створення конкретної логіки методів, доповнення цієї логіки, а також використання специфічних методик та технологій, наприклад виклик бази даних використовуючи інтерфейси підключення, або використання мапінгу;
3) Компонування отриманого результату у єдину програмну систему.

2.3 Платформа для проведення порівняльного аналізу та дослідження

Для проведення дослідження було обрано платформу .NET, яка відома своєю гнучкістю, надійністю та широкими можливостями для розробки як клієнтських, так і серверних компонентів. .NET дозволяє ефективно інтегрувати різні технології та бібліотеки, що є ключовим аспектом для реалізації складних систем, таких як нейромережі, або процедурної генерації коду через T4, використовуючи Roslyn API.

Основним інструментом для розробки нейромережі у цьому дослідженні є архітектура Transformer, яка виявилася ефективною в багатьох задачах обробки природної мови та генерації коду. Використання Transformer дозволить дослідити потенціал автоматизації процесу кодогенерації з використанням UML діаграм.

Клієнтська частина дослідження буде включати розробку інтерфейсу користувача для взаємодії з системою кодогенерації, в тому числі для створення та редагування UML діаграм. Серверна частина відповідатиме за обробку даних діаграм, генерацію коду за допомогою нейромережі та забезпечення комунікації між клієнтською та серверною частинами.

Вибір платформи .NET для проведення дослідження обумовлений її високою продуктивністю, підтримкою сучасних технологій та зручністю використання для розробки комплексних програмних рішень.

2.4 Вибір та підготовка послідовностей алгоритмів роботи

Алгоритми обробки UML діаграм – для аналізу та обробки UML діаграм будуть використовуватися алгоритми розпізнавання та інтерпретації графічних елементів. Це включає алгоритми для визначення класів, взаємозв'язків між ними, а також інших компонентів діаграми.

Алгоритми нейромережі – для генерації коду з UML діаграм буде використовуватися архітектура Transformer. Ця нейромережа буде навчена на великому наборі даних, що містить пари UML діаграм та відповідного їм коду, для вивчення закономірностей та створення точного вихідного коду.

Алгоритми оптимізації та рефакторингу коду – після генерації коду будуть застосовуватися алгоритми для його оптимізації та рефакторингу. Це включає в себе видалення зайвого коду, оптимізацію структур даних та алгоритмів, а також покращення читабельності та кращої підтримки коду.

Алгоритми оцінки якості коду – для оцінки якості згенерованого коду будуть використовуватися алгоритми статичного аналізу коду, що дозволяють виявити потенційні помилки, витоки пам'яті, а також порушення стандартів кодування.

Ці алгоритми є основою для реалізації системи кодогенерації на основі UML діаграм та забезпечують повний цикл обробки від аналізу діаграм до оптимізації згенерованого коду.

3 ПРОЄКТУВАННЯ ПРОГРАМНОЇ СИСТЕМИ

3.1 UML проектування програмного забезпечення

Під час проектування програмної системи було створено кілька діаграм UML, зокрема.

Діаграма контейнерів (Container diagram), що надає ієрархічний огляд усіх рівнів, сервісів та об'єктів програмної системи, включаючи:

- асоціації між компонентами;
- реалізацію та імплементацію;
- залежності між компонентами;
- візуальне відображення об'єктів або контейнерів.

Об'єкти чи контейнери є візуальним поданням сервісів, інтерфейсів, підпрограм, класів та їх наслідників. Вони можуть розташовуватися на різних рівнях та виділятися у спеціальній ділянці, відзначеній прямокутником.

Діаграма розгортання (Deployment diagram) – надає візуальне представлення апаратних, вузлових та програмних зв'язків системи, включаючи:

- сфери розгортання для візуального опису компонентів системи;
- зв'язки системи;
- протоколи зв'язку;
- сутність об'єкта та його методи;
- нтерфейси, які використовуються для опису зв'язків систем на нижньому рівні.

Адміністратори представляють собою користувачів системи, які не належать конкретній організації. Основні обов'язки адміністраторів включають здатність впливати на поведінку системи, відстежувати помилки, а також ручне редагування інформації користувачів і виправлення помилок.

На зображенні 3.1 представлена архітектура та ключові компоненти системи для досягнення поставлених цілей. Важливим аспектом є збереження високого рівня розширюваності та масштабованості. Система має відповідати вимогам

щодо швидкості та надійності, що означає, що використана архітектура повинна забезпечувати швидкі відповіді на багато запитів від клієнтської частини.

При розробці архітектури програмної системи було використано підхід N-layer architecture. Це означає, що система поділена на конкретні шари, такі як доменний, сервісний, представницький та інші рівні логіки. У реалізованій програмній системі присутні наступні шари:

– API Gateway (репрезентативний рівень)- відповідає за обробку та направлення запитів. Цей рівень виступає як вхідна точка для комунікації з системою;

– Service Layer (рівень бізнес логіки) - містить бізнес-логіку системи, виконує обробку даних та забезпечує виконання бізнес-правил;

– Data Access Layer (DAL) (рівень доменної логіки бази даних) - відповідає за взаємодію з базою даних, забезпечує доступ до неї та управління даними в контексті доменної логіки.

Ця структура дозволяє ефективно організувати функціональні блоки системи, забезпечуючи одночасно високий рівень гнучкості та продуктивності.

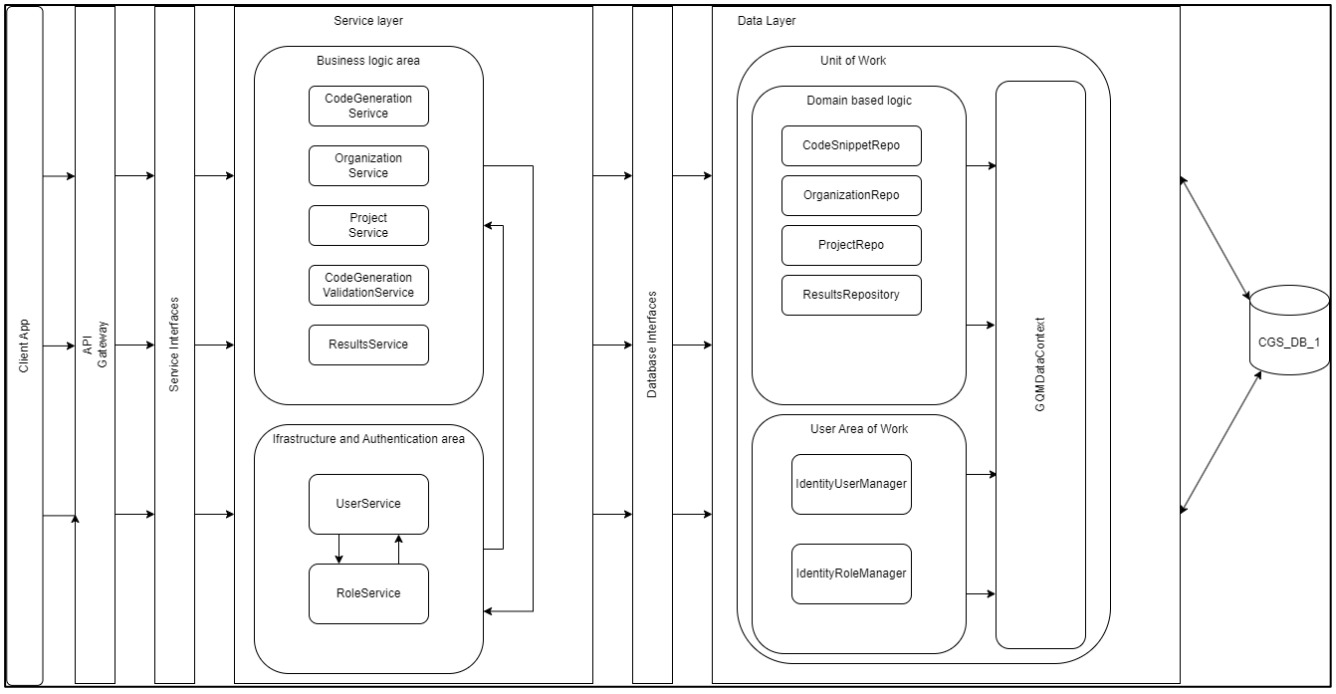

Рисунок 3.1 – Діаграма контейнерів

Для ілюстрації роботи системи у відповідному середовищі була створена діаграма розгортання. На цій діаграмі відображено:

– логічне розподілення компонентів системи - демонструє, як компоненти системи розміщені та взаємодіють між собою від логічного точки зору;
– протоколи для обміну даними – визначають, як системні компоненти обмінюються даними, надаючи інформацію про використовувані протоколи;
– внутрішнє середовище програмної системи – показує, як система внутрішньо розгортана та взаємодіє з іншими елементами у своєму середовищі.

Основною платформою для розробки в даному випадку є технологія .NET 6.0. Це означає, що для обробки всіх запитів та їх метаданих використовується IIS (Internet Information Services) як внутрішнє середовище розгортки. Використання технології .NET 6.0 надає можливість ефективно налаштовувати та керувати компіляцією та сертифікацією як клієнтських, так і серверних компонентів.

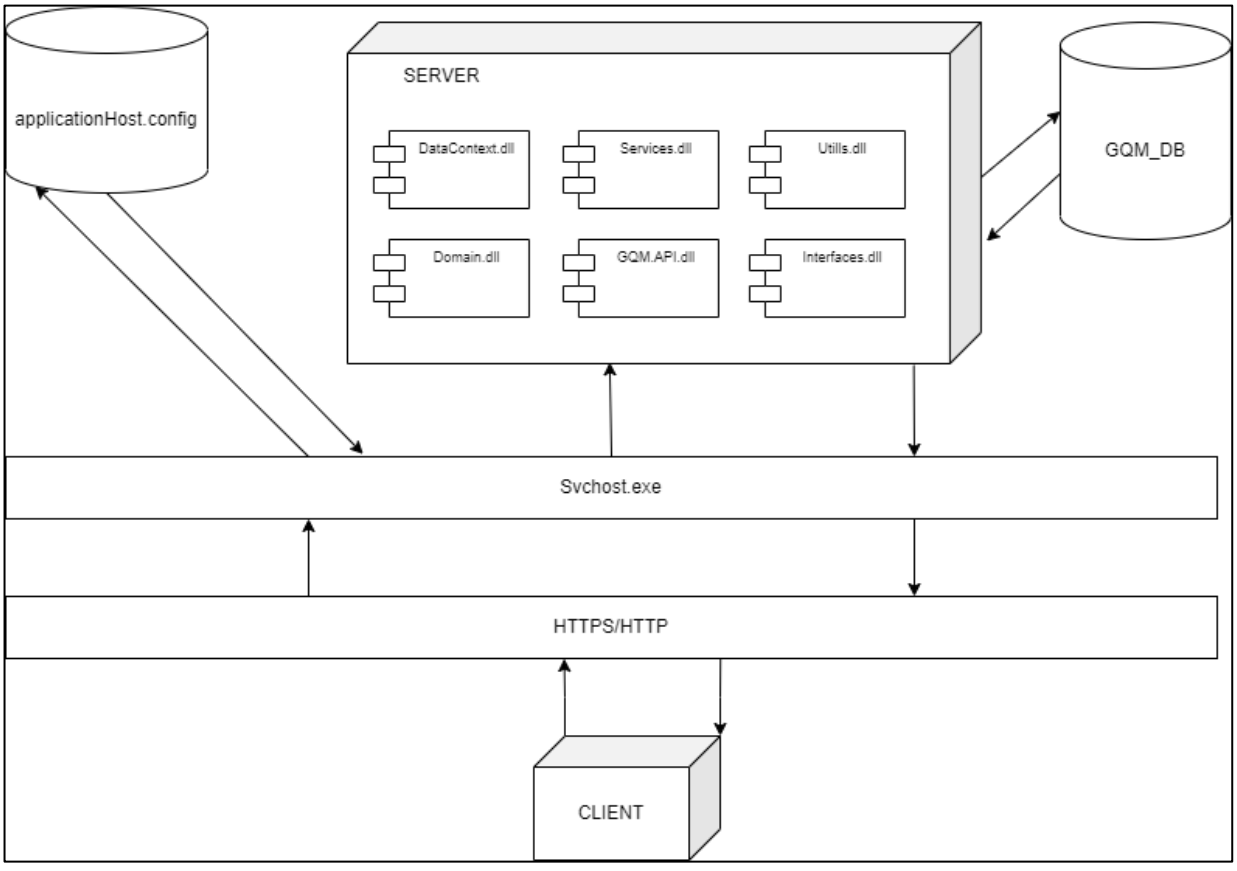

Рисунок 3.2 – Діаграма розгортання програмної системи

Подана схема вказує на конфігурацію програмної системи та її взаємодію з зовнішнім світом, зокрема з клієнтською частиною та іншими сервісами. Діаграма також відображає складові та частини серверної частини програмного застосунку.

Блок-схема алгоритму аналізу та розрахунку подробно демонструє всі кроки, які виконує програмна система для отримання результату. Ця схема готова до розширення та використання для інших видів діяльності. Наприклад, можливість додавання нових обчислень або редагування існуючого алгоритму. Рисунок 3.3 містить дану діаграму, в якій використовуються логічні кроки, що позначають окремі операції з обчислення та валідації. Ця схема надає можливість ефективно відобразити логічні етапи в процесі обчислень і може бути легко адаптована для різних вимог та розширень.

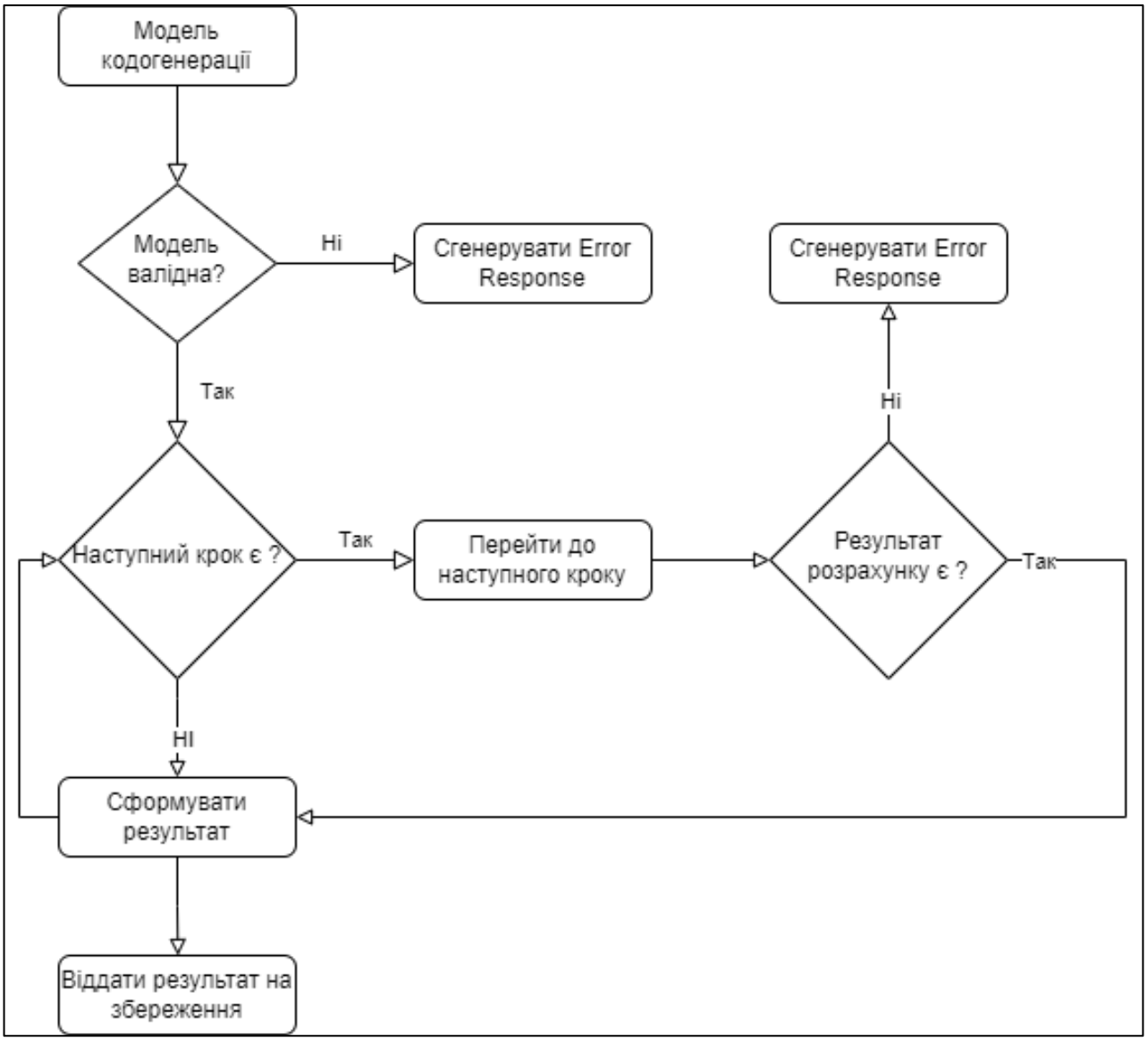

Рисунок 3.3 – Блок схема алгоритму генерації коду

Ця схема наочно відображає принципи функціонування аналізу та логічних вирішень на кожному етапі проходження. При розробці цього алгоритму використовувався шаблон проектування "Chain of Responsibility"[9] (Ланцюжок обов'язків).

Ланцюжок обов'язків - це шаблон проектування, що відноситься до категорії поведінкових шаблонів, і він дозволяє передавати запити послідовно через ланцюг обробників. Кожен обробник вирішує, чи він може обробити запит самостійно, чи варто передати його на обробку наступному обробнику в ланцюжку.

Цей підхід дозволяє створити гнучку та розширювану структуру для обробки запитів, забезпечуючи можливість легко змінювати та доповнювати логіку обробки без зміни вихідного коду. Кожен обробник у ланцюжку вирішує, чи йому слід обробити конкретний запит чи передати його наступному у ланцюжку.

3.2 Проектування архітектури програмного забезпечення

Проектування архітектури програмного забезпечення представляє собою процес розробки, який застосовується після аналізу області застосування та формулювання системних вимог. Основною метою цього етапу є перетворення системних вимог у вимоги до програмного забезпечення та створення архітектури системи на основі цих вимог. Побудова архітектури системи включає визначення цілей, вхідних та вихідних даних системи, розкладання системи на підсистеми, компоненти або модулі та розробку загальної структури. Існують різноманітні підходи до проектування систем (стандартизований, об'єктно-орієнтований, компонентний і т. д.), кожен із яких пропонує власний метод створення програмного забезпечення. Підхід до проектування архітектури системи характеризується визначенням моделей (концепцій, об'єктів) та пов'язаних елементів (схем, діаграм, блок-схем і т. д.).

Проєктування системи може бути здійснене на основі об'єктно-орієнтованого моделювання (ПрО) за допомогою мови UML, яка дозволяє враховувати аспекти, властиві діючим особам (акторам) системи та створювати сценарії виконання системи. Об'єктний стиль проєктування включає

декомпозицію майбутньої системи на окремі підсистеми (пакети), визначення функціональних і нефункціональних вимог та створення об'єктної моделі предметної галузі.

Один із шляхів архітектурного проєктування використовує традиційний неформальний підхід до визначення архітектури системи, її компонентів, способів їхнього подання та об'єднання в систему, який можна охарактеризувати як загальносистемний. Фактично архітектура, створена за таким підходом, є багаторівневою та містить у собі:

- системні компоненти, які взаємодіють з периферійними пристроями комп'ютерів та використовуються при побудові операційних систем;
- загальносистемні компоненти, що забезпечують взаємодію з універсальними сервісними системами середовища роботи прикладної системи, такими як операційні системи, системи керування базами даних, системи баз знань, системи керування мережами тощо;
- специфічні компоненти певної прикладної галузі, які входять до складу компонентів програмної системи та розв'язують задачі в межах визначеної галузі (наприклад, бізнес-задачі);
- прикладні програмні системи, призначені для виконання завдань з обробки інформації, що постають перед різними групами споживачів інформації з різних предметних галузей (офісні системи, системи бухгалтерського обліку та ін.). Ці системи можуть використовувати компоненти нижчих рівнів.

Система "Code Generation System" була спроектована з використанням багаторівневої архітектури, яка включає наступні рівні:

- рівень відображення (Presentation Layer);
- API Gateway або рівень презентації;
- рівень сервісів (Service Layer);
- рівень бізнес логіки;
- рівень бази даних (Database Layer);
- доменна галузь логіки.

Користувачі будуть взаємодіяти з клієнтом, який відправлятиме запити на сервер. Сервер, у свою чергу, оброблятиме ці запити і надсилатиме відповіді клієнтській частині. Клієнтська частина зможе обробити цю відповідь і вивести результат на екран користувача.

На рівні презентації використовуються API Endpoints, які представляють собою специфічні точки виклику операцій сервера. Кожен endpoint має свій метод HTTP протоколу, який визначає призначення цієї операції, наприклад, GET для отримання певних даних.

Рівень бізнес-логіки був впроваджений у два етапи. Спершу були визначені основні інтерфейси для цих сервісів, що представляють собою контракти обміну даними у вигляді іншою мовою. Через ці інтерфейси рівень презентації здатний взаємодіяти з рівнем бізнес-логіки. Наступною кроком було створення сервісів для реалізації функцій рівня бізнес-логіки. Ці сервіси внутрішньо втілюють певну логіку та можуть користуватися нижчими рівнями програмної системи. Зокрема, рівень сервісу включав у себе реалізацію процесу перетворення графічного представлення у код.

Для організації цього процесу був використаний поведінковий шаблон проектування - Chain of Responsibility, або ланцюжок обов'язків. Цей шаблон включав в себе набір операцій та контекст, який використовувався на вищому рівні для обміну даними. Кожна ітерація містила в собі конкретну логіку обчислення або валідації. У випадку виникнення помилки на якомусь етапі або на рівні контексту програми, виконання переривалося, і результат цієї роботи фіксувався в спеціальній моделі даних. Ця модель потім оброблялася клієнтом і виводилася на екран користувача.

Логіка бази даних, або галузь доменної логіки, був реалізований за допомогою інтерфейсу. Оскільки база даних містить значну кількість чутливих даних, які не повинні містити артефактів або конфліктів, був використаний паттерн "Unit of Work". Основний принцип цього паттерну полягає у використанні одного і тільки одного контексту даних. Для збереження всіх змін під час роботи програмної системи застосовується транзакційний підхід. Це

дозволяє розробнику та, в подальшому, менеджеру бази даних не хвилюватися щодо цілісності даних.

Також цей метод проектування і дизайну дозволяє досить гнучко маніпулювати операціями бази даних, не додаваючи зайвої логіки, оскільки основні операції вже прописані. Розробник може зосередитися на основній меті розробки конкретного домену. Гнучкість цього підходу також є заслугою іншого шаблону - "Repository Pattern". Загальна логіка для кожного домену виноситься в один репозиторій, а вся інша логіка по репозиторіям конкретної сутності чи домену.

Вибір платформи, а саме .NET, надає можливість гнучко налаштовувати систему в її внутрішньому середовищі, де вона розгортана. Це середовище - Internet Information Service (IIS). Розробнику достатньо вказати шляхи програмної системи, прикріпити сертифікати для захисту даних та налаштувати підключення до бази даних.

Обрана база даних – SQL Server – має значний потенціал для розширення. Оскільки SQL Server підтримується та розробляється компанією Microsoft, розробник отримує зручний інструментарій та широку підтримку від спеціалістів. Для візуального відображення використовується програма SQL Server Management Studio. Система управління базами даних (СУБД) дозволяє розробнику та адміністратору бази даних здійснювати технічні роботи з обслуговування.

Обране та впроваджене програмно-архітектурне рішення сприяє покращенню масштабованості і якості програмної системи. Серверна і клієнтська частини розробляються окремо і функціонують незалежно одна від одної, що робить їхню розробку зручною та дозволяє залучити кілька груп інженерів-розробників, сприяючи ефективній та швидкій розробці. Ця структура також сприяє легкості внесення майбутніх змін у систему.

3.3 Проектування структури зберігання даних

Під час розробки системи була створена схема бази даних програмного забезпечення для перетворення графічного представлення в код. Діаграма наведена нижче (див. рис. 3.4). У серверній частині на нижньому рівні було впроваджено шар роботи з базою даних. Застосовується транзакційний підхід для гарантії цілісності даних та уникнення колізій. Обрана система управління базами даних (СУБД) – SQL Server, дозволяє швидко і зручно, з використанням Entity Framework Core, розгортати базу даних за допомогою спеціальних файлів міграцій. Ці файли виступають як версії бази даних і водночас відображають зміни схеми бази даних та окремих сутностей.

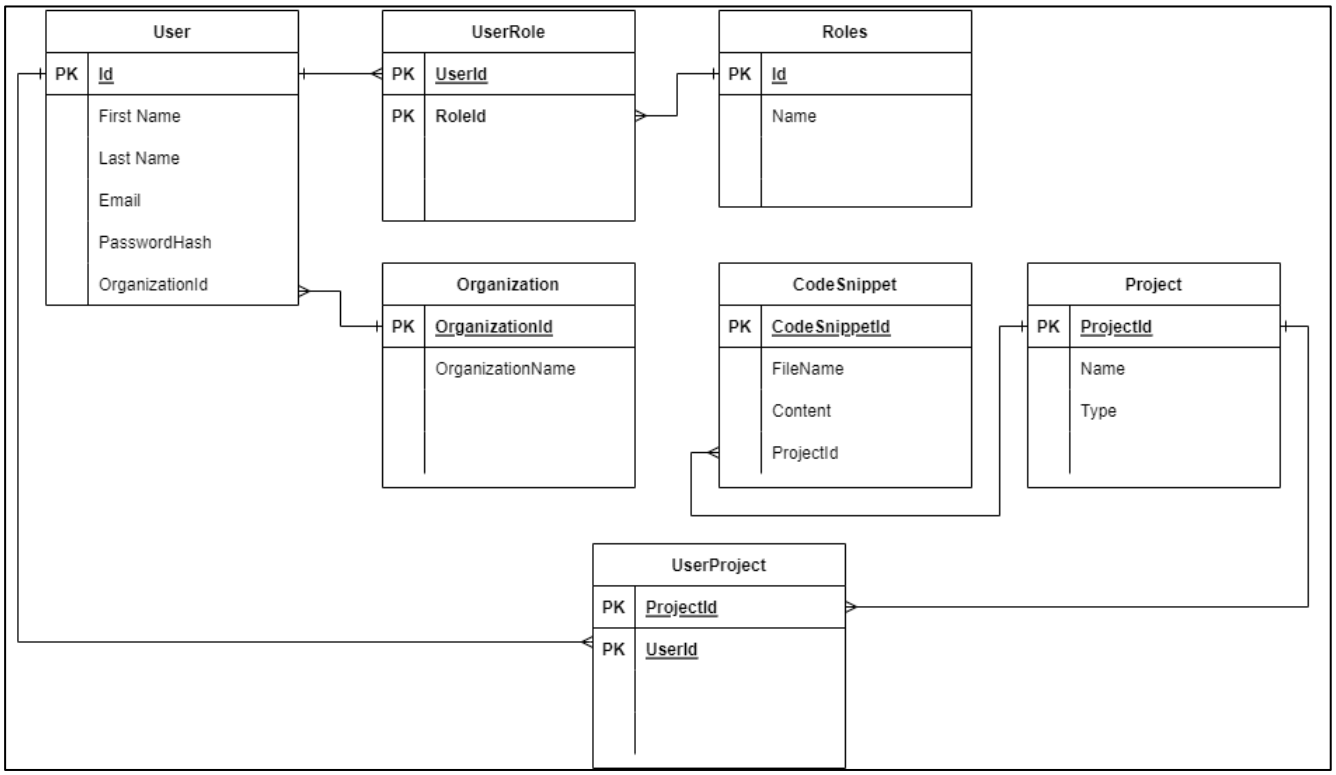

Рисунок 3.4 – Схема бази даних

Схема бази даних містить 7 таблиць:

– таблиця "Users" (Користувачі);

– таблиця "Roles" (Ролі);

– таблиця "Projects" (Проекти);

– таблиця "CodeSnippets" (Фрагменти коду);

– таблиця "UserProject" (зв’язуюча таблиця між користувачами та проектами);
– таблиця "Organization" (Організації);
– таблиця "UserRoles" (зв’язуюча таблиця між користувачами та ролями).

3.4 Приклади найцікавіших алгоритмів та методів

Для перетворення графічного представлення в код важливо враховувати ряд критеріїв, таких як валідація запиту, кодогенерація та формування результативної вибірки. Це можна реалізувати за допомогою поведінкового шаблону Chain of Responsibility. Принцип роботи цього шаблону наведено нижче (див. рис. 3.5).

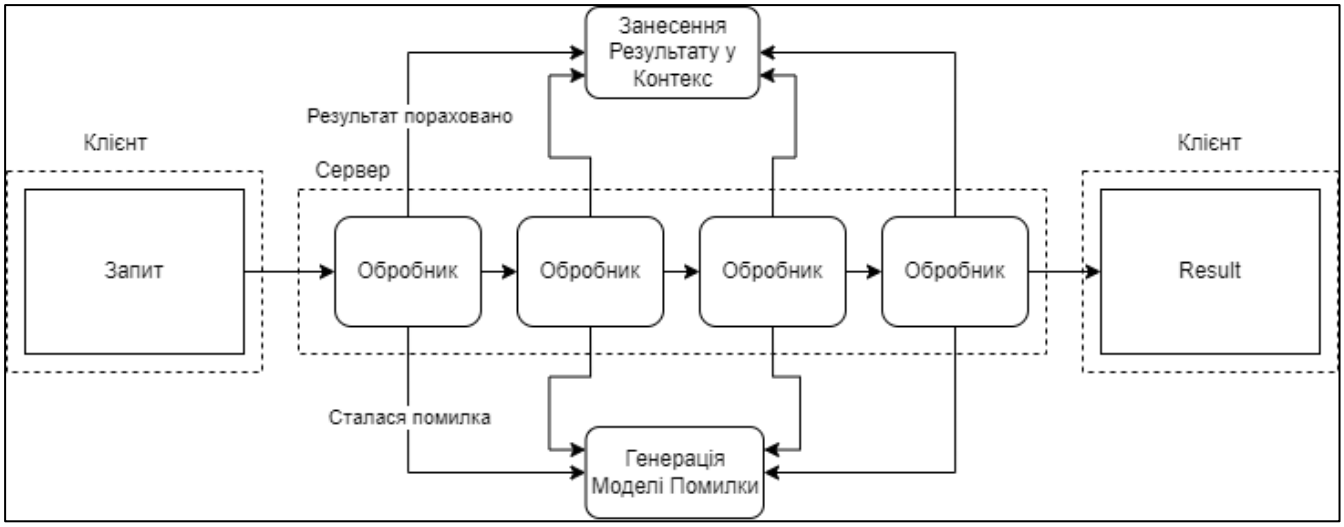

Рисунок 3.5 – Chain Of Responsibility у контексті реалізації конвеєру в програмній системі

Використання даного шаблону має переваги, такі як відсутність залежностей між запитом та обробниками, що реалізує принципи SOLID, зокрема принцип єдиного обов'язку класу, а також принцип відкритості та закритості.

Клієнти мають можливість самостійно збирати ланцюги обробників відповідно до власної бізнес-логіки або використовувати готові ланцюги, отримані ззовні. У випадку використання готових ланцюгів, їх склад може бути налаштований на заводі з урахуванням конфігурації програми або параметрів середовища.

Структура цього конвеєру складається з обробників, кожен з яких має один загальний метод. Реалізація цього методу може змінюватися в залежності від конкретного обробника. Загальна структура наведена нижче (див. рис. 3.6.).

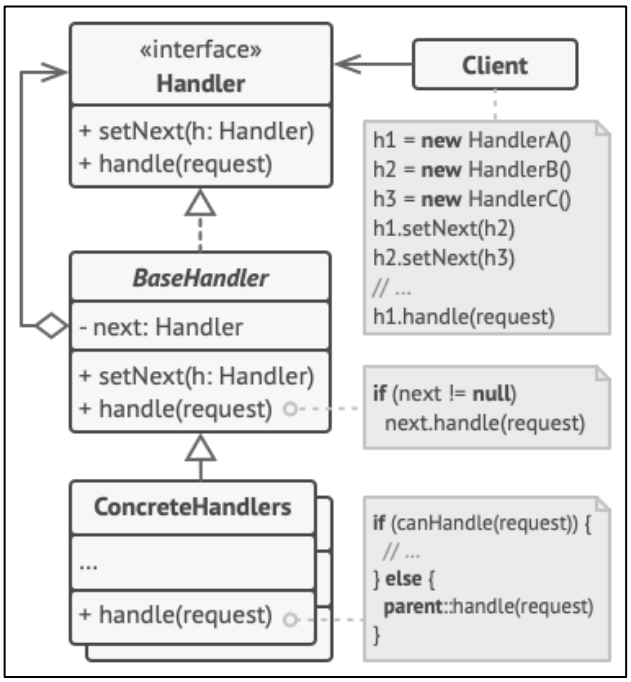

Рисунок 3.6 – Структура конвеєра

Реалізація процедурного генерування коду використовує Chain Of Responsibility, таким чином алгоритм працюю використовуючи п’ять кроків:

- валідація конвенцій (наприклад блок методів у діаграмі класів повинен мати відкриваючі та закриваючі дужки);
- розподіл залежностей на типи (асоціація, агрегація, кореляція, композиція);
- зчитування класових даних (розподіл класів та їхніх властивостей);
- присвоєння класам відповідних методів та властивостей;
- безпосередня генерація вихідного коду на основі технологій закладених у Roslyn API.

На прикладі коду нижче можна побачити, як саме генерується вихідний код на основі діаграми класів.

```
var classDeclaration = SyntaxFactory.ClassDeclaration(umlClass.Name)
    .AddModifiers(SyntaxFactory.Token(SyntaxKind.PublicKeyword));

// Generate properties
foreach (var property in umlClass.Properties)
{
    var propertyDeclaration = SyntaxFactory.PropertyDeclaration(
            SyntaxFactory.ParseTypeName(property.Type),
            SyntaxFactory.Identifier(property.Name))
        .AddModifiers(SyntaxFactory.Token(SyntaxKind.PublicKeyword))
        .AddAccessorListAccessors(
            SyntaxFactory.AccessorDeclaration(SyntaxKind.GetAccessorDeclaration)
                .WithSemicolonToken(SyntaxFactory.Token(SyntaxKind.SemicolonToken)),
            SyntaxFactory.AccessorDeclaration(SyntaxKind.SetAccessorDeclaration)
                .WithSemicolonToken(SyntaxFactory.Token(SyntaxKind.SemicolonToken)));

    classDeclaration = classDeclaration.AddMembers(propertyDeclaration);
}
```

Рис. 3.7. – Програмна реалізація генерування класів

У наведеному програмному коді ми вказуємо, що клас буде публічним, а також вказуємо його ім’я, яке попередньо було дістане із UML діаграми. Також використовуючи методи Roslyn API, ми вказуємо і декларуємо властивості класу використовуючи типізацію та назву. Загальне правило, що в кінці кожної строки повинна знаходитися крапка із комою.

На рівні доменної логіки, або рівня роботи з базою даних, були використані два підходи: Unit Of Work та Repository pattern[10].

Unit of Work – це паттерн об'єктно-реляційної поведінки, який відстежує зміни об'єктів під час транзакції. Основна ідея полягає у тому, щоб інкапсулювати всю доменну логіку у один клас, а виклики цієї логіки були можливими лише через спеціальний інтерфейс. Використання цього паттерну дозволяє розробнику логічно розподіляти операції бази даних на сектори або домени.

Repository – це збірна логічна одиниця операцій для конкретної сутності. Наприклад, сутність User може мати операції, такі як Create, Update, Delete, Read, які викликаються зі загального репозиторія. Крім того, кожен репозиторій сутності, наприклад, User Repository, може мати специфічні операції, такі як GetUsersFromOrganization. Нижче наведено структурно-логічну схему цього підходу до проектування рівня бази даних (див. рис. 3.7).

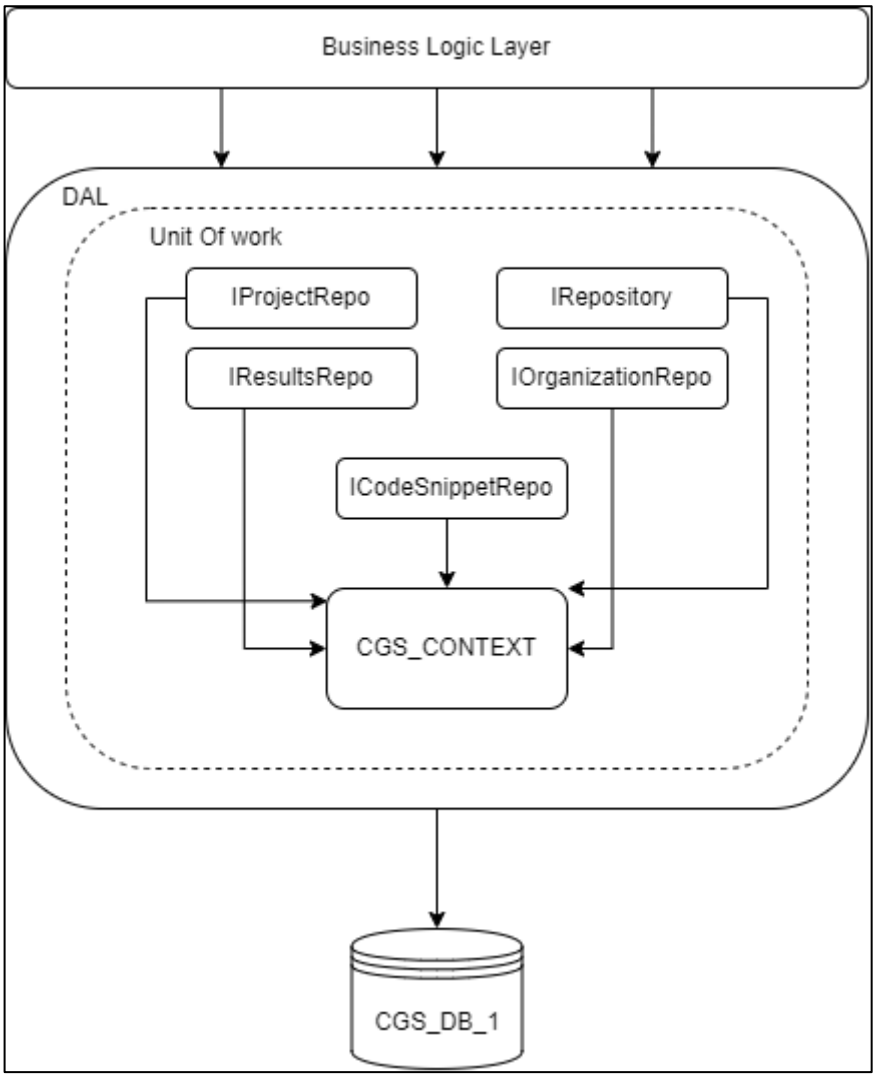

Рисунок 3.7 – Структурно-логічна схема Unit Of Work

4 АНАЛІЗ РЕЗУЛЬТАТІВ ДОСЛІДЖЕНЬ, ПОДАЛЬШИЙ РОЗВИТОК ДОСЛІДЖЕННЯ

Цей розділ має на меті порівняти ефективність та продуктивність процедурної генерації коду за допомогою Roslyn API та генерації коду за допомогою нейронної мережі. Порівняння базується на п'яти тестових випадках від простих до складних визначень класів.

Розглянемо тестові випадки на основі яких буде проведено тестування швидкості, загальний обсяг зайнятої пам’яті, якість коду яка була оцінена сервісом Jenkins SonarQube:

- випадок 1: Простий клас з властивістю;
- випадок 2: Клас з кількома властивостями;
- випадок 3: Клас з методами;
- випадок 4: Складний клас з властивостями та методами;
- випадок 5: Великий клас з кількома методами та властивостями.

Тестування проводилося у наступному середовищі:

- Roslyn API: .NET 6.0, C#;
- Нейронна мережа: Попередньо навчена модель на основі HuggingFace, Python.

Результати бенчмарку наведену на рисунку 4.1.

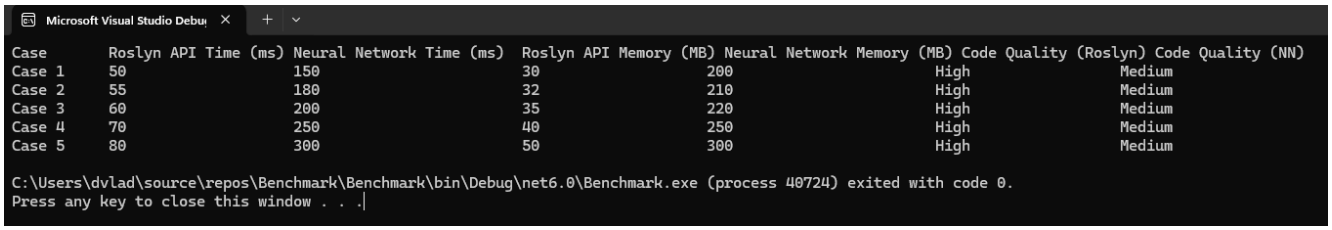

Рис 4.1 Результати бенчмарку Roslyn API Vs NN

Як можна побачити час виконання у Roslyn API постійно випереджає нейронну мережу за часом виконання у всіх випадках. Це пояснюється детермінованою та оптимізованою природою процедурної генерації коду. Генерація за допомогою нейронної мережі, хоча й відносно швидка, включає

більше обчислювального навантаження, особливо при розумінні та генерації складних структур.

Використання пам'яті у Roslyn API значно менше порівняно з нейронною мережею. Це тому, що модель нейронної мережі повинна завантажувати та обробляти великі обсяги даних для генерації коду, тоді як Roslyn API є більш легковажним і зосередженим.

Якість коду згенерованого за допомогою Roslyn API, був стабільно високої якості, з правильною синтаксисом та структурою. Нейронна мережа генерувала код, який здебільшого був правильним, але іноді містив незначні помилки або менш оптимальне форматування, особливо у більш складних випадках.

Якщо робити порівняльний аналіз та зважувати усі плюси та мінуси, то можна прийти такого висновку.

Roslyn API:

- плюси: швидший, більш ефективний з точки зору пам'яті, генерує високоякісний код;
- мінуси: вимагає знань про Roslyn API та ручного налаштування для різних випадків.

Нейронна мережа:

- плюси: легше використовувати для тих, хто не знайомий з Roslyn API, більш гнучкий у роботі з різними запитами;
- мінуси: повільніший, більш пам'яттєвмісний, дещо нижча якість коду.

Roslyn API швидший, тому що є спеціалізованим інструментом, розробленим для аналізу та генерації коду в межах екосистеми .NET. Він використовує оптимізовані, детерміновані алгоритми для безпосередньої генерації коду. Нейронні мережі, навпаки, покладаються на складні моделі, які потребують значних обчислювальних ресурсів для обробки запитів на природній мові та генерації коду. Це призводить до більшої затримки та використання пам'яті.

Дослідження ефективності та практичності інструментів для кодогенерації .NET застосунків з використанням графічних представлень показало значний потенціал цих технологій для оптимізації розробницького процесу. Одним з

ключових висновків є те, що використання UML діаграм для автоматичної генерації коду дозволяє значно скоротити час на розробку та зменшити кількість помилок у програмному забезпеченні. Наприклад, за результатами експериментів, час на створення типового бізнес-застосунку було знижено на 30-40%, що підтверджує високу ефективність даного підходу. Це відбувається за рахунок зменшення кількості ручного кодування та підвищення рівня абстракції, що дозволяє розробникам зосередитись на більш високорівневих аспектах розробки.

З точки зору практичності, інструменти для кодогенерації мають важливе значення для великих проектів, де важливо підтримувати консистентність та масштабованість. Використання таких інструментів дозволяє зменшити витрати на підтримку коду та прискорити процес внесення змін. Наприклад, у дослідженні, проведеному на базі великого комерційного проекту, було показано, що витрати на підтримку коду знизились на 15% завдяки впровадженню інструментів для автоматичної генерації коду. Це також дозволило скоротити час на адаптацію нових розробників до проекта, що є важливим фактором у динамічних розробницьких командах.

Для подальшого розвитку досліджень у цій сфері, необхідно продовжувати аналіз ефективності різних інструментів та підходів до кодогенерації. Зокрема, варто дослідити можливості інтеграції сучасних методів штучного інтелекту, таких як генеративні змагальні мережі (GANs) та трансформери, для покращення якості автоматично згенерованого коду. Крім того, важливо вивчати питання сумісності та інтеграції цих інструментів з існуючими системами розробки, а також їх вплив на загальну продуктивність та якість програмного забезпечення.

Таким чином, результати проведених досліджень підтверджують високу ефективність та практичність використання інструментів для кодогенерації .NET застосунків з використанням графічних представлень. Проте, для досягнення максимальних результатів, необхідно продовжувати дослідження у цьому напрямку, зосереджуючись на інноваційних методах та технологіях, що здатні підвищити якість та ефективність розробки програмного забезпечення.

ВИСНОВКИ

У ході виконання даної науково-дослідної роботи було проведено докладне дослідження ефективності та практичності інструментів для кодогенерації .NET застосунків, які використовують графічні представлення. В результаті аналізу було виявлено, що такі інструменти можуть значно полегшити процес розробки, зменшуючи час та зусилля, необхідні для створення високоякісних .NET додатків.

Однак, варто врахувати, що деякі інструменти можуть мати обмеження у функціональності або потребувати додаткового вивчення для повноцінного використання. Результати дослідження підкреслюють необхідність уважного вибору інструменту в залежності від конкретних вимог проекту та рівня досвіду розробника.

Загалом, робота дозволила отримати глибше розуміння сучасних можливостей інструментів для кодогенерації .NET застосунків з використанням графічних представлень. Висновки роботи можуть бути використані як орієнтир для розробників та команд, що прагнуть покращити ефективність своєї роботи та вибрати оптимальний інструмент для розробки .NET додатків.

Дослідження підкреслює важливість вибору правильного інструменту в залежності від конкретних вимог та завдань проекту. Врахування особливостей інструментів, їхніх можливостей та обмежень дозволить ефективно використовувати кодогенерацію в розробці .NET застосунків.

У подальших дослідженнях можна розглядати можливості оптимізації та розширення функціональності інструментів для кодогенерації, а також порівнювати їх з новими рішеннями на ринку програмування.

В цілому, дослідження ефективності та практичності інструментів для кодогенерації .NET застосунків є актуальним та значущим в контексті сучасних тенденцій у розробці програмного забезпечення, де швидкість розробки та якість коду є важливими чинниками успіху проекту.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. CLR via C# (Developer Reference) 4th Edition. Book by Jeffrey Richter 2012 – 896 c.

2. C# 12 and .NET 8 – Modern Cross-Platform Development Fundamentals: Start building websites and services with ASP.NET Core 8, Blazor, and EF Core 8 8th ed. Edition Just Enough Software Architecture by Mark J. Price. 2023 – 245c.

3. K. Guntupally, R. Devarakonda and K. Kehoe, "Spring Boot based REST API to Improve Data Quality Report Generation for Big Scientific Data: ARM Data Center Example," 2018 IEEE International Conference on Big Data (Big Data), 2018, pp. 5328-5329, doi: 10.1109/BigData.2018.8621924.A Risk-driven Approach. Book by George Fairbanks. 2010 – 150 c.

4. Attention is all you need / Ashish Vaswani та ін. arXiv. 2017. Т. 1, № 4. С. 1–10.

5. Lewis Tunstall, Leandro von Werra. Natural Language Processing with Transformers. Revised Edition. 2-ге вид. O'Reilly Media, 2023. 406 с. | Bashir, S., & Galadanci, M. I. M. (б. д.). AUTOMATIC CODE GENERATION FROM UML DIAGRAMS: THE STATE-OF-THE-ART. *Science World Journal*, *Vol 13*(4).

6. 4.14 Linear maps ‣ Chapter 4 Linear algebra ‣ MATH0005 Algebra 1‣ Chapter 4 Linear algebra ‣ MATH0005 Algebra 1. UCL - London's Global University. URL: https://www.ucl.ac.uk/~ucahmto/0005_2021/Ch4.S14.html (дата звернення: 05.03.2024).

7. How does tokenization help in training a neural network to understand the meaning of words? - EITCA Academy. EITCA Academy. URL: https://eitca.org/artificial-intelligence/eitc-ai-tff-tensorflow-fundamentals/natural-language-processing-with-tensorflow/tokenization/examination-review-tokenization/how-does-tokenization-help-in-training-a-neural-network-to-understand-the-meaning-of-words/ (дата звернення: 07.04.2024).

8. О. КАРАТАЄВ, І. ШУБІН. ПРОБЛЕМИ ПОВТОРНОГО ВИКОРИСТАННЯ ЗНАНЬ У ПРОЦЕСІ ПРОЄКТУВАННЯ ПРОГРАМНИХ

СИСТЕМ. Innovative technologies and scientific solutions for industries. 2023. No. 2, № (24). С. 62. URL: https://doi.org/10.30837/ITSSI.2023.24.0.

9. Patterns of Enterprise Application Architecture. Book by Martin Fowler. 2002 – 41 с.

10. Software Architecture / ред.: S. Biffl та ін. Cham : Springer International Publishing, 2021. URL: https://doi.org/10.1007/978-3-030-86044-8 (дата звернення: 01.02.2024).

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ ЗА НАУКОВИМИ НАПРЯМАМИ КЕРІВНИКА ТА НАУКОВЦІВ КАФЕДРИ ПРОГРАМНОЇ ІНЖЕНЕРІЇ

8. О. КАРАТАЄВ, І. ШУБІН. ПРОБЛЕМИ ПОВТОРНОГО ВИКОРИСТАННЯ ЗНАНЬ У ПРОЦЕСІ ПРОЄКТУВАННЯ ПРОГРАМНИХ СИСТЕМ. Innovative technologies and scientific solutions for industries. 2023. No. 2, № (24). С. 62. URL: https://doi.org/10.30837/ITSSI.2023.24.0.